Mastering Docker

Third Edition

Unlock new opportunities using Docker's most advanced features

Packt

www.packt.com

Russ McKendrick and Scott Gallagher

Mastering Docker
Third Edition

Unlock new opportunities using Docker's most advanced features

Russ McKendrick
Scott Gallagher

Packt

BIRMINGHAM - MUMBAI

Mastering Docker
Third 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
Content Development Editor: Sharon Raj
Technical Editor: Mohit Hassija
Copy Editor: Safis Editing
Project Coordinator: Drashti Panchal
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Graphics: Tom Scaria
Production Coordinator: Shantanu Zagade

First published: December 2015
Second edition: July 2017
Third edition: October 2018

Production reference: 2061118

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

ISBN 978-1-78961-660-6

`www.packtpub.com`

Why subscribe?

Packt.com

Contributors

About the authors

Russ McKendrick is an experienced system administrator who has been working in IT and related industries for over 25 years. During his career, he has had varied responsibilities, from looking after an entire IT infrastructure to providing first-line, second-line, and senior support in both client-facing and internal teams for large organizations.

Russ supports open source systems and tools on public and private clouds at Node4 Limited, where he is the Practice Manager (SRE and DevOps).

> *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.*

Scott Gallagher has been fascinated with technology since he played Oregon Trail in elementary school. His love for it continued through middle school as he worked on more Apple IIe computers. In high school, he learned how to build computers and program in BASIC. His college years were all about server technologies such as Novell, Microsoft, and Red Hat. After college, he continued to work on Novell, all the while maintaining an interest in all technologies. He then moved on to manage Microsoft environments and, eventually, what he was most passionate about Linux environments. Now, his focus is on Docker and cloud environments.

About the reviewer

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 because of habit. While reviewing this book, Paul has been working for Healthy Performance Ltd, helping to apply cutting-edge technology to a cutting-edge approach to well-being.

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

Docker has been a game-changer when it comes to how modern applications are deployed and architectured. It has now grown into a key driver of innovation beyond system administration, and it has an impact on the world of web development and more. 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 also cover basic topics, such as building, managing, and storing images, along with best practices to make you confident 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 sense of exactly what's possible with Docker and how seamlessly it fits into your local workflow, as well as to highly available public cloud platforms and other tools.

Who this book is for

If you are an IT professional and recognize Docker's importance in innovation in everything from system administration to web development, but aren't sure 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`, *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 7`, *Docker Machine*, looks at Docker Machine, a tool that allows you to launch and manage Docker hosts on various platforms.

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

`Chapter 9`, *Docker and Kubernetes*, takes a look at Kubernetes. Like Docker Swarm, you can use Kubernetes to create and manage clusters that run your container-based applications.

`Chapter 10`, *Running Docker in Public Clouds*, looks at using the tools provided by Docker to launch a Docker Swarm cluster in Amazon Web Services, and also Microsoft Azure. We will then look at the container solutions offered by Amazon Web Services, Microsoft Azure, and Google Cloud.

`Chapter 11`, *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 12`, *Docker Security*, takes a look at Docker security. We will cover everything from the Docker host, to how you launch your images, to where you get them from, and also the contents of your images.

`Chapter 13`, *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 14`, *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 or above, running one of the following OSes:

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

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

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.packt.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 & Errata**.
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-Third-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!

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/9781789616606_ColorImages.pdf`.

Code in Action

Visit the following link to check out videos of the code being run:
`http://bit.ly/2PUB9ww`

Conventions used

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

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The first file is `nginx.conf`, which contains a basic nginx configuration file."

A block of code is set as follows:

```
user nginx;
worker_processes 1;

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

events {
    worker_connections 1024;
}
```

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

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

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: "Upon clicking on **Create**, you will be taken to a screen similar to the next screenshot."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: 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.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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

1
Docker Overview

Welcome to *Mastering Docker, Third 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 with the basics, so that subsequent chapters don't feel as heavy. By the end of the book, you should be a Docker master, and will be 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 high-level topics:

- Understanding Docker
- The differences between dedicated hosts, virtual machines, and Docker
- 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

Check out the following video to see the Code in Action:

`http://bit.ly/2NXf3rd`

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 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 difficult to achieve. Everyone has their way of working and their own personal preferences—enforcing consistency across multiple platforms is difficult enough when there is a single engineer 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 easily wrap their code in a container that they have either defined themselves, or created as a Dockerfile while working alongside a sys-admin 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*.

They can continue to use their chosen 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; in fact, 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, *We need to deploy Application 2 on the same servers that are running Application 1*. On the face of it, this is no problem—you can tweak your Puppet or Chef configuration to add new users, vhosts, pull the new code down, and so on. However, you notice that Application 2 requires a higher version of the software that 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 backwards compatible.

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

1. Ask for more servers? While this traditionally is probably the safest technical solution, it does not automatically mean that there will be the budget for additional resources.
2. 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 also reducing the redundancy for Application 1: there was probably a reason why you were running three web and two database servers in the first place.
3. 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 actually 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 simply 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 described previously, as they have both developers and operators; 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 aforementioned risk, along with 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 introduce 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 feature, 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

Let me start by saying that 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, all of a sudden, your testing process and procedures are a lot more flexible, and you can continue to reuse the same environment, rather than redeploying or reimaging servers for the next set of testing.

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

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

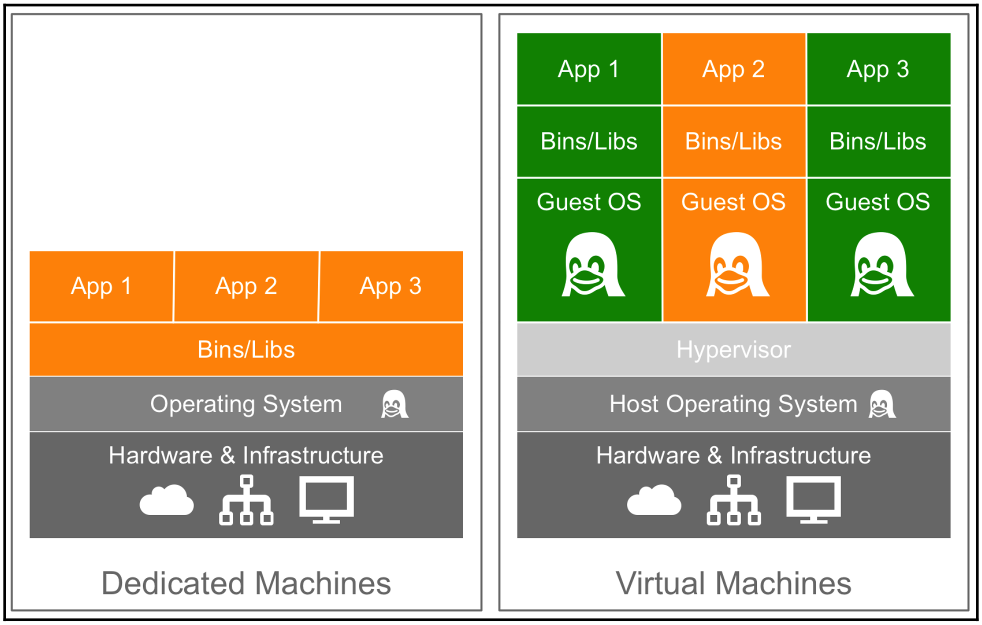
The differences between dedicated hosts, virtual machines, and Docker

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

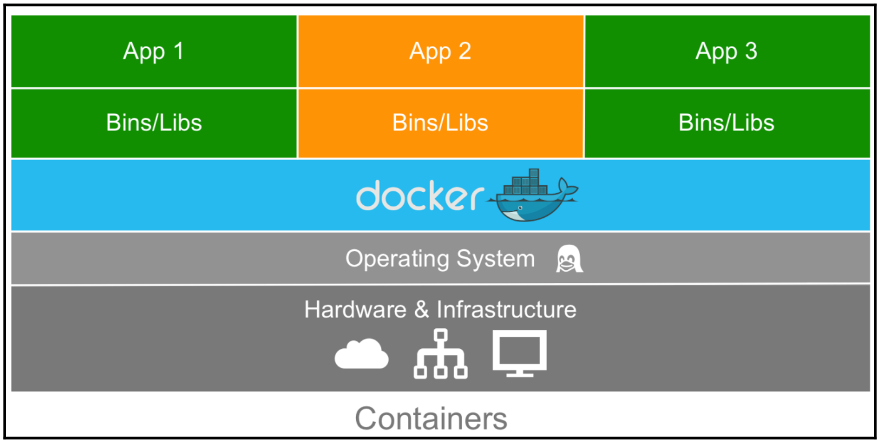
Docker is a container management system that helps us easily manage **Linux Containers (LXC)** in an easier and universal fashion. This lets you create images in virtual environments on your laptop and run commands against them. The actions you perform to the containers, running in these environments 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.

The following diagram demonstrates the difference between a dedicated, bare-metal server and a server running virtual machines:



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



This diagram gives us a lot of insight into the biggest 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 to actually 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 largest 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 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

In addition, you can run them on public clouds, such as Amazon Web Services, Microsoft Azure, and DigitalOcean, to name a few. With each of the various types of installers listed previously, Docker actually operates in different ways on the operating system. For example, Docker runs natively on Linux, so if you are using Linux, then how Docker runs on your system is pretty straightforward. 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 (Ubuntu 18.04)

As already mentioned, this is the most straightforward installation out of the three systems we will be looking at. 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:

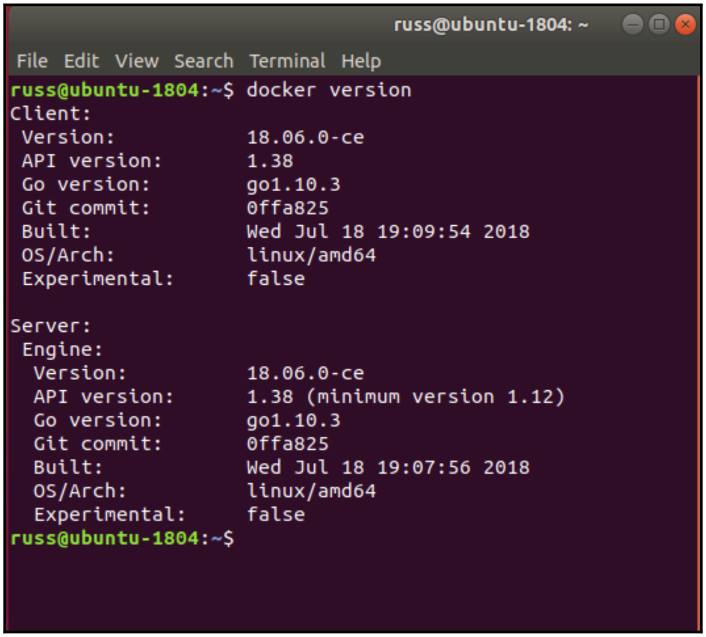
```
$ sudo usermod -aG docker username
```

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 18.06.

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

```
$ docker version
```

You should see something similar to the following output:



```
russ@ubuntu-1804:~$ docker version
Client:
 Version:           18.06.0-ce
 API version:       1.38
 Go version:        go1.10.3
 Git commit:        0ffa825
 Built:             Wed Jul 18 19:09:54 2018
 OS/Arch:           linux/amd64
 Experimental:      false

Server:
 Engine:
  Version:          18.06.0-ce
  API version:      1.38 (minimum version 1.12)
  Go version:       go1.10.3
  Git commit:       0ffa825
  Built:            Wed Jul 18 19:07:56 2018
  OS/Arch:          linux/amd64
  Experimental:     false
russ@ubuntu-1804:~$
```

There are two supporting tools 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 these tools in later chapters, we should install them now. The first tool is **Docker Machine**. To install this, 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/machine/releases/`. At the time of writing, the version was 0.15.0—update the version number in the commands in the following code block with whatever the latest version is when you install it:

```
$ MACHINEVERSION=0.15.0
$ curl -L
https://github.com/docker/machine/releases/download/v$MACHINEVERSION/docker
-machine-$(uname -s)-$(uname -m) >/tmp/docker-machine
$ chmod +x /tmp/docker-machine
$ sudo mv /tmp/docker-machine /usr/local/bin/docker-machine
```

To download and install the next and final tool, **Docker Compose**, run the following commands, again checking that you are running the latest version by visiting the releases page at `https://github.com/docker/compose/releases/`:

```
$ COMPOSEVERSION=1.22.0
$ 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 confirm the versions of the software is correctly:

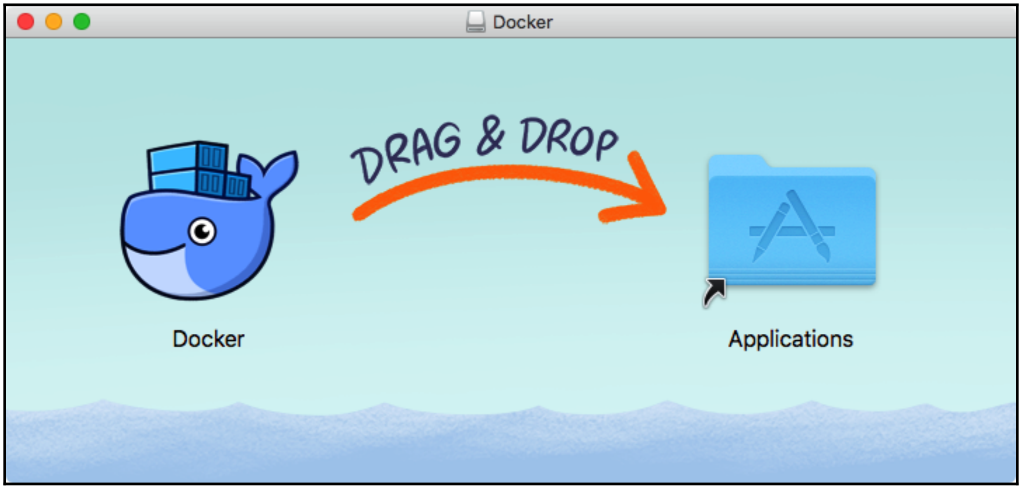
```
$ docker-machine version
$ docker-compose version
```

Installing Docker on macOS

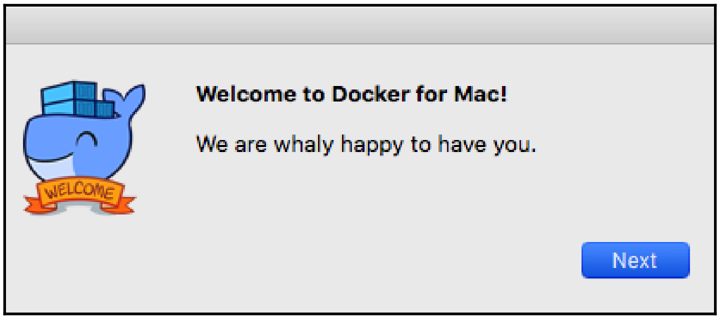
Unlike the command-line Linux installation, Docker for Mac has a graphical installer.

Before downloading, you should make sure that you are running Apple macOS Yosemite 10.10.3 or above. If you are running an older version, all is not lost; you can still run Docker. Refer to the other older operating systems section of this chapter.

You can download the installer from the Docker store, at `https://store.docker.com/editions/community/docker-ce-desktop-mac`. Just click on the **Get Docker** link. 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:



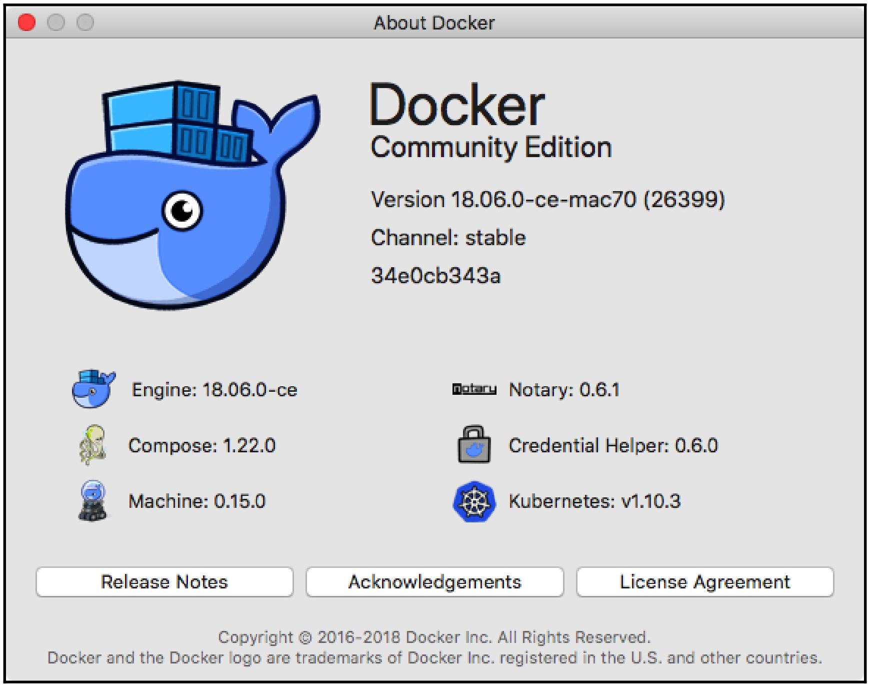
Once you have dragged the Docker icon to your **Applications** folder, double-click on it and you will be asked whether you want to open the application you have downloaded. Clicking **Yes** will open the Docker installer, showing the following:



Welcome to Docker for Mac!

We are whaly happy to have you.

Click on **Next** and follow the onscreen instructions. Once it is installed and started, you should see a Docker icon in the top-left icon bar on your screen. Clicking on the icon and selecting **About Docker** should show you something similar to the following:



About Docker

Docker
Community Edition

Version 18.06.0-ce-mac70 (26399)

Channel: stable

34e0cb343a

Engine: 18.06.0-ce

Notary: 0.6.1

Compose: 1.22.0

Credential Helper: 0.6.0

Machine: 0.15.0

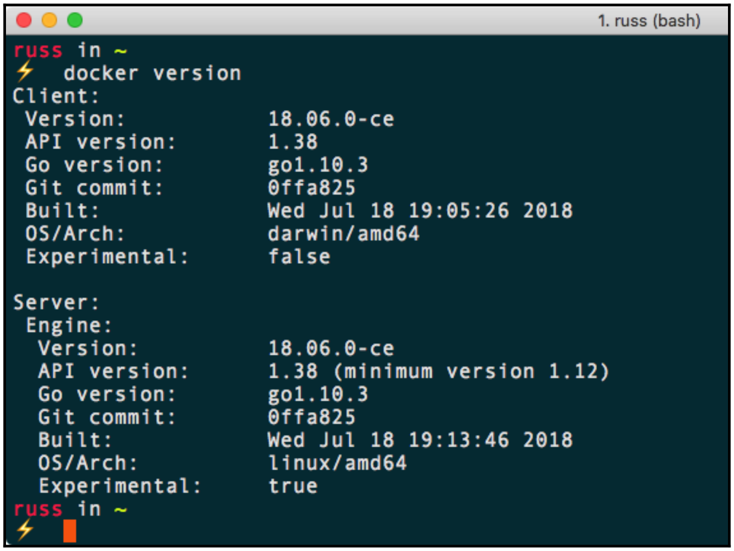
Kubernetes: v1.10.3

Copyright © 2016-2018 Docker Inc. All Rights Reserved.
Docker and the Docker logo are trademarks of Docker Inc. registered in the U.S. and other countries.

You can also open a Terminal window. Run the following command, just as we did in the Linux installation:

```
$ docker version
```

You should see something similar to the following Terminal output:



```
russ in ~
⚡  docker version
Client:
 Version:           18.06.0-ce
 API version:       1.38
 Go version:        go1.10.3
 Git commit:        0ffa825
 Built:             Wed Jul 18 19:05:26 2018
 OS/Arch:           darwin/amd64
 Experimental:      false

Server:
 Engine:
  Version:          18.06.0-ce
  API version:      1.38 (minimum version 1.12)
  Go version:       go1.10.3
  Git commit:       0ffa825
  Built:            Wed Jul 18 19:13:46 2018
  OS/Arch:          linux/amd64
  Experimental:     true
russ in ~
⚡
```

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
$ docker-machine version
```

Installing Docker on Windows 10 Professional

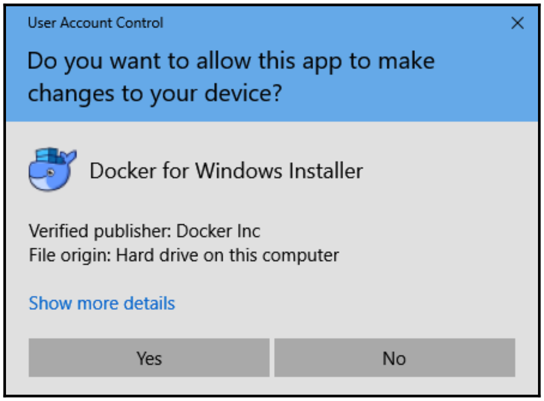
Like Docker for Mac, Docker for Windows uses a graphical installer.

TIP

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 other 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.

You can download the Docker for Windows installer from the Docker store at `https://store.docker.com/editions/community/docker-ce-desktop-windows/`. Just click on the **Get Docker** button to download the installer. Once it's downloaded, run the MSI package and you will be greeted with the following:



User Account Control

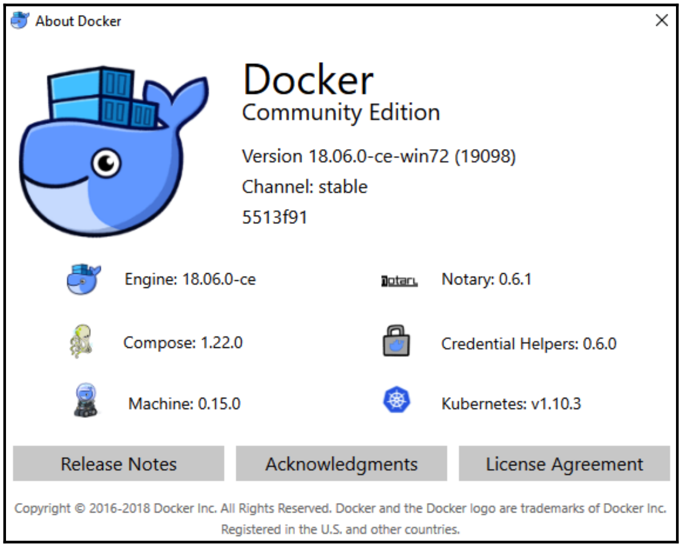
Do you want to allow this app to make changes to your device?

Docker for Windows Installer

Verified publisher: Docker Inc
File origin: Hard drive on this computer

Click on **Yes**, and then follow the onscreen prompts, which will go through not only installing Docker, but also enabling Hyper-V, if you do not already have it enabled.

Once it's installed, 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:



About Docker

Docker
Community Edition

Version 18.06.0-ce-win72 (19098)
Channel: stable
5513f91

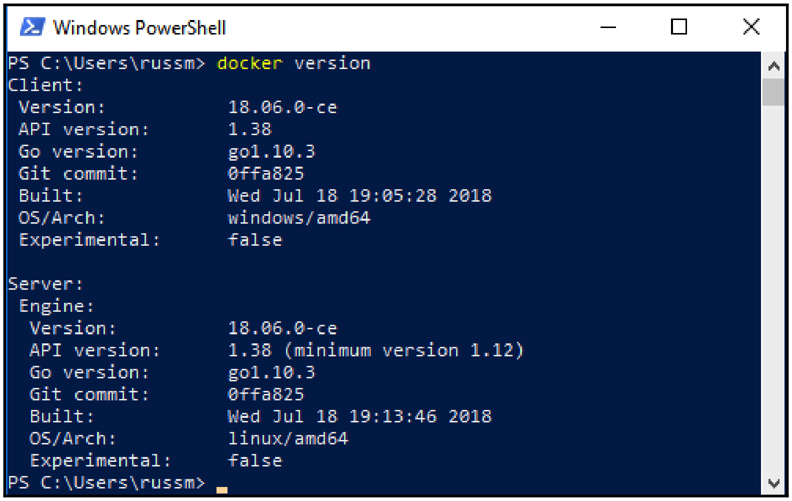
Engine: 18.06.0-ce
Notary: 0.6.1
Compose: 1.22.0
Credential Helpers: 0.6.0
Machine: 0.15.0
Kubernetes: v1.10.3

Copyright © 2016-2018 Docker Inc. All Rights Reserved. Docker and the Docker logo are trademarks of Docker Inc. Registered in the U.S. and other countries.

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\russm> docker version
Client:
 Version:           18.06.0-ce
 API version:       1.38
 Go version:        go1.10.3
 Git commit:        0ffa825
 Built:             Wed Jul 18 19:05:28 2018
 OS/Arch:           windows/amd64
 Experimental:      false

Server:
 Engine:
  Version:          18.06.0-ce
  API version:      1.38 (minimum version 1.12)
  Go version:       go1.10.3
  Git commit:       0ffa825
  Built:            Wed Jul 18 19:13:46 2018
  OS/Arch:          linux/amd64
  Experimental:     false
PS C:\Users\russm>
```

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
$ docker-machine version
```

Again, 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. This will be covered in greater detail later in 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 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 running a small, lightweight operating system based on Alpine Linux. The virtual machine runs using Docker's own libraries, which connect to the built-in hypervisor for your chosen environment.

For macOS, this is the built-in Hypervisor.framework, and for Windows, 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.

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 VirtualBox, see `https://www.virtualbox.org/`.

For more information on **Docker Toolbox**, see the project's website at `https://www.docker.com/products/docker-toolbox/`, where you can also download the macOS and Windows installers.

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.

The Docker command-line client

Now that we have Docker installed, let's look at some Docker commands that you should be familiar with already. 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
> Docker has restructured their command-line client into more logical groupings of commands, as the number of features provided by the client grows quickly and commands start to cross over each other. Throughout this book, we will be using the new structure.

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. For further help with a particular command, you can run the following:

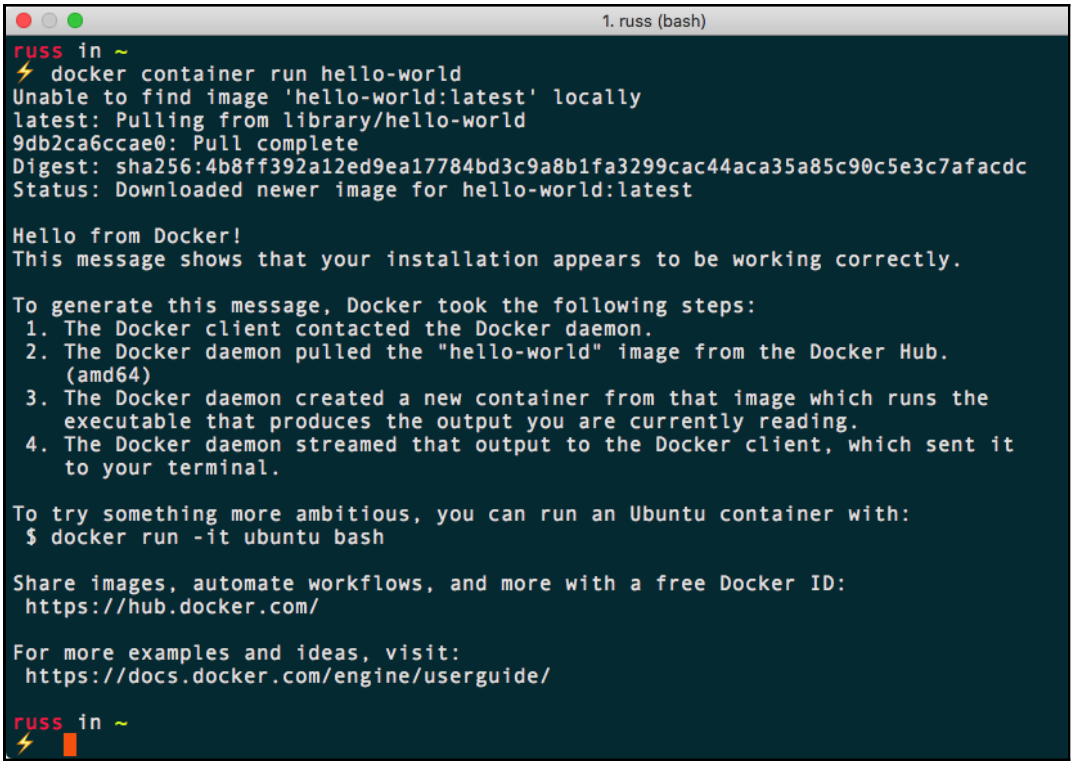
```
$ docker <COMMAND> --help
```

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:



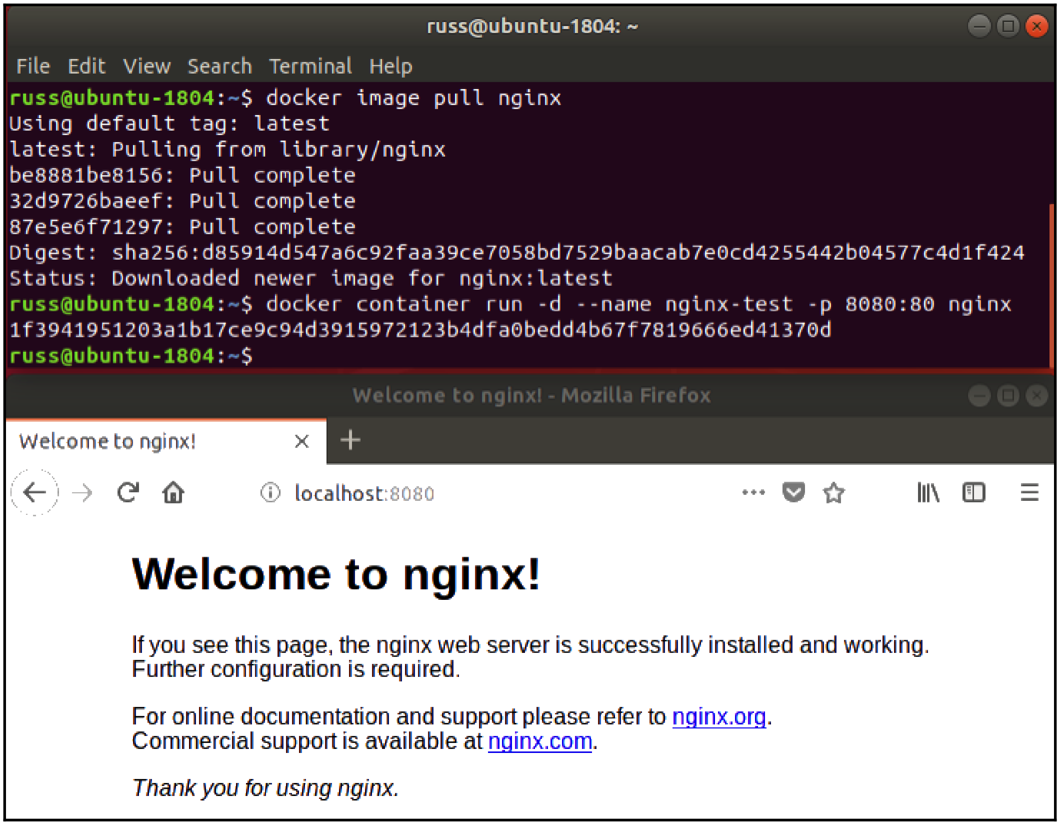
```
russ in ~
⚡ docker container run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
9db2ca6ccae0: Pull complete
Digest: sha256:4b8ff392a12ed9ea17784bd3c9a8b1fa3299cac44aca35a85c90c5e3c7afacdc
Status: Downloaded newer image for hello-world:latest

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/engine/userguide/

russ in ~
⚡
```

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

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

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:



```
russ@ubuntu-1804:~$ docker image pull nginx
Using default tag: latest
latest: Pulling from library/nginx
be8881be8156: Pull complete
32d9726baeef: Pull complete
87e5e6f71297: Pull complete
Digest: sha256:d85914d547a6c92faa39ce7058bd7529baacab7e0cd4255442b04577c4d1f424
Status: Downloaded newer image for nginx:latest
russ@ubuntu-1804:~$ docker container run -d --name nginx-test -p 8080:80 nginx
1f3941951203a1b17ce9c94d3915972123b4dfa0bedd4b67f7819666ed41370d
russ@ubuntu-1804:~$
```

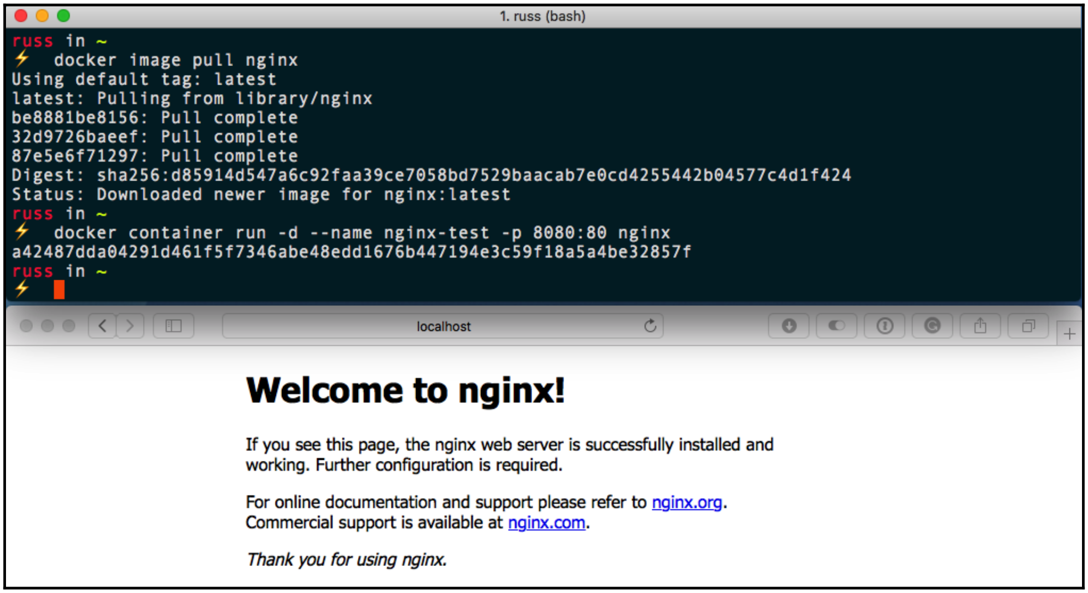
Welcome to nginx!

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.

This is the result on macOS:



```
russ in ~
⚡  docker image pull nginx
Using default tag: latest
latest: Pulling from library/nginx
be8881be8156: Pull complete
32d9726baeef: Pull complete
87e5e6f71297: Pull complete
Digest: sha256:d85914d547a6c92faa39ce7058bd7529baacab7e0cd4255442b04577c4d1f424
Status: Downloaded newer image for nginx:latest
russ in ~
⚡  docker container run -d --name nginx-test -p 8080:80 nginx
a42487dda04291d461f5f7346abe48edd1676b447194e3c59f18a5a4be32857f
russ in ~
⚡
```

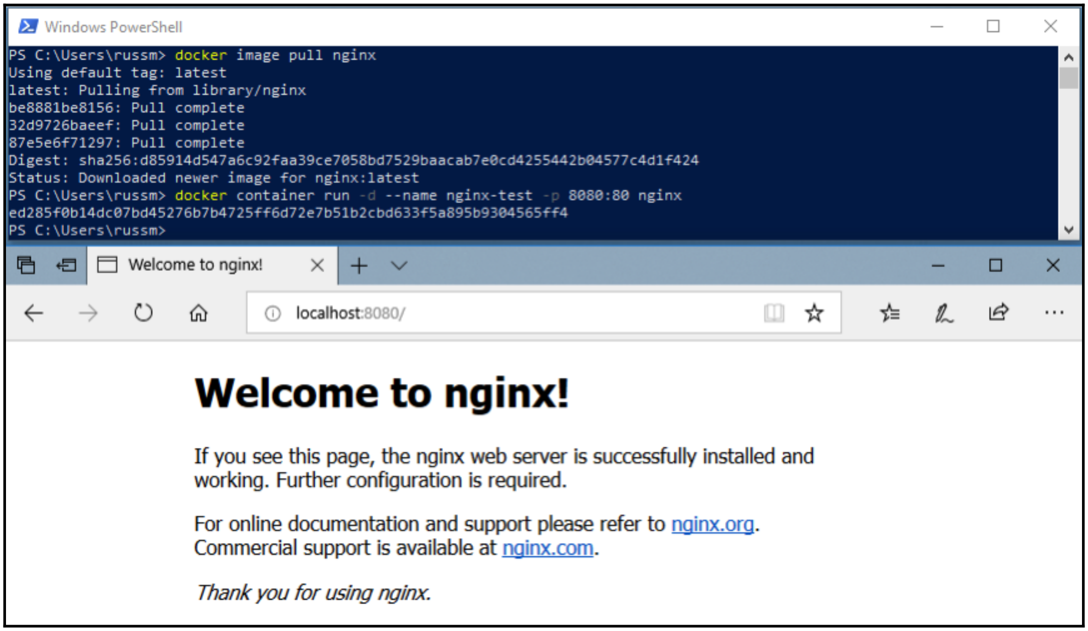
Welcome to nginx!

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.

And this is how it looks on Windows:



```
PS C:\Users\russm> docker image pull nginx
Using default tag: latest
latest: Pulling from library/nginx
be8881be8156: Pull complete
32d9726baeef: Pull complete
87e5e6f71297: Pull complete
Digest: sha256:d85914d547a6c92faa39ce7058bd7529baacab7e0cd4255442b04577c4d1f424
Status: Downloaded newer image for nginx:latest
PS C:\Users\russm> docker container run -d --name nginx-test -p 8080:80 nginx
ed285f0b14dc07bd45276b7b4725ff6d72e7b51b2cbd633f5a895b9304565ff4
PS C:\Users\russm>
```

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.

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 also a very different experience on each platform. Traditionally, this has been one of the reasons for the difference in local development environments.

Docker and the container ecosystem

If you have been following the rise of Docker and containers, you will have noticed that, over the period of 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 do 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 other companies that sprung up to support all sorts of container technologies.

So, let's try and unpick 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 CE and Docker EE**: This is the core collection of free-to-use and commercially supported Docker tools built on top of the open source components.
- **Docker, Inc.**: This is the company founded to support and develop the core Docker tools.

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 two 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 allows you to build distributed systems using the same raft consensus algorithm as Docker Swarm.
- **LinuxKit** is a framework that allows you to build and compile a small portable Linux operating system for running containers.
- **InfraKit** is a collection of tools that you can use to define 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 which are maintained by Docker, Inc. We will go a little more into these projects in our final chapter.

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 the Docker **Enterprise Edition** (**EE**) and the Docker **Community Edition** (**CE**). We will be using Docker CE throughout this book.

From September 2018, the release cycle for the stable version of Docker CE will be biannual, which means that it will have a seven-month maintenance cycle. This means that you have plenty of time to review and plan any upgrades. At the time of writing, the current timetable for Docker CE releases is:

- **Docker 18.06 CE**: This is the last of the quarterly Docker CE releases, released July 18th 2018.
- **Docker 18.09 CE**: This release, due late September/early October 2018, is the first release of the biannual release cycle of Docker CE.
- **Docker 19.03 C**E: The first supported Docker CE of 2019 is scheduled to be released March/April 2019.
- **Docker 19.09 CE**: The second supported release of 2019 is scheduled to be released September/October 2019.

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 also monthly builds of Docker for Mac and Docker for Windows via the Edge channel.

Docker also provides the following tools and services:

- **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 7`, *Docker Machine*.
- **Docker Hub**: A repository for your Docker images, covered in the next three chapters.
- **Docker Store**: A storefront for official Docker images and plugins as well as licensed products. Again, we will cover this in the next three chapters.
- **Docker Swarm**: A multi-host-aware orchestration tool, covered in detail in `Chapter 8`, *Docker Swarm*.
- **Docker for Mac**: We have covered Docker for Mac in this chapter.

- **Docker for Windows**: We have covered Docker for Windows in this chapter.
- **Docker for Amazon Web Services**: A best-practice Docker Swarm installation that targets AWS, covered in `Chapter 10`, Running *Docker in Public Clouds*.
- **Docker for Azure**: A best-practice Docker Swarm installation that targets Azure, covered in `Chapter 10`, Running *Docker in Public Clouds.*

Docker, Inc.

Docker, Inc. is the company formed to develop Docker CE and Docker EE. It also provides SLA-based support services for Docker EE. Finally, they are offer consultative services to companies who wish take their existing applications and containerize them as part of Docker's **Modernize Traditional Apps** (**MTA**) program.

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 chapters, 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 for Mac and Docker for 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. How many months are in the support lifecycle for a stable Docker CE release?
5. Which command would you run to find out more information on the Docker container subset of commands?

Further reading

In this chapter we have mentioned the following hypervisors:

- macOS Hypervisor framework: `https://developer.apple.com/reference/hypervisor/`
- Hyper-V: `https://www.microsoft.com/en-gb/cloud-platform/server-virtualization`

We referenced the following blog posts from Docker:

- Docker CLI restructure blog post: `https://blog.docker.com/2017/01/whats-new-in-docker-1-13/`
- Docker Extended Support Announcement: `https://blog.docker.com/2018/07/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/docker/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`

- SwarmKit: `https://github.com/docker/swarmkit`
- LinuxKit: `https://github.com/linuxkit/linuxkit`
- InfraKit: `https://github.com/docker/infrakit`
- The OCI specification: `https://github.com/opencontainers/runtime-spec/`

Finally, the meme mentioned at the start of the chapter can be found here:

- *Worked fine in Dev, Ops problem now* - `http://www.developermemes.com/2013/12/13/worked-fine-dev-ops-problem-now/`

2
Building Container Images

In this chapter, we are going to get you started building container images. We will look at several different ways with which you can define and build your images using the tools built into Docker. We will cover the following topics:

- Introducing the Dockerfile
- Building container images using a Dockerfile
- Building container images using an existing container
- Building container images from scratch
- Building container images using environmental variables
- Building container images using multi-stage builds

Technical requirements

In the previous chapter, we installed Docker on the following target operating systems:

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

In this chapter, we will be using our Docker installation to build images. 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 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.

A full copy of the code used in this chapter can be found at: `https://github.com/PacktPublishing/Mastering-Docker-Third-Edition/tree/master/chapter02`

Check out the following video to see the Code in Action:

`http://bit.ly/2D0JA6v`

Introducing the Dockerfile

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

A **Dockerfile** is simply a plain text file that contains a set of user-defined instructions. When the 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 like the following:

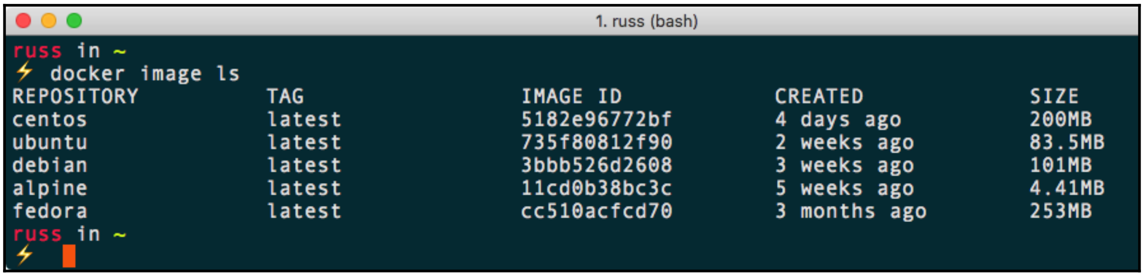
```
FROM alpine:latest
LABEL maintainer="Russ McKendrick <russ@mckendrick.io>"
LABEL description="This example Dockerfile installs NGINX."
RUN apk add --update nginx && \
    rm -rf /var/cache/apk/* && \
    mkdir -p /tmp/nginx/

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 to working 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 (see the following section), 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 it offers proactive protection dozens of potential zero-day and other vulnerabilities.

Alpine Linux, due both to 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:



```
russ in ~
⚡ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
centos              latest              5182e96772bf        4 days ago          200MB
ubuntu              latest              735f80812f90        2 weeks ago         83.5MB
debian              latest              3bbb526d2608        3 weeks ago         101MB
alpine              latest              11cd0b38bc3c        5 weeks ago         4.41MB
fedora              latest              cc510acfcd70        3 months ago        253MB
russ in ~
⚡
```

As you can see from the Terminal output, Alpine Linux weighs in at only 4.41 MB, as opposed to the biggest image, which is Fedora, at 253 MB. A bare-metal installation of Alpine Linux comes in at around 130 MB, which is still almost half the size of the Fedora container image.

Reviewing the Dockerfile in depth

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

- `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 already mentioned, we are using Alpine Linux, so we simply have to put 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 have to 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 to filter just the labels:

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

In our example Dockerfile, we add two labels:

1. maintainer="Russ McKendrick <russ@mckendrick.io>" adds a label which helps identify, to the end user of the image, who is maintaining it
2. 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 our 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.

We are using the `&&` operator to move on to the next command if the previous command was successful. To make it more obvious which commands we are running, we are also using `\` so that we can split the command over multiple lines, making it easy to read.

The next command in our chain removes any temporary files and so on 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 to be considered inefficient as it can add to the overall size of the image, which for the most part we should try to avoid. There are some valid use cases for this, which we will look at later in the chapter. 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; however, there are some important differences. 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 contains a basic 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 most simple virtual host that we can configure, and has 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 case, it looks like the following:

```
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 puts the resulting folders and files at the path we tell 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 http://www.myremotesource.com/files/html.tar.gz /usr/share/nginx/
```

The preceding command line would download `html.tar.gz` from `http://www.myremotesource.com/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, meaning that the file would have to be added before the `RUN` instruction, so that we could 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, but instead, 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
```

ENTRYPOINT and CMD

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

For reference, if you think of some of the CLI commands you might use, you have to 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, but be sure to use a command that keeps the container alive. In our case, 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, as we have defined that as our ENTRYPOINT, and then whatever we have as the 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 the following:

```
$ 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. As we have the nginx binary as our entry point, any command we pass overrides the CMD that had 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 and then the process would stop. We will look at this example later in this chapter, once we have built our image.

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 has to 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 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 tell them, 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 environment variables 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 the best practices of writing our own Dockerfiles:

- You should try to get into the habit of using a `.dockerignore` file. We will cover the `.dockerignore` file in the next section; it will seem very familiar if you are used to using a `.gitignore` file. It will essentially ignore the items you have 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 install per image. One of the biggest goals you want to achieve while building your images is to keep them as small 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 best practice to use a new container to run that application in.
- Keep things simple; over-complicating your Dockerfile will add bloat and also potentially cause you issues further 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 container 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 time by completing the hard work; 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 could use while using the `docker build` command. So, let's use the always handy `--help` switch on the `docker image build` command to view all that we can do:

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

There are then 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 of 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 networking configuration of the container launched to build our image.

Typically, you don't use the `--file` or `-f` switch, as 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 also worth mentioning that, while you are able to pass additional environment variables 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` as, by default, all 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.

Using a Dockerfile to build a container image

The first method that we are going to look at for use in building your base container images is by 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 this image. The first way would be by specifying the `-f` switch when we use the `docker image build` command. We will also utilize the `-t` switch to give the new image a unique name:

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

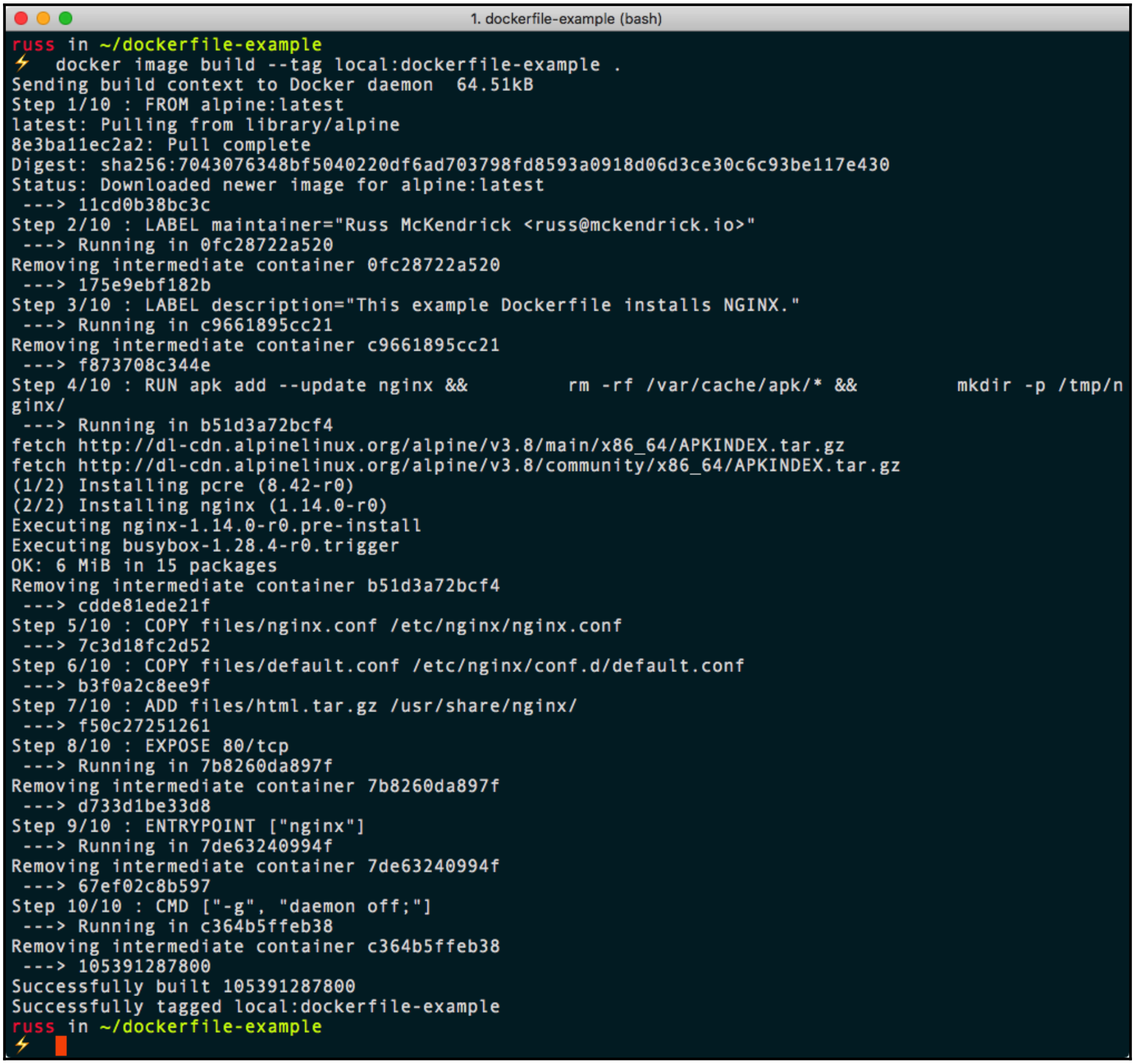
Now, `<REPOSITORY>` is typically the username you signed up for on Docker Hub. We will look at this in more detail in `Chapter 3`, *Storing and Distributing Images*; for now, we will be using `local`, and `<TAG>` is the unique container value you want to provide. Typically, this will be a version number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

```
$ docker image build --file /path/to/your/dockerfile --tag
local:dockerfile-example .
```

Typically, the `--file` switch isn't used, and it can be a little tricky when you have other files that need to be included with the new image. An easier way to do the build is to place the Dockerfile in a separate folder by itself, along with any other file that you will be injecting into your image using the `ADD` or `COPY` instructions:

```
$ 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:

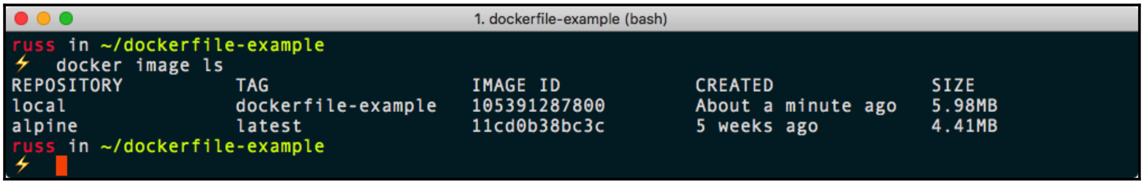


```
russ in ~/dockerfile-example
⚡  docker image build --tag local:dockerfile-example .
Sending build context to Docker daemon  64.51kB
Step 1/10 : FROM alpine:latest
latest: Pulling from library/alpine
8e3ba11ec2a2: Pull complete
Digest: sha256:7043076348bf5040220df6ad703798fd8593a0918d06d3ce30c6c93be117e430
Status: Downloaded newer image for alpine:latest
 ---> 11cd0b38bc3c
Step 2/10 : LABEL maintainer="Russ McKendrick <russ@mckendrick.io>"
 ---> Running in 0fc28722a520
Removing intermediate container 0fc28722a520
 ---> 175e9ebf182b
Step 3/10 : LABEL description="This example Dockerfile installs NGINX."
 ---> Running in c9661895cc21
Removing intermediate container c9661895cc21
 ---> f873708c344e
Step 4/10 : RUN apk add --update nginx &&         rm -rf /var/cache/apk/* &&         mkdir -p /tmp/n
ginx/
 ---> Running in b51d3a72bcf4
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
(1/2) Installing pcre (8.42-r0)
(2/2) Installing nginx (1.14.0-r0)
Executing nginx-1.14.0-r0.pre-install
Executing busybox-1.28.4-r0.trigger
OK: 6 MiB in 15 packages
Removing intermediate container b51d3a72bcf4
 ---> cdde81ede21f
Step 5/10 : COPY files/nginx.conf /etc/nginx/nginx.conf
 ---> 7c3d18fc2d52
Step 6/10 : COPY files/default.conf /etc/nginx/conf.d/default.conf
 ---> b3f0a2c8ee9f
Step 7/10 : ADD files/html.tar.gz /usr/share/nginx/
 ---> f50c27251261
Step 8/10 : EXPOSE 80/tcp
 ---> Running in 7b8260da897f
Removing intermediate container 7b8260da897f
 ---> d733d1be33d8
Step 9/10 : ENTRYPOINT ["nginx"]
 ---> Running in 7de63240994f
Removing intermediate container 7de63240994f
 ---> 67ef02c8b597
Step 10/10 : CMD ["-g", "daemon off;"]
 ---> Running in c364b5ffeb38
Removing intermediate container c364b5ffeb38
 ---> 105391287800
Successfully built 105391287800
Successfully tagged local:dockerfile-example
russ in ~/dockerfile-example
⚡
```

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

```
$ docker image ls
```

As you can see from the following Terminal output, my image size is 5.98 MB:



```
russ in ~/dockerfile-example
⚡  docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED              SIZE
local               dockerfile-example  105391287800        About a minute ago   5.98MB
alpine              latest              11cd0b38bc3c        5 weeks ago          4.41MB
russ in ~/dockerfile-example
⚡
```

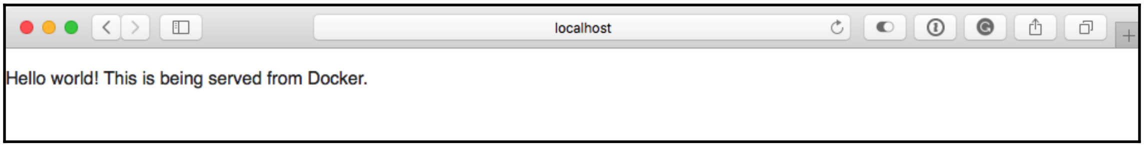
You can launch a container with your newly built image by running this command:

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

This will launch a container called `dockerfile-example`, you can check it is running using the following command:

```
$ docker container ls
```

Opening your browser and going to `http://localhost:8080/` should show you an extremely simple webpage that looks like the following:

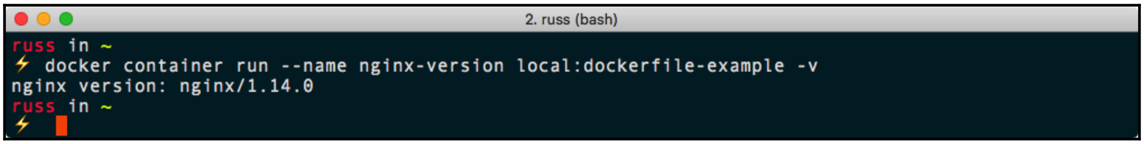


Hello world! This is being served from Docker.

Next up, we can quickly run a few of the commands mentioned in the previous section of the chapter, starting with the following:

```
$ 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 1.14.0:

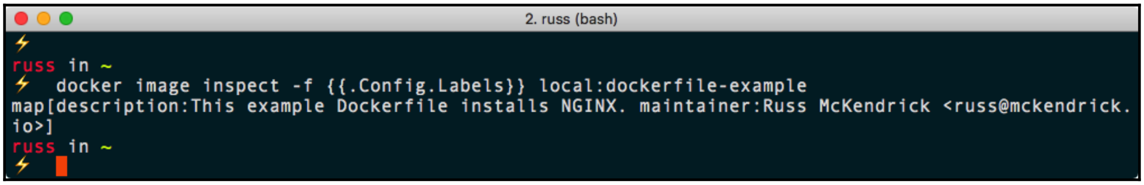


```
russ in ~
⚡  docker container run --name nginx-version local:dockerfile-example -v
nginx version: nginx/1.14.0
russ in ~
⚡
```

The next command we can look at running, now that we have our first image built, displays the labels that we embedded at build time. To view this information run the following:

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

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



```
⚡
russ in ~
⚡  docker image inspect -f {{.Config.Labels}} local:dockerfile-example
map[description:This example Dockerfile installs NGINX. maintainer:Russ McKendrick <russ@mckendrick.
io>]
russ in ~
⚡
```

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 the 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.

I would only use this method during a prototyping phase to check that the commands I am 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 before, 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:

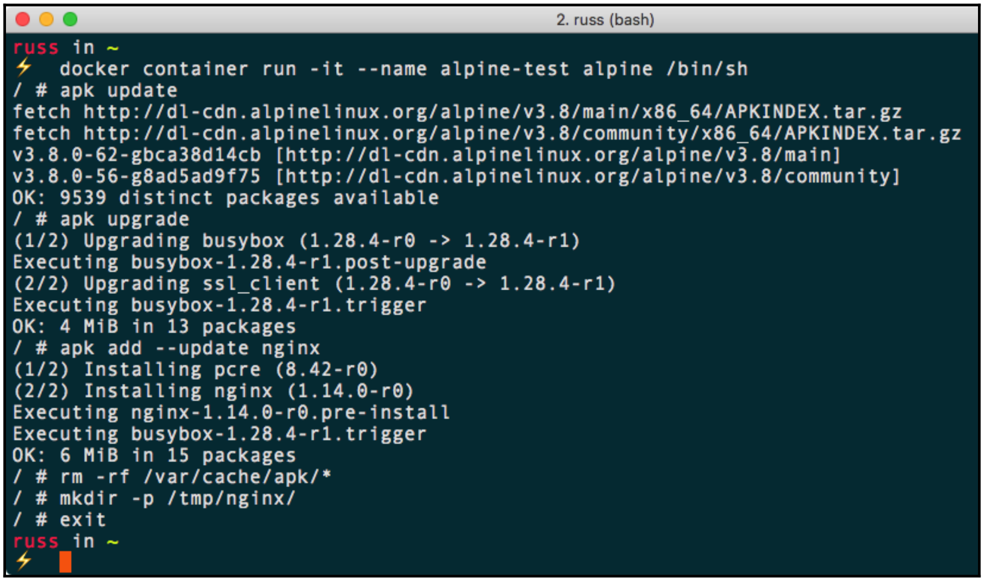
```
$ docker container run -it --name alpine-test alpine /bin/sh
```

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

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 Terminal output as follows:



```
russ in ~
⚡  docker container run -it --name alpine-test alpine /bin/sh
/ # apk update
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
v3.8.0-62-gbca38d14cb [http://dl-cdn.alpinelinux.org/alpine/v3.8/main]
v3.8.0-56-g8ad5ad9f75 [http://dl-cdn.alpinelinux.org/alpine/v3.8/community]
OK: 9539 distinct packages available
/ # apk upgrade
(1/2) Upgrading busybox (1.28.4-r0 -> 1.28.4-r1)
Executing busybox-1.28.4-r1.post-upgrade
(2/2) Upgrading ssl_client (1.28.4-r0 -> 1.28.4-r1)
Executing busybox-1.28.4-r1.trigger
OK: 4 MiB in 13 packages
/ # apk add --update nginx
(1/2) Installing pcre (8.42-r0)
(2/2) Installing nginx (1.14.0-r0)
Executing nginx-1.14.0-r0.pre-install
Executing busybox-1.28.4-r1.trigger
OK: 6 MiB in 15 packages
/ # rm -rf /var/cache/apk/*
/ # mkdir -p /tmp/nginx/
/ # exit
russ in ~
⚡
```

TIP

It is at this point that you should really stop; I do not recommend you use the preceding commands to create and distribute images, apart from the one use case we will cover in the next part of this section.

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
```

Notice how I called my image `broken-container`? As 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.

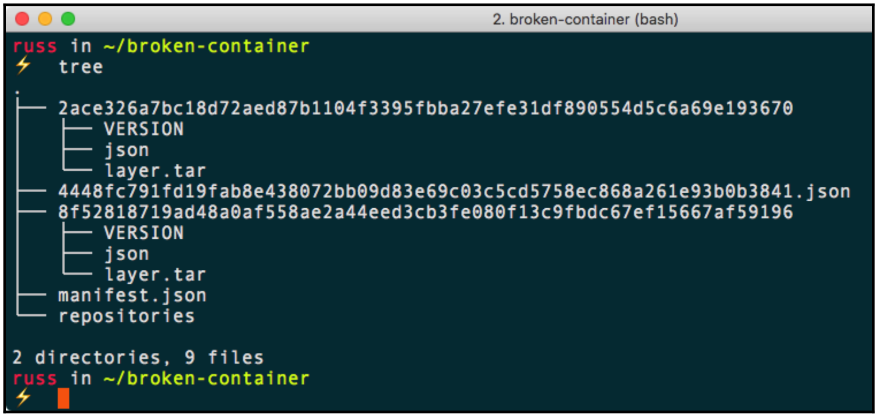
To save the image file, simply run the following command:

```
$ 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 6.6 MB file called `broken-container.tar`. While we have this file, you can uncompress it and have a look around, as you can see from the following structure:



```
russ in ~/broken-container
⚡  tree
.
├── 2ace326a7bc18d72aed87b1104f3395fbba27efe31df890554d5c6a69e193670
│   ├── VERSION
│   ├── json
│   └── layer.tar
├── 4448fc791fd19fab8e438072bb09d83e69c03c5cd5758ec868a261e93b0b3841.json
├── 8f52818719ad48a0af558ae2a44eed3cb3fe080f13c9fbdc67ef15667af59196
│   ├── VERSION
│   ├── json
│   └── layer.tar
├── manifest.json
└── repositories

2 directories, 9 files
russ in ~/broken-container
⚡
```

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

The biggest reason is trust—as 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 during 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, as 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 a consideration anymore.

Building a container image from scratch

So far, we have been using prepared images from the Docker Hub as our base image. It is possible to avoid this altogether (sort of) and roll out your own image 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 the 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, let's look at 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 the 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 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 in the following example:

```
FROM scratch
ADD files/alpine-minirootfs-3.8.0-x86_64.tar.gz /
CMD ["/bin/sh"]
```

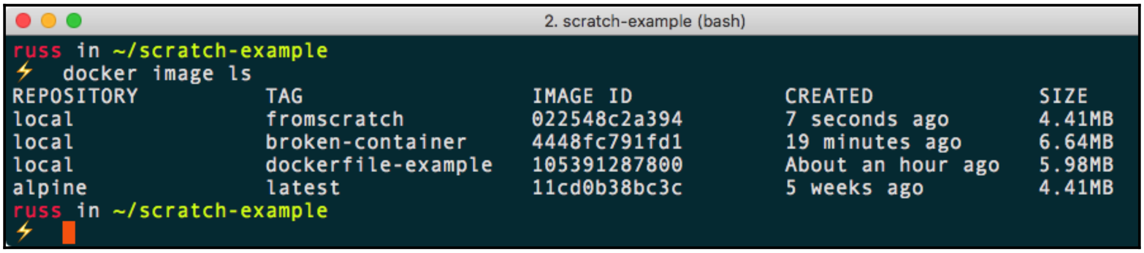
Now that you have your Dockerfile and operating system in a TAR file, you can build your image as you would any other Docker image by running the following command:

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

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 Hug:



2. scratch-example (bash)

```
russ in ~/scratch-example
⚡  docker image ls
REPOSITORY          TAG                  IMAGE ID            CREATED             SIZE
local               fromscratch          022548c2a394        7 seconds ago       4.41MB
local               broken-container     4448fc791fd1        19 minutes ago      6.64MB
local               dockerfile-example   105391287800        About an hour ago   5.98MB
alpine              latest               11cd0b38bc3c        5 weeks ago         4.41MB
russ in ~/scratch-example
⚡  
```

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

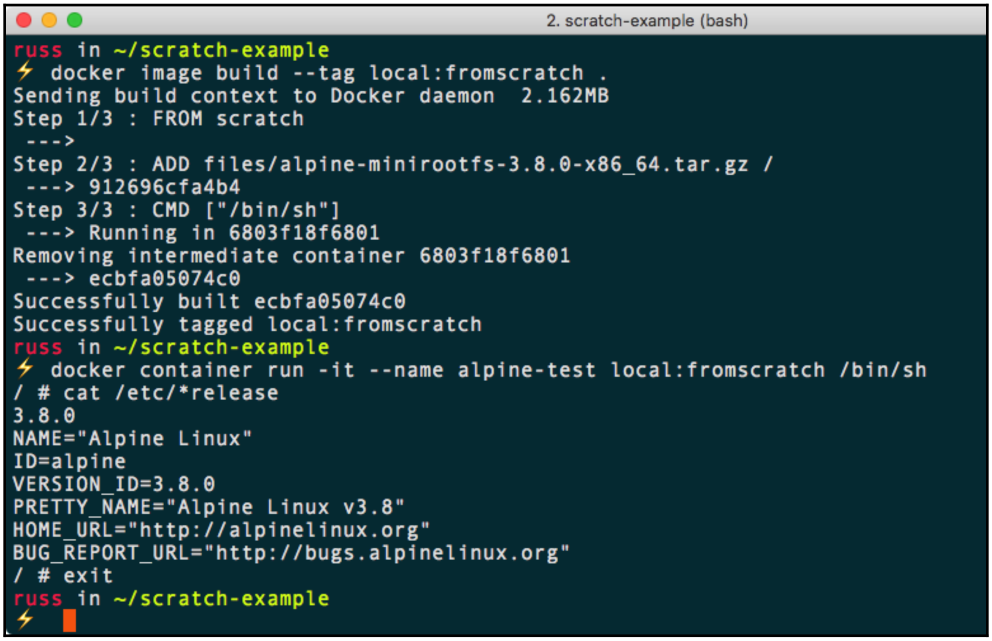
```
$ docker container run -it --name alpine-test local:fromscratch /bin/sh
```

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 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:



```
russ in ~/scratch-example
⚡ docker image build --tag local:fromscratch .
Sending build context to Docker daemon  2.162MB
Step 1/3 : FROM scratch
 --->
Step 2/3 : ADD files/alpine-minirootfs-3.8.0-x86_64.tar.gz /
 ---> 912696cfa4b4
Step 3/3 : CMD ["/bin/sh"]
 ---> Running in 6803f18f6801
Removing intermediate container 6803f18f6801
 ---> ecbfa05074c0
Successfully built ecbfa05074c0
Successfully tagged local:fromscratch
russ in ~/scratch-example
⚡ docker container run -it --name alpine-test local:fromscratch /bin/sh
/ # cat /etc/*release
3.8.0
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.8.0
PRETTY_NAME="Alpine Linux v3.8"
HOME_URL="http://alpinelinux.org"
BUG_REPORT_URL="http://bugs.alpinelinux.org"
/ # exit
russ in ~/scratch-example
⚡
```

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 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 detail on how to use any of these tools here because, if you have to consider this approach, you probably have some pretty specific requirements. There is a list of tools in the further reading section at the end of this chapter.

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, allowing you to host your old legacy application on a modern, supportable operating system/architecture.

Using environmental variables

In this section, we will cover the very powerful **environmental variables** (**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 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 use an equals sign between the two:

```
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. Let's look at building something a little more dynamic. Using Alpine Linux, we will do the following:

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

Our Dockerfile looks like the following:

```
FROM alpine:latest
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 image `local/apache-php` and tagging the version as `7`. The full output obtained by running the preceding command can be found here:

```
Sending build context to Docker daemon 2.56kB
Step 1/8 : FROM alpine:latest
 ---> 11cd0b38bc3c
```

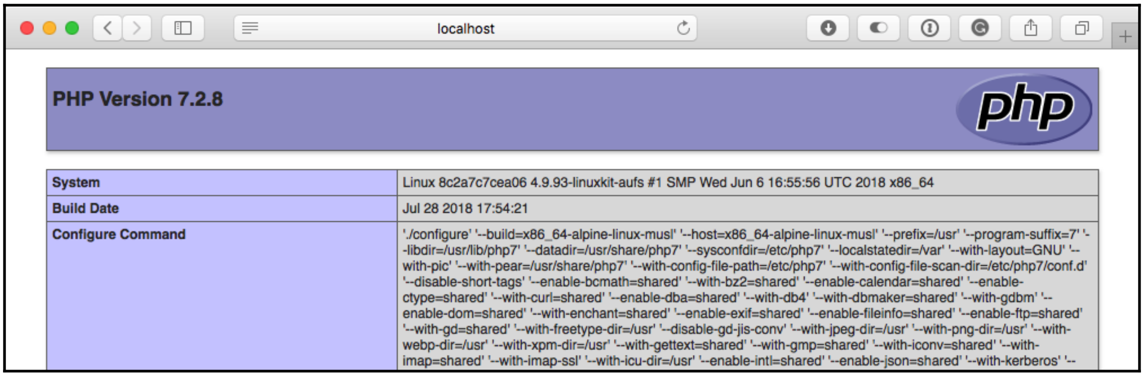
```
Step 2/8 : LABEL maintainer="Russ McKendrick <russ@mckendrick.io>"
 ---> Using cache
 ---> 175e9ebf182b
Step 3/8 : LABEL description="This example Dockerfile installs Apache &
PHP."
 ---> Running in 095e42841956
Removing intermediate container 095e42841956
 ---> d504837e80a4
Step 4/8 : ENV PHPVERSION=7
 ---> Running in 0df665a9b23e
Removing intermediate container 0df665a9b23e
 ---> 7f2c212a70fc
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 ea77c54e08bf
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 (2.2.5-r0)
(4/14) Installing apr-util (1.6.1-r2)
(5/14) Installing pcre (8.42-r0)
(6/14) Installing apache2 (2.4.33-r1)
Executing apache2-2.4.33-r1.pre-install
(7/14) Installing php7-common (7.2.8-r1)
(8/14) Installing ncurses-terminfo-base (6.1-r0)
(9/14) Installing ncurses-terminfo (6.1-r0)
(10/14) Installing ncurses-libs (6.1-r0)
(11/14) Installing libedit (20170329.3.1-r3)
(12/14) Installing libxml2 (2.9.8-r0)
(13/14) Installing php7 (7.2.8-r1)
(14/14) Installing php7-apache2 (7.2.8-r1)
Executing busybox-1.28.4-r0.trigger
OK: 26 MiB in 27 packages
Removing intermediate container ea77c54e08bf
 ---> 49b49581f8e2
Step 6/8 : EXPOSE 80/tcp
 ---> Running in e1cbc518ef07
Removing intermediate container e1cbc518ef07
 ---> a061e88eb39f
Step 7/8 : ENTRYPOINT ["httpd"]
 ---> Running in 93ac42d6ce55
Removing intermediate container 93ac42d6ce55
 ---> 9e09239021c2
```

```
Step 8/8 : CMD ["-D", "FOREGROUND"]
 ---> Running in 733229cc945a
Removing intermediate container 733229cc945a
 ---> 649b432e8d47
Successfully built 649b432e8d47
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/` and you should see a page showing that PHP7 is being used:



PHP Version 7.2.8

System	Linux 8c2a7c7cea06 4.9.93-linuxkit-aufs #1 SMP Wed Jun 6 16:55:56 UTC 2018 x86_64
Build Date	Jul 28 2018 17:54:21
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' '--

Don't be confused by the next part; there is no PHP6. For an explanation of why not, go to `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 .
```

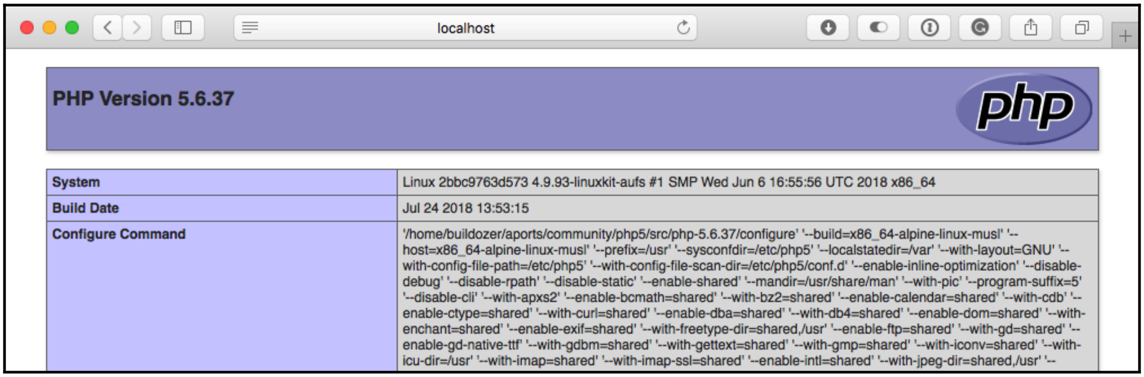
As you can see from the following Terminal output, the majority of the output is the same, apart from the packages that are being installed:

```
Sending build context to Docker daemon 2.56kB
Step 1/8 : FROM alpine:latest
 ---> 11cd0b38bc3c
Step 2/8 : LABEL maintainer="Russ McKendrick <russ@mckendrick.io>"
 ---> Using cache
 ---> 175e9ebf182b
Step 3/8 : LABEL description="This example Dockerfile installs Apache &
PHP."
 ---> Using cache
 ---> d504837e80a4
Step 4/8 : ENV PHPVERSION=5
 ---> Running in 0646b5e876f6
Removing intermediate container 0646b5e876f6
 ---> 3e17f6c10a50
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 d55a7726e9a7
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
(1/10) Installing libuuid (2.32-r0)
(2/10) Installing apr (1.6.3-r1)
(3/10) Installing expat (2.2.5-r0)
(4/10) Installing apr-util (1.6.1-r2)
(5/10) Installing pcre (8.42-r0)
(6/10) Installing apache2 (2.4.33-r1)
Executing apache2-2.4.33-r1.pre-install
(7/10) Installing php5 (5.6.37-r0)
(8/10) Installing php5-common (5.6.37-r0)
(9/10) Installing libxml2 (2.9.8-r0)
(10/10) Installing php5-apache2 (5.6.37-r0)
Executing busybox-1.28.4-r0.trigger
OK: 32 MiB in 23 packages
Removing intermediate container d55a7726e9a7
 ---> 634ab90b168f
Step 6/8 : EXPOSE 80/tcp
 ---> Running in a59f40d3d5df
Removing intermediate container a59f40d3d5df
 ---> d1aadf757f59
Step 7/8 : ENTRYPOINT ["httpd"]
 ---> Running in c7a1ab69356d
Removing intermediate container c7a1ab69356d
 ---> 22a9eb0e6719
Step 8/8 : CMD ["-D", "FOREGROUND"]
 ---> Running in 8ea92151ce22
Removing intermediate container 8ea92151ce22
 ---> da34eaff9541
Successfully built da34eaff9541
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:



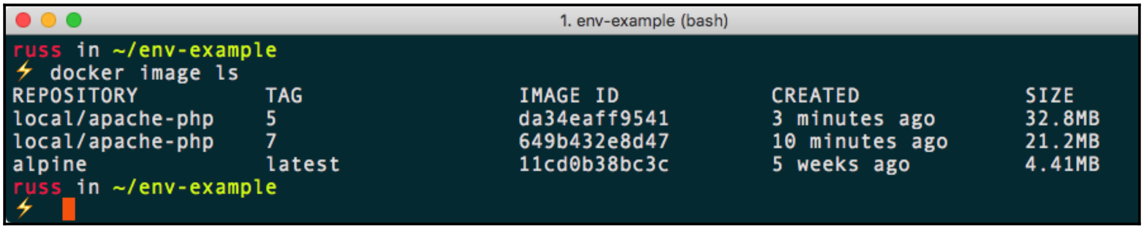
PHP Version 5.6.37

System	Linux 2bbc9763d573 4.9.93-linuxkit-aufs #1 SMP Wed Jun 6 16:55:56 UTC 2018 x86_64
Build Date	Jul 24 2018 13:53:15
Configure Command	'/home/buildozer/aports/community/php5/src/php-5.6.37/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' '--

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

```
$ docker image ls
```

You should see the following Terminal output:



```
russ in ~/env-example
⚡ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
local/apache-php    5                   da34eaff9541        3 minutes ago       32.8MB
local/apache-php    7                   649b432e8d47        10 minutes ago      21.2MB
alpine              latest              11cd0b38bc3c        5 weeks ago         4.41MB
russ in ~/env-example
⚡
```

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 of 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
```

Or, 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 environment variables 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=1.2.2
CONSUL_SHA256=7fa3b287b22b58283b8bd5479291161af2badbc945709eb5412840d91b912
060

RUN apk add --update ca-certificates wget && \
 wget -O consul.zip
https://releases.hashicorp.com/consul/${CONSUL_VERSION}/consul_${CONSUL_VER
SION}_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 use of ENVs can help with the 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 the building of a new image—well, it would do if we had configured it to do so; we will be looking at this in the next chapter.

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

Using multi-stage builds

In this, 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 of this part of the chapter, we looked at adding binaries directly to our images either via a package manager, such as Alpine Linux's APK, or, in the last 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. Downloading the build environment container image and starting a "build" container
2. Copying the source code to the "build" container
3. Compiling the source code on the "build" container
4. Copying the compiled binary outside of the "build" container
5. Removing the "build" container
6. Using 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 the 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/hel
lo_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"]
```

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 command can be found in the following code block—the interesting bits happen between steps 5 and 6:

```
Sending build context to Docker daemon 9.216kB
Step 1/8 : FROM golang:latest as builder
latest: Pulling from library/golang
55cbf04beb70: Pull complete
1607093a898c: Pull complete
9a8ea045c926: Pull complete
d4eee24d4dac: Pull complete
9c35c9787a2f: Pull complete
6a66653f6388: Pull complete
102f6b19f797: Pull complete
Digest:
sha256:957f390aceead48668eb103ef162452c6dae25042ba9c41762f5210c5ad3aeea
Status: Downloaded newer image for golang:latest
 ---> d0e7a411e3da
Step 2/8 : WORKDIR /go-http-hello-world/
 ---> Running in e1d56745f358
Removing intermediate container e1d56745f358
```

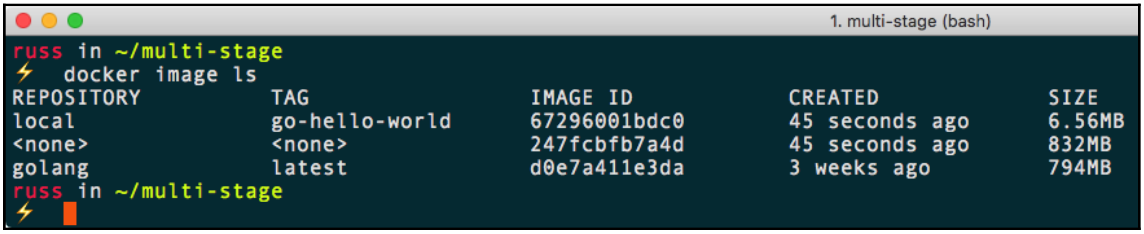
```
 ---> f18dfc0166a0
Step 3/8 : RUN go get -d -v golang.org/x/net/html
 ---> Running in 5e97d81db53c
Fetching https://golang.org/x/net/html?go-get=1
Parsing meta tags from https://golang.org/x/net/html?go-get=1 (status code
200)
get "golang.org/x/net/html": found meta tag
get.metaImport{Prefix:"golang.org/x/net", VCS:"git",
RepoRoot:"https://go.googlesource.com/net"} at
https://golang.org/x/net/html?go-get=1
get "golang.org/x/net/html": verifying non-authoritative meta tag
Fetching https://golang.org/x/net?go-get=1
Parsing meta tags from https://golang.org/x/net?go-get=1 (status code 200)
golang.org/x/net (download)
Removing intermediate container 5e97d81db53c
 ---> f94822756a52
Step 4/8 : ADD
https://raw.githubusercontent.com/geetarista/go-http-hello-world/master/hel
lo_world/hello_world.go ./hello_world.go
Downloading 393B
 ---> ecf3944740e1
Step 5/8 : RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o
app .
 ---> Running in 6e2d39c4d8ba
Removing intermediate container 6e2d39c4d8ba
 ---> 247fcbfb7a4d
Step 6/8 : FROM scratch
 --->
Step 7/8 : COPY --from=builder /go-http-hello-world/app .
 ---> a69cf59ab1d3
Step 8/8 : CMD ["./app"]
 ---> Running in c99076fad7fb
Removing intermediate container c99076fad7fb
 ---> 67296001bdc0
Successfully built 67296001bdc0
Successfully tagged local:go-hello-world
```

As you can see, between steps 5 and 6, our binary has been compiled and the container that contains the `builder` environment is removed, leaving us with an image storing our binary. Step 7 copies the binary to a fresh container which has been launched using scratch, leaving us with just the content we need.

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 screenshot of our output shows that the `golang` image is `794MB`; with our source code and prerequisites added, the size increases to `832MB`:



```
russ in ~/multi-stage
⚡  docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
local               go-hello-world      67296001bdc0        45 seconds ago      6.56MB
<none>              <none>              247fcbfb7a4d        45 seconds ago      832MB
golang              latest              d0e7a411e3da        3 weeks ago         794MB
russ in ~/multi-stage
⚡
```

However, the final image is just `6.56MB`. I am sure you will agree that this is quite a dramatic saving of space. It also adheres to the best practices, discussed earlier in the chapter, by only having content relevant to our application shipped within the image, as well as being really, really small.

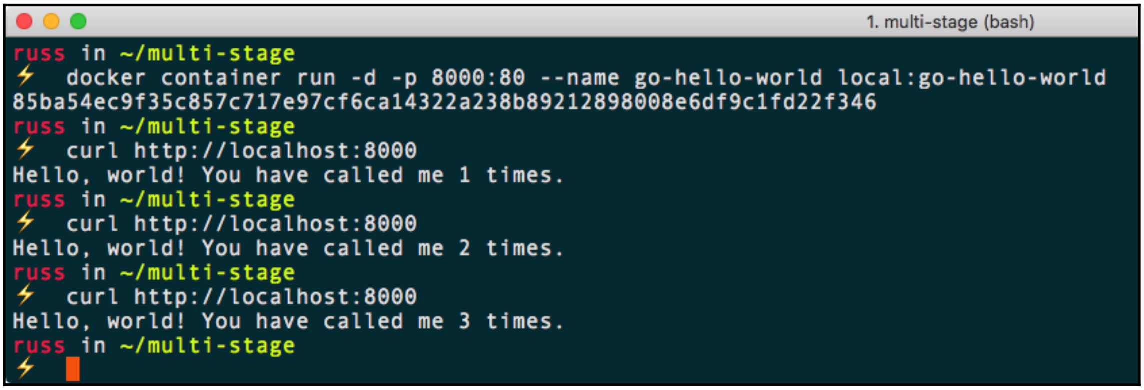
You can test the application by launching a container with the following command:

```
$ 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 in ~/multi-stage
⚡  docker container run -d -p 8000:80 --name go-hello-world local:go-hello-world
85ba54ec9f35c857c717e97cf6ca14322a238b89212898008e6df9c1fd22f346
russ in ~/multi-stage
⚡  curl http://localhost:8000
Hello, world! You have called me 1 times.
russ in ~/multi-stage
⚡  curl http://localhost:8000
Hello, world! You have called me 2 times.
russ in ~/multi-stage
⚡  curl http://localhost:8000
Hello, world! You have called me 3 times.
russ in ~/multi-stage
⚡
```

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.

Summary

In this chapter, we looked at an in-depth view of Dockerfiles, the best practices for writing them, the docker image build command, and the various ways we can build containers. We also learned about the environmental variables that you can use to pass from your Dockerfile to the various items inside your containers.

In the next chapter, now that we know how to build images using Dockerfiles, we will be taking a look at the 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 you can roll your own place to store images without the fees of Docker Enterprise, as well as third-party registry services.

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 the following:

- Debootstrap: `https://wiki.debian.org/Debootstrap/`
- Yumbootstrap: `https://github.com/dozzie/yumbootstrap/`
- Rinse: `https://salsa.debian.org/debian/rinse/`
- Docker contrib scripts: `https://github.com/moby/moby/tree/master/contrib/`

Finally, 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 allow you to store your images, and also Docker Registry, which you can use to run your local storage for Docker containers. We will review the differences between the services and when and how to use each of them.

This chapter will also cover how to set up automated builds using Webhooks, as well as all the pieces that are required to set them up. Let's take a quick look at the topics we will be covering in this chapter:

- Docker Hub
- Docker Store
- Docker Registry
- Third-party registries
- Microbadger

Technical requirements

In this chapter, we will be using our Docker installation to build images. As before, although the screenshots in this chapter will be from my preferred operating system, macOS, the commands we will be running will work on all three of the operating systems covered in the previous chapter. A full copy of the code used in this chapter can be found at: `https://github.com/PacktPublishing/Mastering-Docker-Third-Edition/tree/master/chapter03`.

Check out the following video to see the Code in Action:

`http://bit.ly/2EBVJjJ`

Docker Hub

While 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.

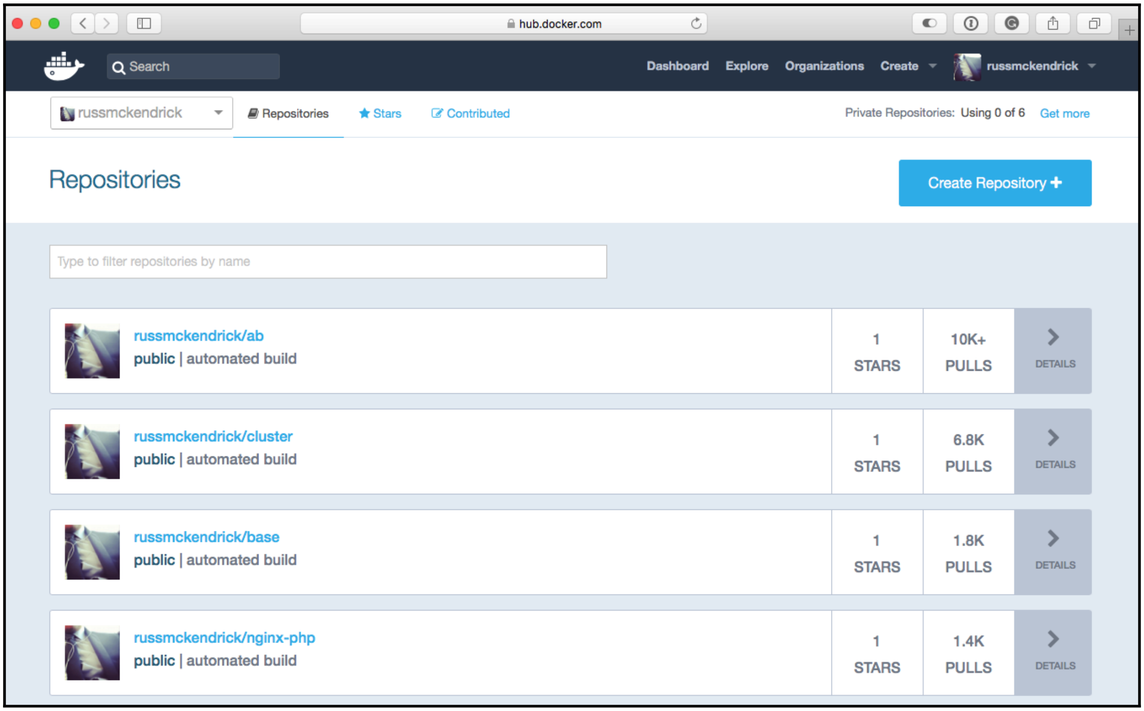
In this section, we will focus on Docker Hub, which has both a freely available option, where you can only host publicly accessible images, and also a subscription option, which allows you to host your own private images. We will focus on the web aspect of Docker Hub and the management you can do there.

The home page, which can be found at `https://hub.docker.com/`, contains a **Sign-Up** form and, in 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**.

Docker Hub is free to use, and if you do not need to upload or manage your own images, you do not need an account to search for pull images.

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:



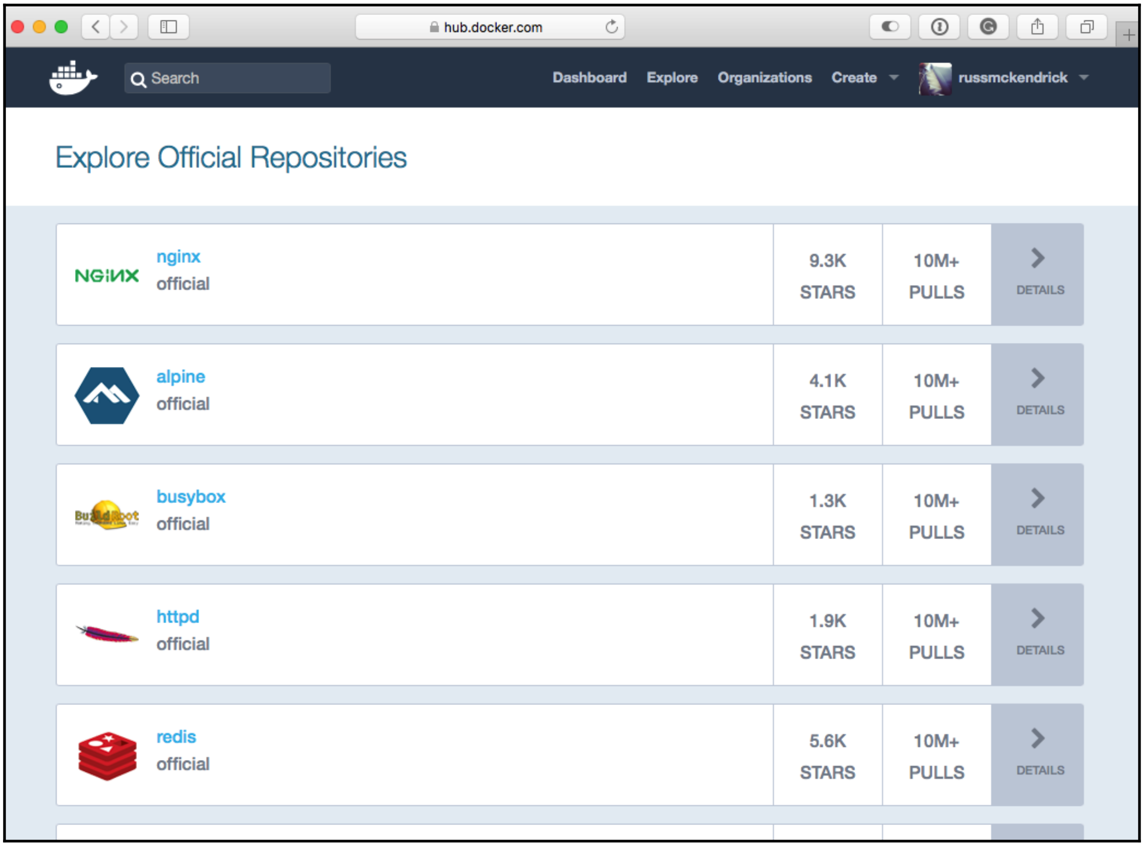
Repository	Stars	Pulls
russmckendrick/ab public | automated build	1 STARS	10K+ PULLS
russmckendrick/cluster public | automated build	1 STARS	6.8K PULLS
russmckendrick/base public | automated build	1 STARS	1.8K PULLS
russmckendrick/nginx-php public | automated build	1 STARS	1.4K PULLS

From here, you can get to all the other sub-pages of Docker Hub. However, before we look at those sections, we should talk a little about the dashboard. From here, you can view all of 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 see on the dashboard, starting with the dark blue menu at the top of the page.

Explore

The **Explore** option takes you to a list of the official Docker images; like your **Dashboard**, they are ordered by stars and then pulls. As you can see from the following screen, each of the official images has had over 10 million pulls:



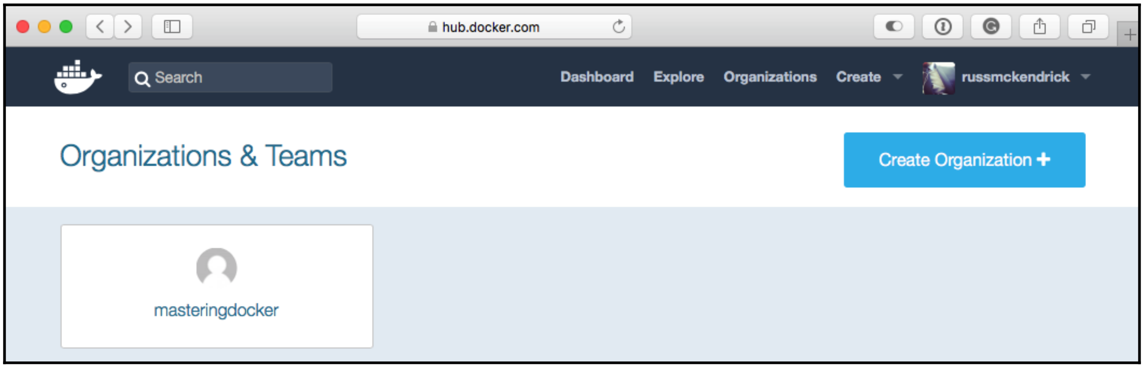
Explore Official Repositories

nginx official	9.3K STARS	10M+ PULLS	DETAILS
alpine official	4.1K STARS	10M+ PULLS	DETAILS
busybox official	1.3K STARS	10M+ PULLS	DETAILS
httpd official	1.9K STARS	10M+ PULLS	DETAILS
redis official	5.6K STARS	10M+ PULLS	DETAILS

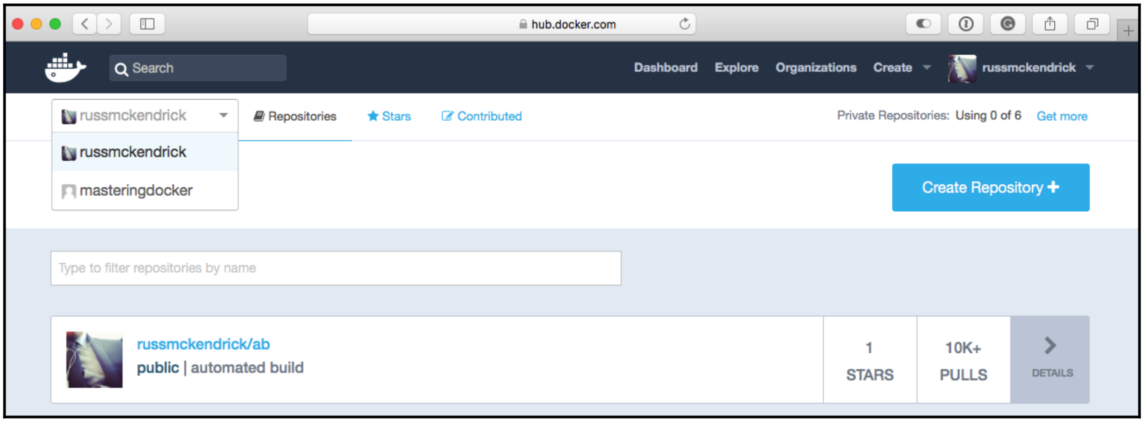
This isn't the preferred Docker Store method of downloading official images. Docker would prefer you used the Docker Store now, but as we will be looking at this in more detail later in the chapter, we won't go into any more detail here.

Organizations

Organizations are those which you have either created or have 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.



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:

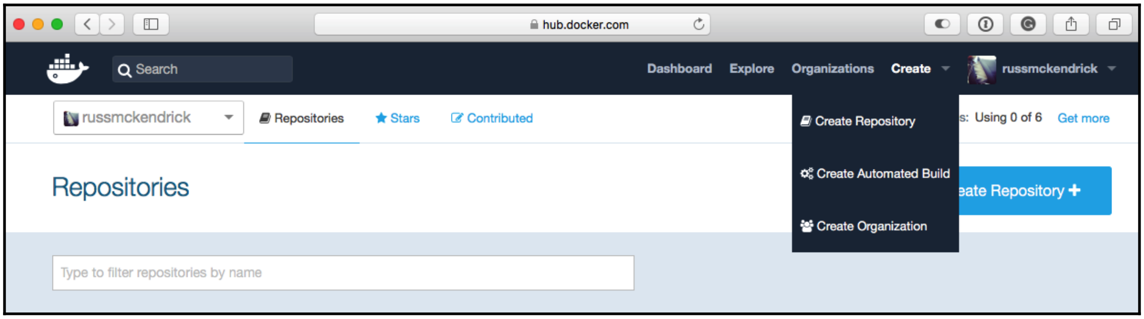


Create

We will go into more detail about creating a repository and an automated build in a later section, so I will not go into any detail here, other than to say that the **Create** menu gives you three options:

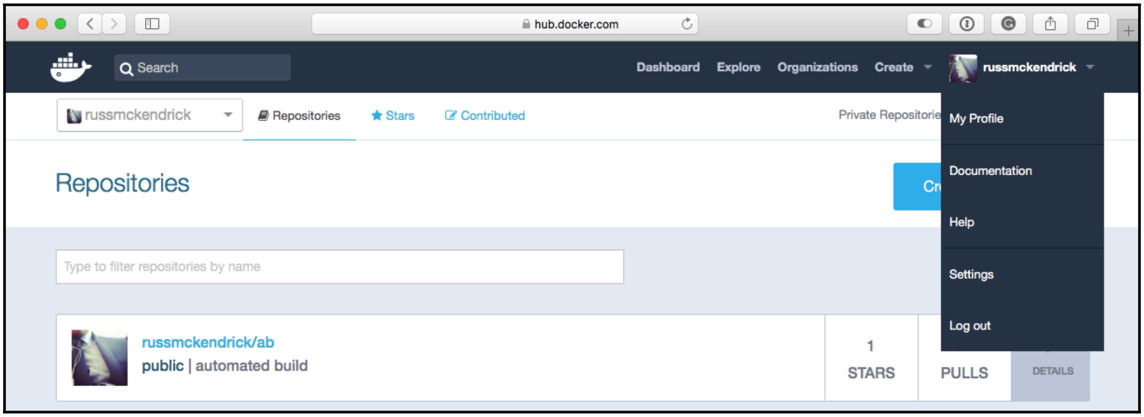
- **Create Repository**
- **Create Automated Build**
- **Create Organization**

These options can be seen in the following screenshot:



Profile and settings

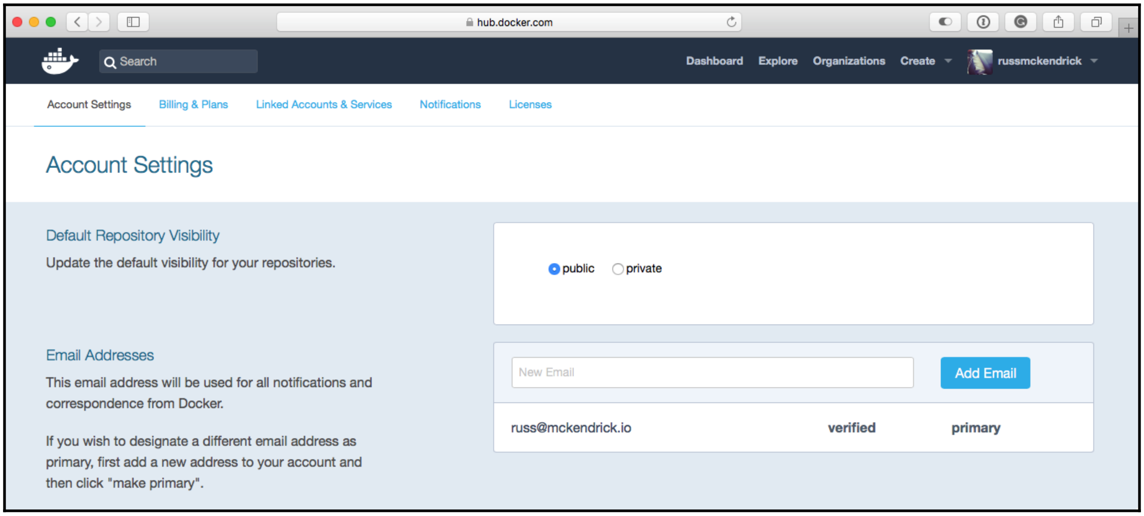
The final option in the top menu is about managing **My Profile** and **Settings**:



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

- Changing your password
- Seeing what organization you belong to
- Seeing what subscriptions for email updates you have
- Setting specific notifications you would like to receive
- Setting which authorized services have access to your information
- Seeing linked accounts (such as your GitHub or Bitbucket accounts)
- Viewing your enterprise licenses, billing, and global settings

The only global setting as of now is the choice between having your repositories default to **public** or **private** upon creation. The default is to create them as **public** repositories:



Account Settings

Default Repository Visibility

Update the default visibility for your repositories.

● public ○ private

Email Addresses

This email address will be used for all notifications and correspondence from Docker.

If you wish to designate a different email address as primary, first add a new address to your account and then click "make primary".

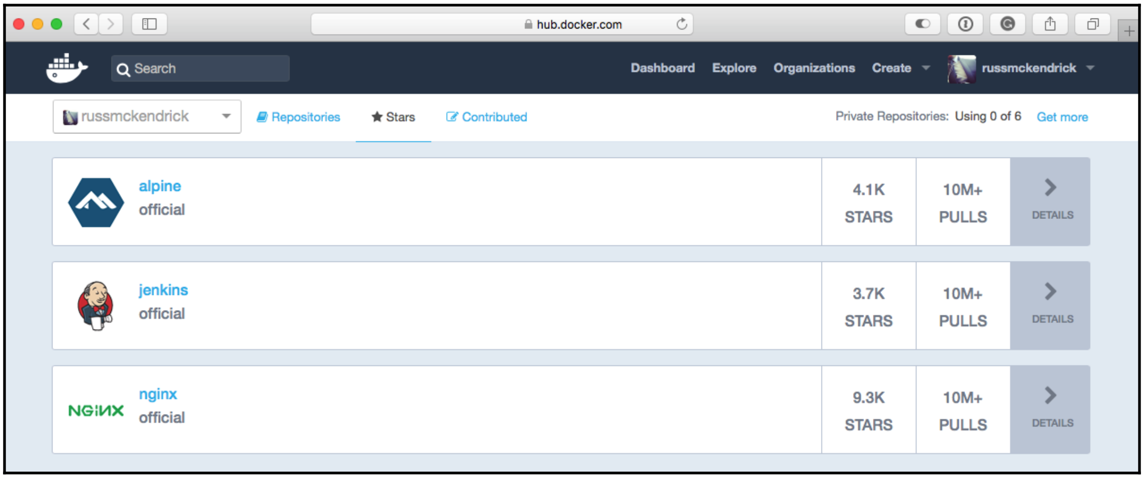
New Email

russ@mckendrick.io verified primary

The **My Profile** menu item takes you to your public profile page; mine can be found at `https://hub.docker.com/u/russmckendrick/`.

Other menu options

Below the dark blue bar at the top of the **Dashboard** page are two more areas that we haven't yet covered. The first, the **Stars** page, allows you to see which repositories you yourself have starred:



alpine official	4.1K STARS	10M+ PULLS	DETAILS
jenkins official	3.7K STARS	10M+ PULLS	DETAILS
nginx official	9.3K STARS	10M+ PULLS	DETAILS

This is very useful if you come across some repositories that you prefer to use, and want to access them to see whether they have been updated recently, or whether any other changes have occurred on these repositories.

The second is a new setting, **Contributed**. Clicking this will reveal a section in which there will be a list of repositories you have made contributions to outside of the ones within your own **Repositories** list.

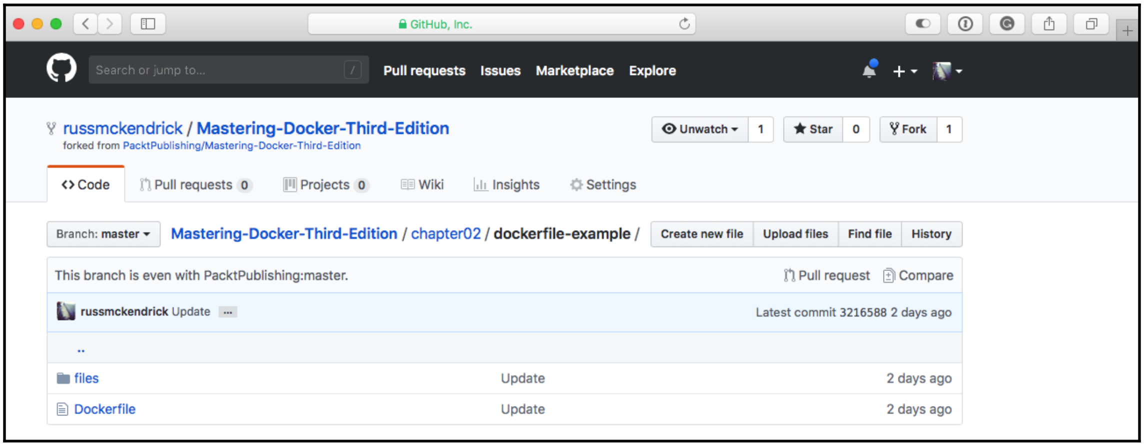
Creating an automated build

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, and by the end, you'll be able to automate all your 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 our example, I will be using GitHub, but the setup will be the same for GitHub and Bitbucket.

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



russmckendrick / Mastering-Docker-Third-Edition
forked from PacktPublishing/Mastering-Docker-Third-Edition

Mastering-Docker-Third-Edition / chapter02 / dockerfile-example /

This branch is even with PacktPublishing:master.

russmckendrick Update ... Latest commit 3216588 2 days ago

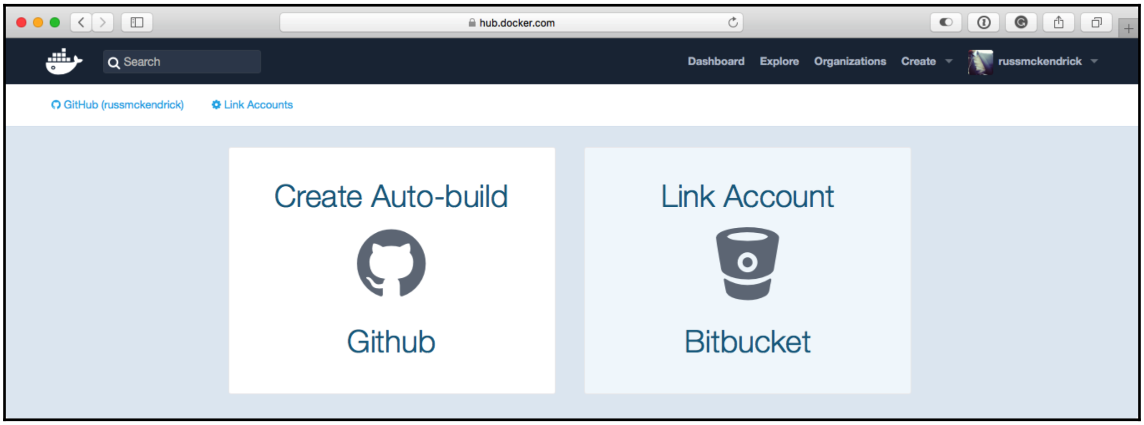
..

files	Update	2 days ago
Dockerfile	Update	2 days ago

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

Setting up Docker Hub

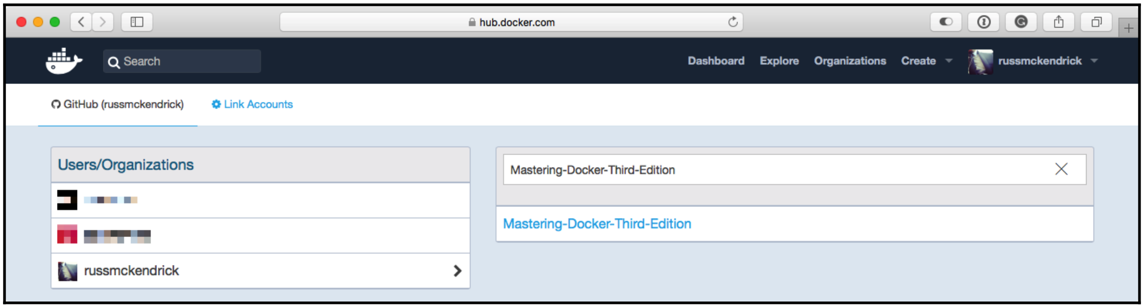
In Docker Hub, we are going to use the **Create** drop-down menu and select **Create Automated Build**. After selecting it, we will be taken to a screen that will show you the accounts you have linked to either GitHub or Bitbucket:



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, and all that I had to do was to 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 and Private**: This is the recommended option. Docker Hub will have access to all of your public and private repositories, as well as organizations. Docker Hub will also be able to configure the Webhooks needed when setting up automated builds.
- **Limited Access**: This limits Docker Hubs access to publicly available repositories and organizations. If you link your accounts using this option Docker Hub won't be able to configure the Webhooks 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 Webhook that instructs that when a commit is done on a selected code repository, a new build will be created on Docker Hub.



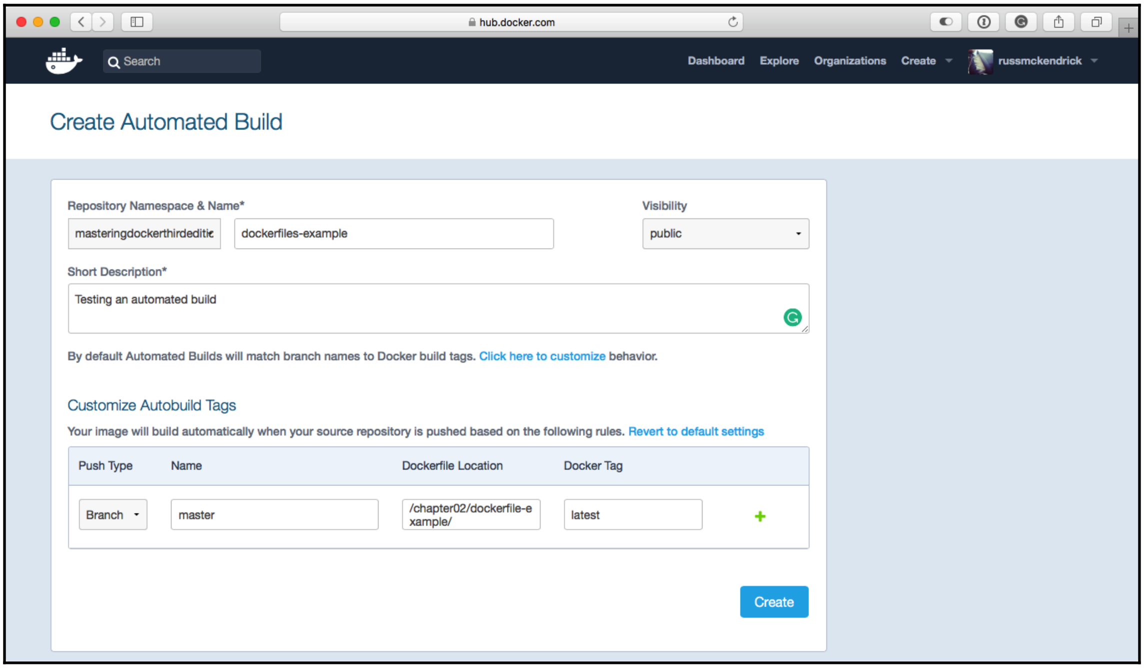
Users/Organizations

- [redacted]
- [redacted]
- russmckendrick

Mastering-Docker-Third-Edition

Mastering-Docker-Third-Edition

In the preceding screenshot, I selected `Mastering-Docker-Third-Edition` 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 customize**. We can let Docker Hub know the location of our Dockerfile as follows:



Create Automated Build

Repository Namespace & Name*: masteringdockerthirdeditic / dockerfiles-example

Visibility: public

Short Description*: Testing an automated build

By default Automated Builds will match branch names to Docker build tags. Click here to customize behavior.

Customize Autobuild Tags

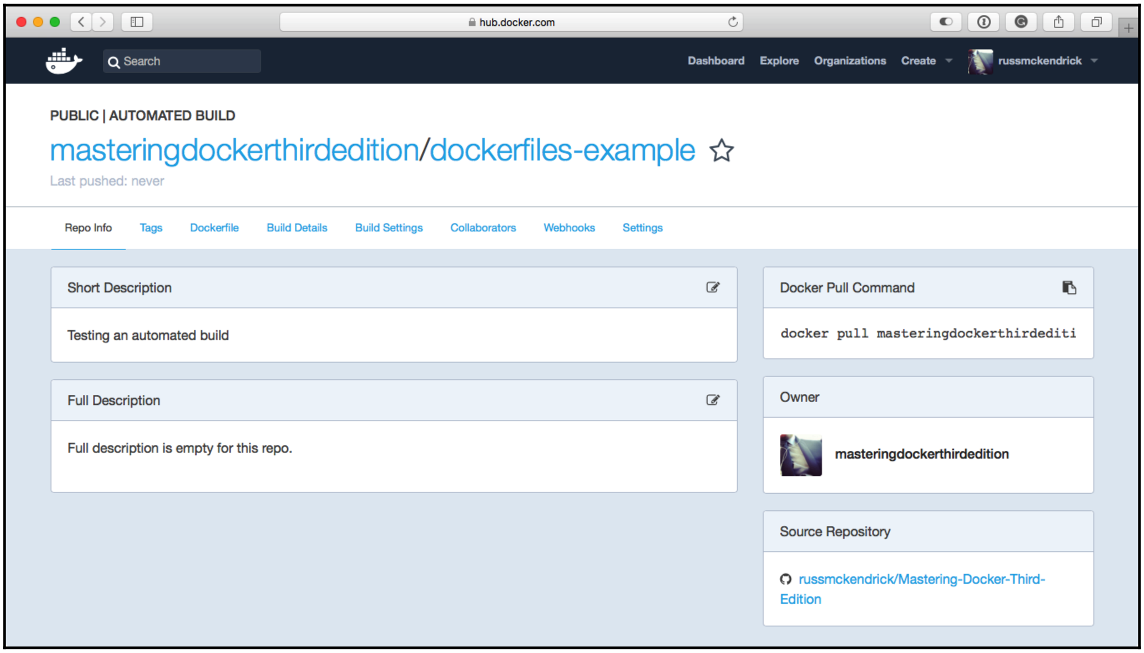
Your image will build automatically when your source repository is pushed based on the following rules. Revert to default settings

Push Type	Name	Dockerfile Location	Docker Tag
Branch	master	/chapter02/dockerfile-example/	latest

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

- **Repository Namespace & Name:** `dockerfile-example`
- **Visibility:** public
- **Short Description:** `Testing an automated build`
- **Push Type: Branch**
- **Name:** `master`
- **Dockerfile Location:** `/chapter02/dockerfile-example/`
- **Docker Tag: latest**

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



PUBLIC | AUTOMATED BUILD

masteringdockerthirdedition/dockerfiles-example

Last pushed: never

Short Description

Testing an automated build

Full Description

Full description is empty for this repo.

Docker Pull Command

`docker pull masteringdockerthirdediti`

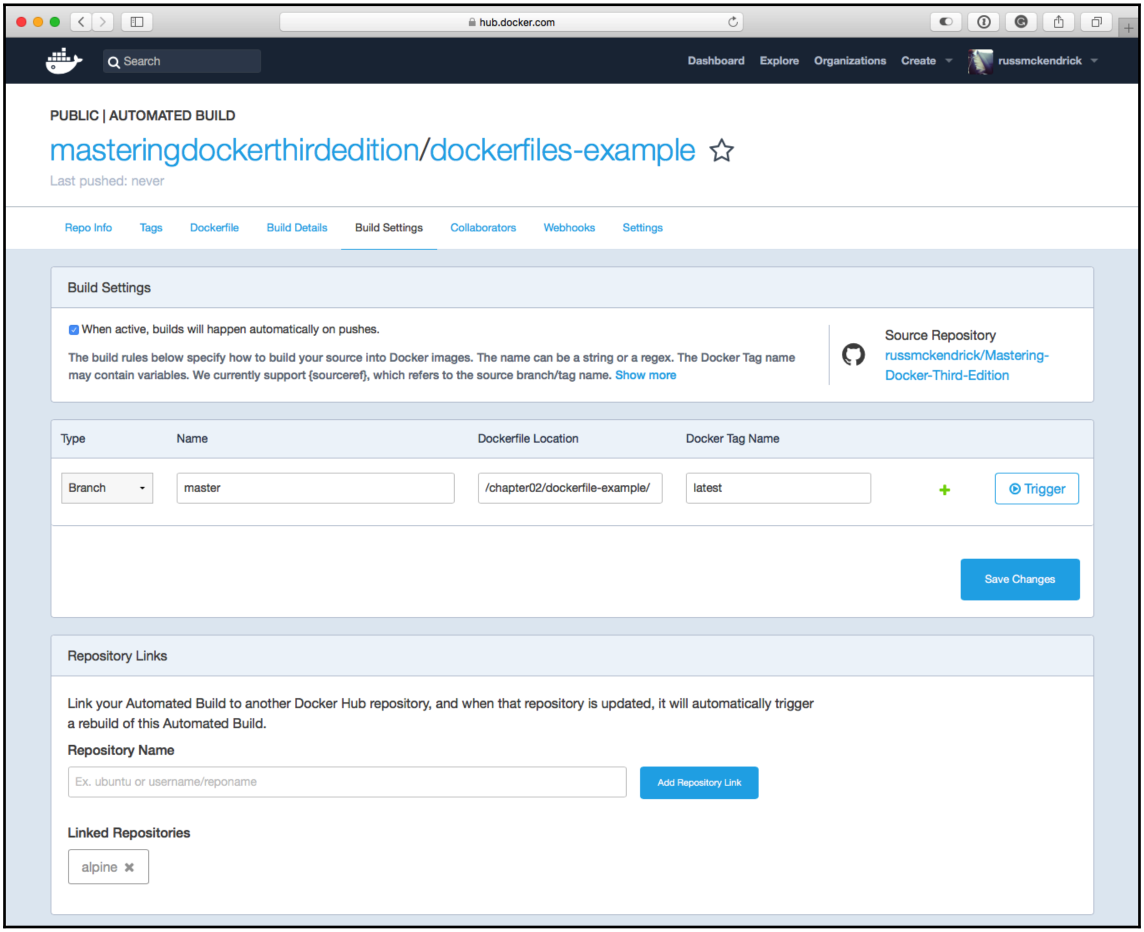
Owner

masteringdockerthirdedition

Source Repository

russmckendrick/Mastering-Docker-Third-Edition

Now that we have our build defined, we can add some additional configurations by clicking on **Build Settings**. As we are using the official Alpine Linux image, we can link that to our own build. To do that, enter Alpine in the **Repository Links** section and then click on **Add Repository Link**. This will kick off an unattended build each time a new version of the official Alpine Linux image is published.



PUBLIC | AUTOMATED BUILD

masteringdockerthirdedition/dockerfiles-example

Last pushed: never

Build Settings

☑ When active, builds will happen automatically on pushes.

The build rules below specify how to build your source into Docker images. The name can be a string or a regex. The Docker Tag name may contain variables. We currently support {sourceref}, which refers to the source branch/tag name. Show more

Source Repository russmckendrick/Mastering-Docker-Third-Edition

Type	Name	Dockerfile Location	Docker Tag Name
Branch	master	/chapter02/dockerfile-example/	latest

Repository Links

Link your Automated Build to another Docker Hub repository, and when that repository is updated, it will automatically trigger a rebuild of this Automated Build.

Repository Name

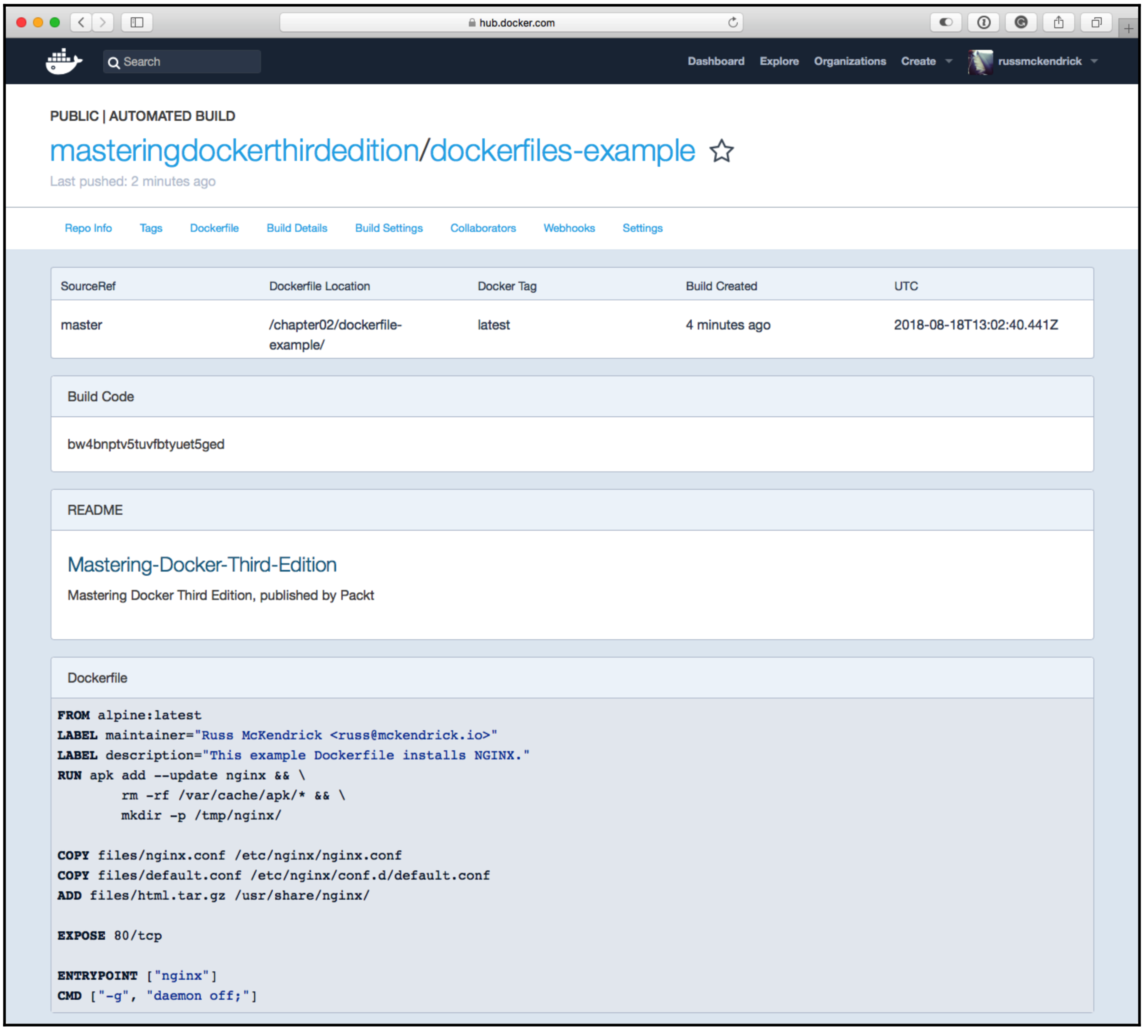
Ex. ubuntu or username/reponame

Linked Repositories

alpine ✖

So now our image will automatically be 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 to manually kick off a build. You will notice that the button turns green for a short time, which confirms that a build has been scheduled in the background.

Once have triggered your build, clicking on **Build Details** will bring up a list of all of the builds for the image, both successful and failed ones. You should see a build underway; clicking on it will bring up the logs for the build:



SourceRef	Dockerfile Location	Docker Tag	Build Created	UTC
master	/chapter02/dockerfile-example/	latest	4 minutes ago	2018-08-18T13:02:40.441Z

Build Code

bw4bnptv5tuvfbtyuet5ged

README

Mastering-Docker-Third-Edition

Mastering Docker Third Edition, published by Packt

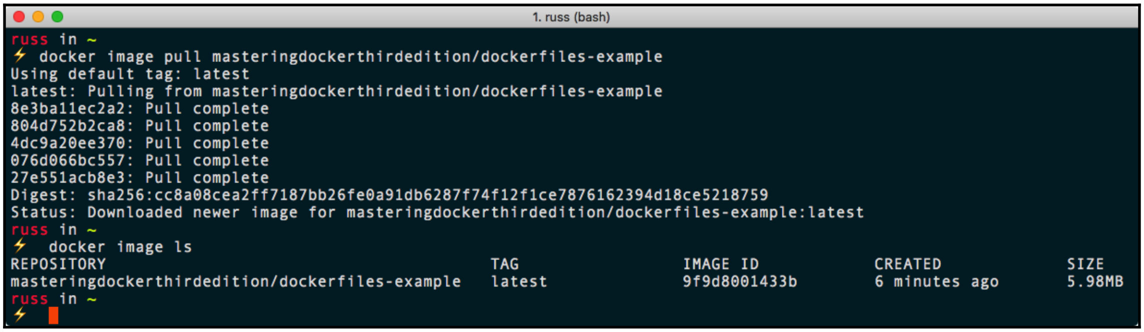
Dockerfile

```
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;"]
```

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 masteringdockerthirdedition/dockerfiles-example
$ docker image ls
```

The commands are shown in the following screenshot:



```
russ in ~
⚡ docker image pull masteringdockerthirdedition/dockerfiles-example
Using default tag: latest
latest: Pulling from masteringdockerthirdedition/dockerfiles-example
8e3ba11ec2a2: Pull complete
804d752b2ca8: Pull complete
4dc9a20ee370: Pull complete
076d066bc557: Pull complete
27e551acb8e3: Pull complete
Digest: sha256:cc8a08cea2ff7187bb26fe0a91db6287f74f12f1ce7876162394d18ce5218759
Status: Downloaded newer image for masteringdockerthirdedition/dockerfiles-example:latest
russ in ~
⚡  docker image ls
REPOSITORY                                        TAG                 IMAGE ID            CREATED             SIZE
masteringdockerthirdedition/dockerfiles-example   latest              9f9d8001433b        6 minutes ago       5.98MB
russ in ~
⚡
```

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 you can see from the following logs, which start off with a little bit of information about Docker's build environment:

```
Building in Docker Cloud's infrastructure...
Cloning into '.'...

KernelVersion: 4.4.0-1060-aws
Components: [{u'Version': u'18.03.1-ee-1-tp5', u'Name': u'Engine',
u'Details': {u'KernelVersion': u'4.4.0-1060-aws', u'Os': u'linux',
u'BuildTime': u'2018-06-23T07:58:56.000000000+00:00', u'ApiVersion':
u'1.37', u'MinAPIVersion': u'1.12', u'GitCommit': u'1b30665', u'Arch':
u'amd64', u'Experimental': u'false', u'GoVersion': u'go1.10.2'}}]
Arch: amd64
BuildTime: 2018-06-23T07:58:56.000000000+00:00
ApiVersion: 1.37
Platform: {u'Name': u''}
Version: 18.03.1-ee-1-tp5
MinAPIVersion: 1.12
GitCommit: 1b30665
Os: linux
GoVersion: go1.10.2
```

The build then starts by compiling our code as follows:

```
Starting build of index.docker.io/masteringdockerthirdedition/multi-
stage:latest...
Step 1/8 : FROM golang:latest as builder
 ---> d0e7a411e3da
Step 2/8 : WORKDIR /go-http-hello-world/
Removing intermediate container ea4bd2a1e92a
 ---> 0735d98776ef
Step 3/8 : RUN go get -d -v golang.org/x/net/html
 ---> Running in 5b180ef58abf
Fetching https://golang.org/x/net/html?go-get=1
Parsing meta tags from https://golang.org/x/net/html?go-get=1 (status code
200)
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
https://golang.org/x/net/html?go-get=1
get "golang.org/x/net/html": verifying non-authoritative meta tag
Fetching https://golang.org/x/net?go-get=1
Parsing meta tags from https://golang.org/x/net?go-get=1 (status code 200)
golang.org/x/net (download)
Removing intermediate container 5b180ef58abf
 ---> e2d566167ecd
Step 4/8 : ADD
https://raw.githubusercontent.com/geetarista/go-http-hello-world/master/hel
lo_world/hello_world.go ./hello_world.go
 ---> c5489fee49e0
Step 5/8 : RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o
app .
 ---> Running in 0c5892f9db02
Removing intermediate container 0c5892f9db02
 ---> 94087063b79a
```

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

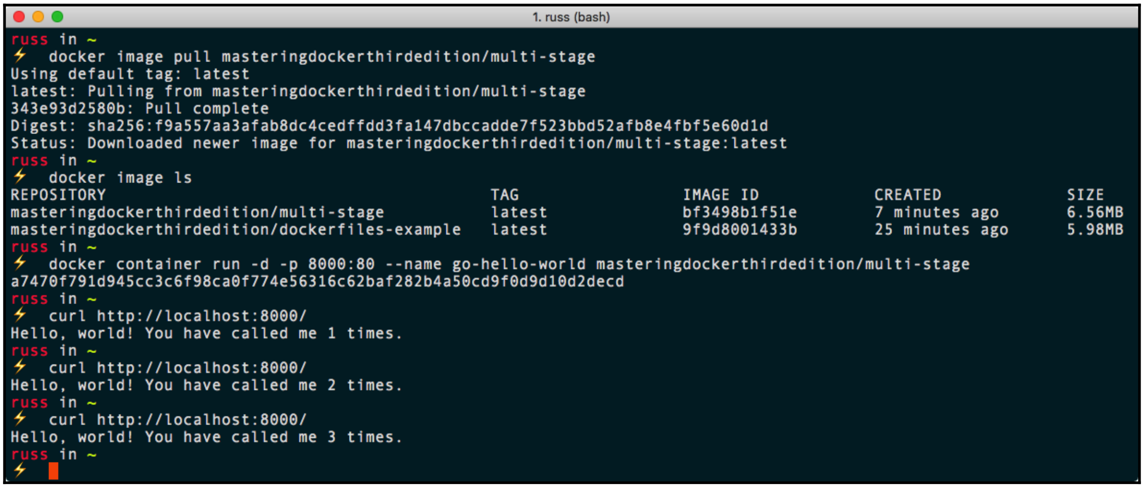
```
Step 6/8 : FROM scratch
 --->
Step 7/8 : COPY --from=builder /go-http-hello-world/app .
 ---> e16f25bc4201
Step 8/8 : CMD ["./app"]
 ---> Running in c93cfe262c15
Removing intermediate container c93cfe262c15
 ---> bf3498b1f51e

Successfully built bf3498b1f51e
Successfully tagged masteringdockerthirdedition/multi-stage:latest
```

```
Pushing index.docker.io/masteringdockerthirdedition/multi-stage:latest...
Done!
Build finished
```

You can pull and launch a container using the image with the following commands:

```
$ docker image pull masteringdockerthirdedition/multi-stage
$ docker image ls
$ docker container run -d -p 8000:80 --name go-hello-world
masteringdockerthirdedition/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:



```
russ in ~
⚡  docker image pull masteringdockerthirdedition/multi-stage
Using default tag: latest
latest: Pulling from masteringdockerthirdedition/multi-stage
343e93d2580b: Pull complete
Digest: sha256:f9a557aa3afab8dc4cedffdd3fa147dbccadde7f523bbd52afb8e4fbf5e60d1d
Status: Downloaded newer image for masteringdockerthirdedition/multi-stage:latest
russ in ~
⚡  docker image ls
REPOSITORY                                   TAG                 IMAGE ID            CREATED             SIZE
masteringdockerthirdedition/multi-stage      latest              bf3498b1f51e        7 minutes ago       6.56MB
masteringdockerthirdedition/dockerfiles-example   latest         9f9d8001433b        25 minutes ago      5.98MB
russ in ~
⚡  docker container run -d -p 8000:80 --name go-hello-world masteringdockerthirdedition/multi-stage
a7470f791d945cc3c6f98ca0f774e56316c62baf282b4a50cd9f0d9d10d2decd
russ in ~
⚡  curl http://localhost:8000/
Hello, world! You have called me 1 times.
russ in ~
⚡  curl http://localhost:8000/
Hello, world! You have called me 2 times.
russ in ~
⚡  curl http://localhost:8000/
Hello, world! You have called me 3 times.
russ in ~
⚡
```

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

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

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

Pushing your own image

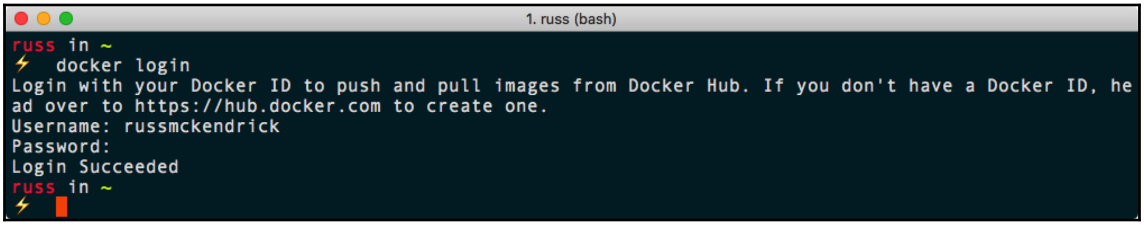
In `Chapter 2`, *Building Container Images*, we discussed creating an image without using a Dockerfile. While it is still not a good idea and should only be used when you really need to, you can push your own images to Docker Hub.

> 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:

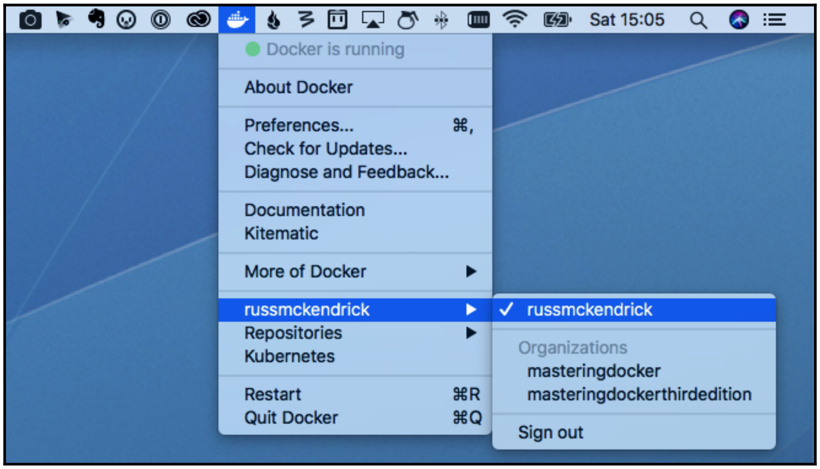
```
$ docker login
```

You will then be prompted for your Docker ID and password:



```
russ in ~
⚡  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
russ in ~
⚡
```

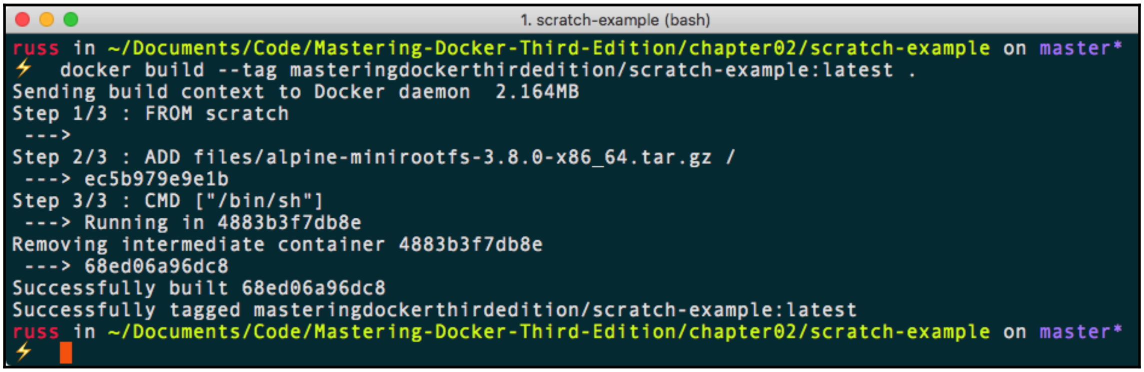
Also, if you are using Docker for Mac or Docker for Windows, you will now be logged in via the app and should be able to access Docker Hub from the menu:



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 masteringdockerthirdedition/scratch-example:latest .
```

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

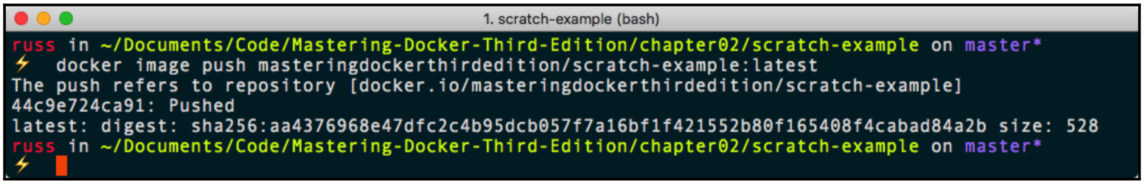


```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter02/scratch-example on master*
⚡  docker build --tag masteringdockerthirdedition/scratch-example:latest .
Sending build context to Docker daemon  2.164MB
Step 1/3 : FROM scratch
 --->
Step 2/3 : ADD files/alpine-minirootfs-3.8.0-x86_64.tar.gz /
 ---> ec5b979e9e1b
Step 3/3 : CMD ["/bin/sh"]
 ---> Running in 4883b3f7db8e
Removing intermediate container 4883b3f7db8e
 ---> 68ed06a96dc8
Successfully built 68ed06a96dc8
Successfully tagged masteringdockerthirdedition/scratch-example:latest
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter02/scratch-example on master*
⚡
```

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

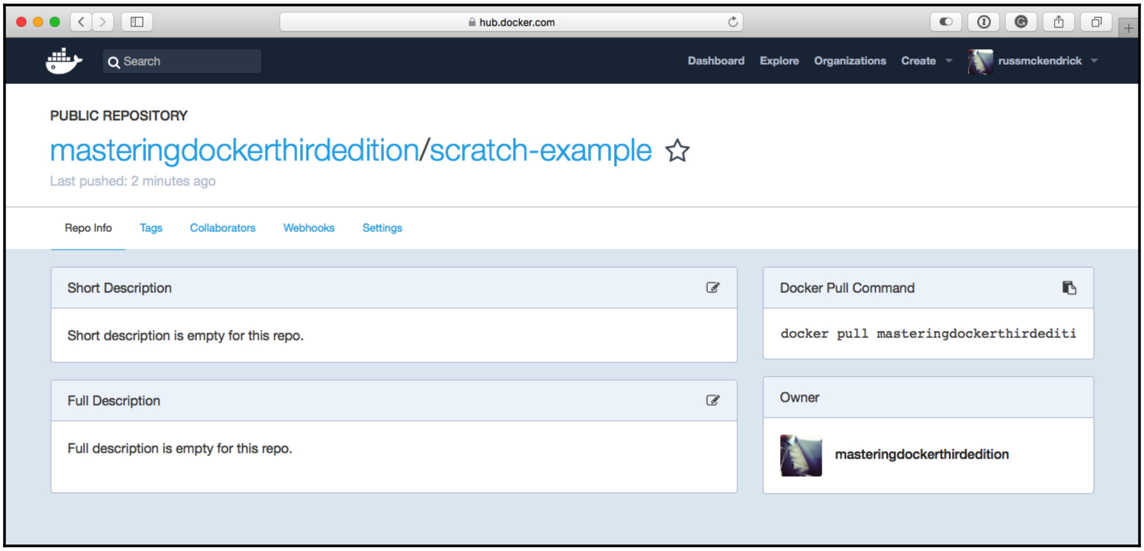
```
$ docker image push masteringdockerthirdedition/scratch-example:latest
```

The following screenshot shows the output:



```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter02/scratch-example on master*
⚡  docker image push masteringdockerthirdedition/scratch-example:latest
The push refers to repository [docker.io/masteringdockerthirdedition/scratch-example]
44c9e724ca91: Pushed
latest: digest: sha256:aa4376968e47dfc2c4b95dcb057f7a16bf1f421552b80f165408f4cabad84a2b size: 528
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter02/scratch-example on master*
⚡
```

As you can see, because we defined `masteringdockerthirdedition/scratch-example:latest` when we built the image, Docker automatically uploaded the image to that location, which in turn added a new image to the `Mastering Docker Third Edition` organization:



PUBLIC REPOSITORY

masteringdockerthirdedition/scratch-example

Last pushed: 2 minutes ago

Short Description

Short description is empty for this repo.

Docker Pull Command

```
docker pull masteringdockerthirdediti
```

Full Description

Full description is empty for this repo.

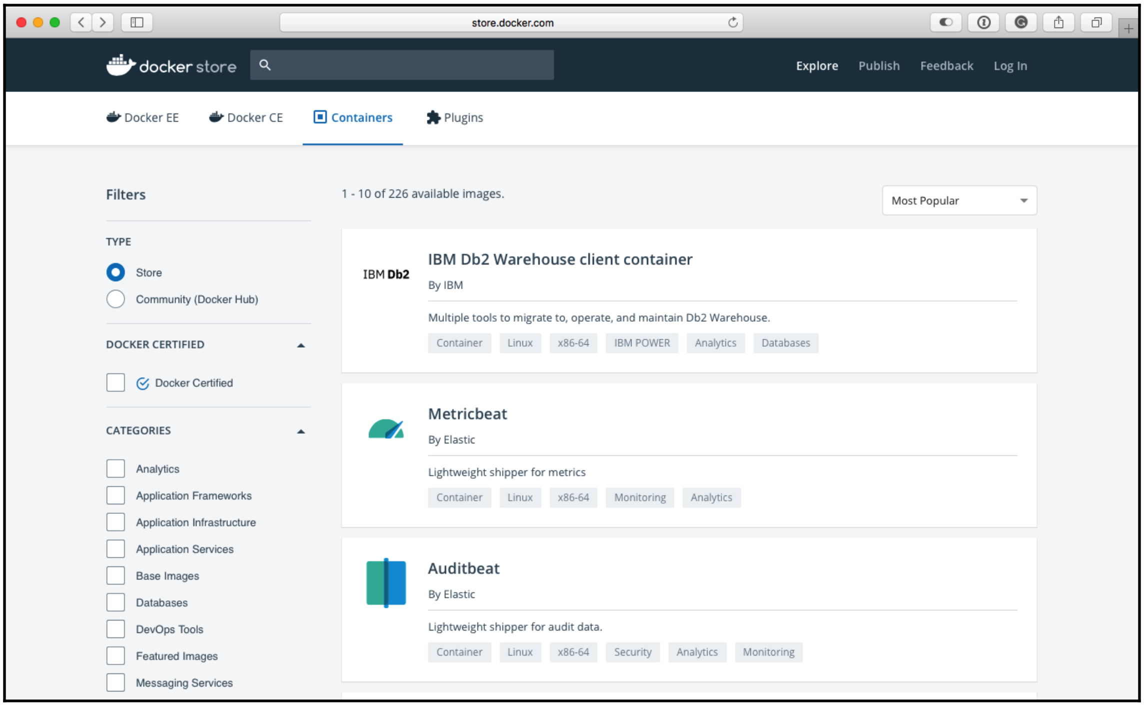
Owner

masteringdockerthirdedition

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.

Docker Store

You may remember that in `Chapter 1`, *Docker Overview*, we downloaded Docker for macOS and Docker for Windows from the Docker Store. 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 both **Docker Images** and **Docker Plugins**.



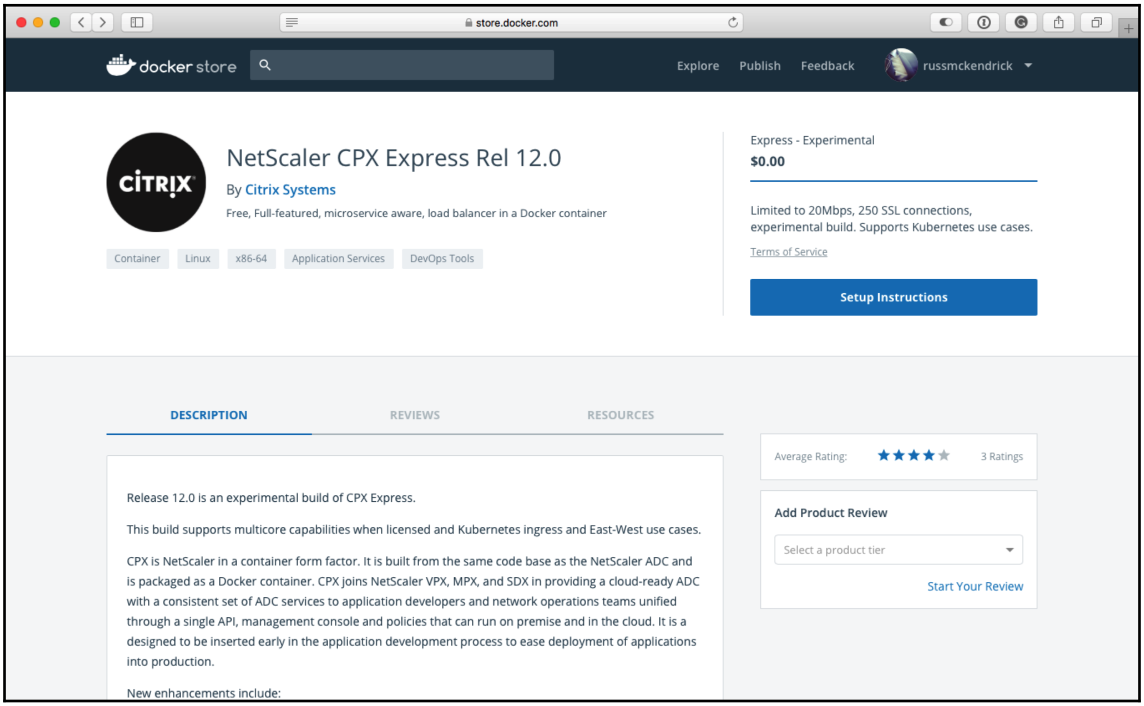
1 - 10 of 226 available images.

IBM Db2 Warehouse client container
By IBM
Multiple tools to migrate to, operate, and maintain Db2 Warehouse.
Container | Linux | x86-64 | IBM POWER | Analytics | Databases

Metricbeat
By Elastic
Lightweight shipper for metrics
Container | Linux | x86-64 | Monitoring | Analytics

Auditbeat
By Elastic
Lightweight shipper for audit data.
Container | Linux | x86-64 | Security | Analytics | Monitoring

While you will only find official and certified images in the Docker Store, there is an option to use the Docker Store interface to search through Docker Hub. Also, you can download images that are not available from Docker Hub, such as the Citrix NetScaler CPX Express image:



NetScaler CPX Express Rel 12.0

By Citrix Systems

Free, Full-featured, microservice aware, load balancer in a Docker container

Express - Experimental

$0.00

Limited to 20Mbps, 250 SSL connections, experimental build. Supports Kubernetes use cases.

Terms of Service

Add Product Review

Release 12.0 is an experimental build of CPX Express.

This build supports multicore capabilities when licensed and Kubernetes ingress and East-West use cases.

CPX is NetScaler in a container form factor. It is built from the same code base as the NetScaler ADC and is packaged as a Docker container. CPX joins NetScaler VPX, MPX, and SDX in providing a cloud-ready ADC with a consistent set of ADC services to application developers and network operations teams unified through a single API, management console and policies that can run on premise and in the cloud. It is a designed to be inserted early in the application development process to ease deployment of applications into production.

New enhancements include:

If you notice, the image has a price attached to it (the Express version is $0.00), meaning that you can buy commercial software through the Docker Store, as it has payments and licensing built in. If you are a software publisher, you are able to sign and distribute your own software through the Docker Store.

We will be looking at the Docker Store in a little more detail in later chapters, when we cover Docker plugins.

Docker Registry

In this section, we will be looking 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 look at the comparison between Docker Registry and Docker Hub, and how to choose between the two. By the end of the section, you will learn how to run your own Docker Registry and see 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. Next, let's take a look at some comparisons between Docker Hub and Docker Registry to help you can make an educated decision as to which platform to choose to store your images.

Docker Registry has the following features:

- Host and manage your own registry from which you can serve all the repositories as private, public, or a mix between the two
- 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:

- 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

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
```

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
```

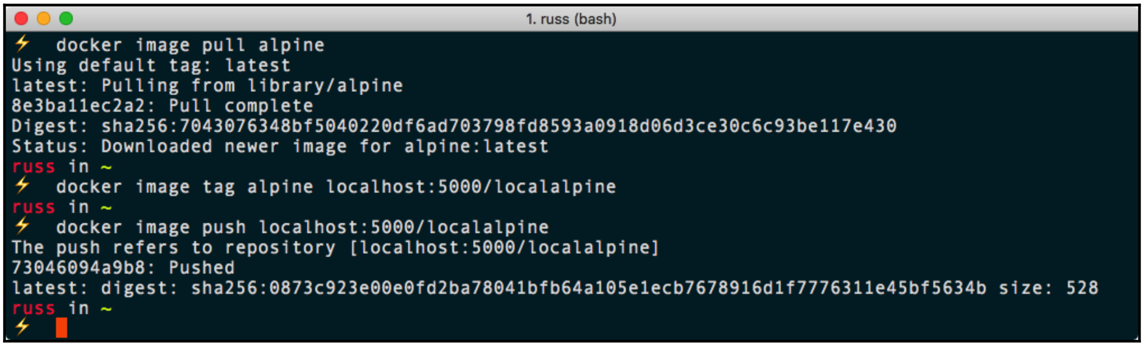
Now that we have a copy of the Alpine Linux image, we need to push it to our local Docker Registry, which is available at `localhost:5000`. To do this, we need to tag the Alpine Linux image with the URL of our local Docker Registry, along with a different image name:

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

Now that we have our image tagged, 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:

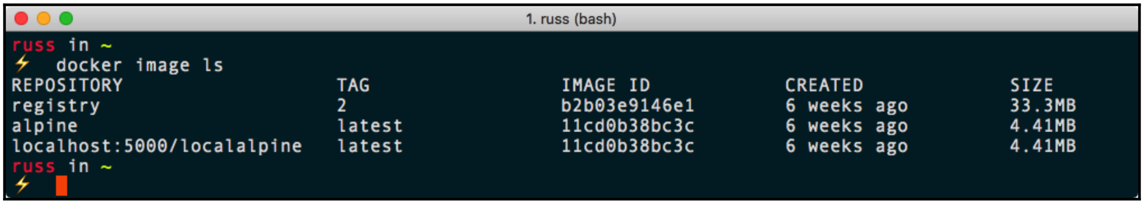


```
⚡  docker image pull alpine
Using default tag: latest
latest: Pulling from library/alpine
8e3ba11ec2a2: Pull complete
Digest: sha256:7043076348bf5040220df6ad703798fd8593a0918d06d3ce30c6c93be117e430
Status: Downloaded newer image for alpine:latest
russ in ~
⚡  docker image tag alpine localhost:5000/localalpine
russ in ~
⚡  docker image push localhost:5000/localalpine
The push refers to repository [localhost:5000/localalpine]
73046094a9b8: Pushed
latest: digest: sha256:0873c923e00e0fd2ba78041bfb64a105e1ecb7678916d1f7776311e45bf5634b size: 528
russ in ~
⚡
```

Try running the following command:

```
$ docker image ls
```

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



```
russ in ~
⚡   docker image ls
REPOSITORY                   TAG                 IMAGE ID            CREATED             SIZE
registry                     2                   b2b03e9146e1        6 weeks ago         33.3MB
alpine                       latest              11cd0b38bc3c        6 weeks ago         4.41MB
localhost:5000/localalpine   latest              11cd0b38bc3c        6 weeks ago         4.41MB
russ in ~
⚡   █
```

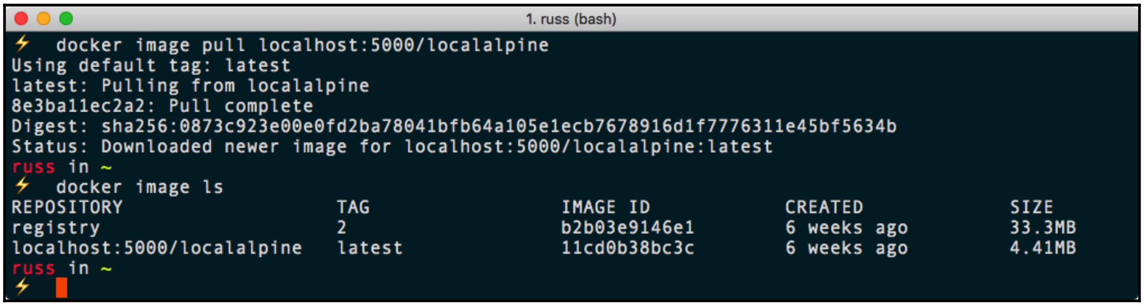
Before we pull the image back down from our local Docker Registry, we should remove the two local copies of the image. We need to use the `REPOSITORY` name to do this, rather than the `IMAGE ID`, as we have two images from two locations with the same ID, and Docker will throw an error:

```
$ 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
8e3ba11ec2a2: Pull complete
Digest: sha256:0873c923e00e0fd2ba78041bfb64a105e1ecb7678916d1f7776311e45bf5634b
Status: Downloaded newer image for localhost:5000/localalpine:latest
russ in ~
⚡   docker image ls
REPOSITORY                   TAG                 IMAGE ID            CREATED             SIZE
registry                     2                   b2b03e9146e1        6 weeks ago         33.3MB
localhost:5000/localalpine   latest              11cd0b38bc3c        6 weeks ago         4.41MB
russ in ~
⚡   █
```

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 around 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 it says; all 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 of the storage engines supported are all highly available, distributed object-level storage. We will look at these cloud services in a later chapter.

Docker Trusted Registry

One of the components that ships with the commercial **Docker Enterprise Edition** (**Docker EE**) is **Docker Trusted Registry** (**DTR**). 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 Docker

Third-party registries

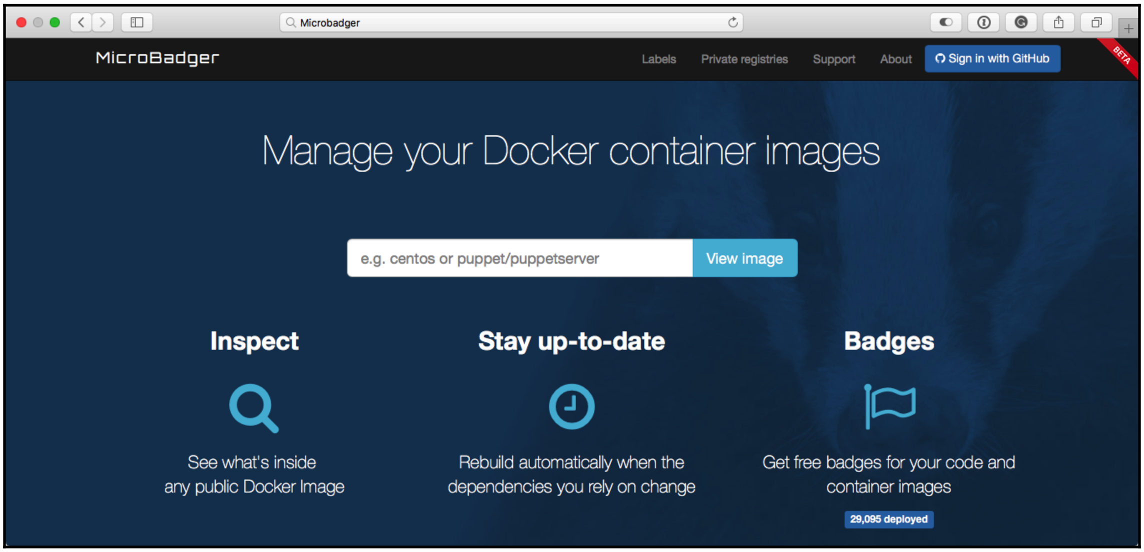
It is not only Docker that offers image registry services; companies such as Red Hat offer their own registry, where you can find the Red Hat Container Catalog, which hosts containerized versions of all of Red Hat's product offerings, along with containers to support its OpenShift offering.

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 by CoreOS, who are now owned by Red Hat, and also services from Amazon Web Services and Microsoft Azure. We will take a look at these services when we move on to looking at Docker in the cloud.

Microbadger

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

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



Manage your Docker container images

Inspect

See what's inside any public Docker Image

Stay up-to-date

Rebuild automatically when the dependencies you rely on change

Badges

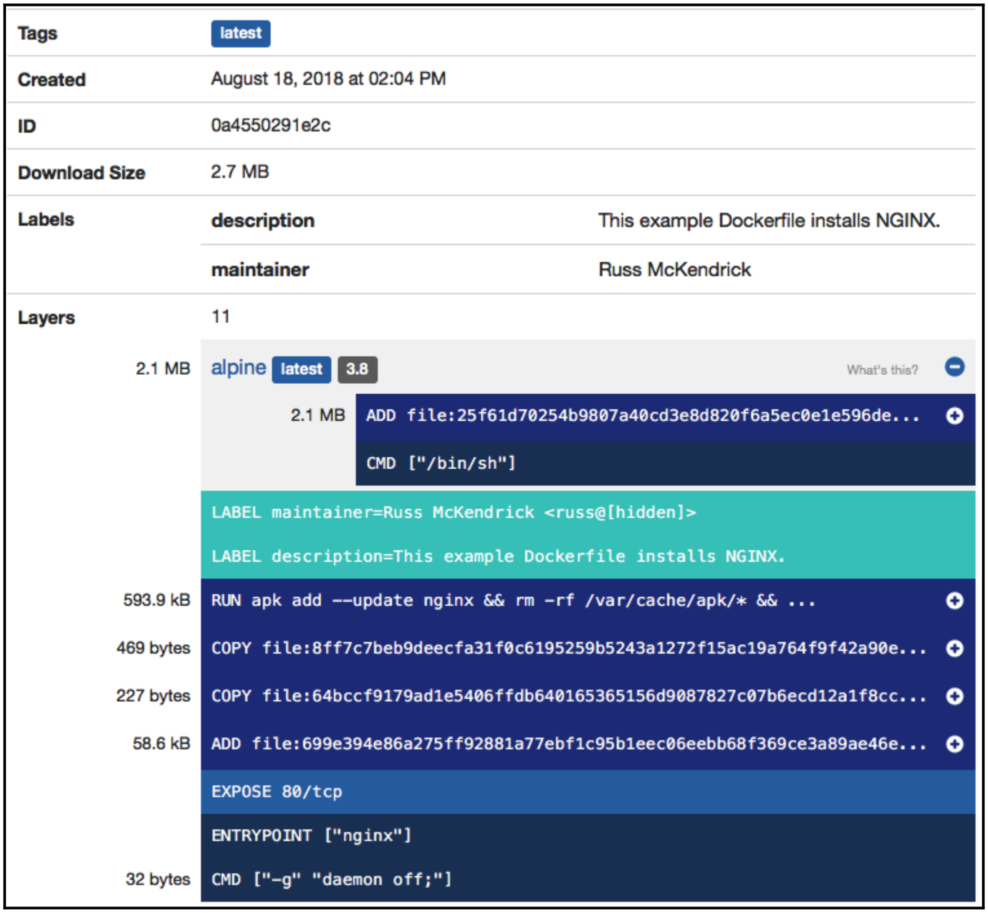
Get free badges for your code and container images

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

In this example, we are going to search for the `masteringdockerthirdedition/dockerfiles-example` image that we pushed earlier in the chapter, and select the latest tag. As you can see from the following screenshot, Docker Hub is automatically searched with results returned in real time as you type.

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 you can see from the following screenshot, Microbadger presents information on how many layers your image contains:

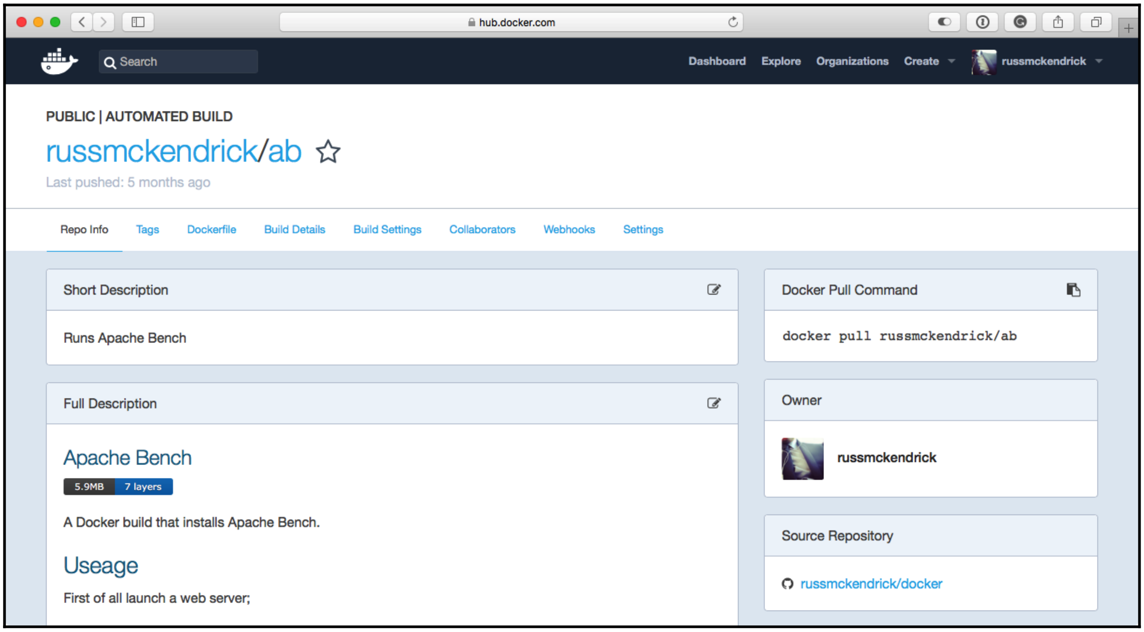


Tags	latest	
Created	August 18, 2018 at 02:04 PM	
ID	0a4550291e2c	
Download Size	2.7 MB	
Labels	**description**	This example Dockerfile installs NGINX.
	maintainer	Russ McKendrick
Layers	11	

2.1 MB	alpine latest 3.8	What's this?
	2.1 MB	`ADD file:25f61d70254b9807a40cd3e8d820f6a5ec0e1e596de...`
		`CMD ["/bin/sh"]`
	`LABEL maintainer=Russ McKendrick <russ@[hidden]>`	
	`LABEL description=This example Dockerfile installs NGINX.`	
593.9 kB	`RUN apk add --update nginx && rm -rf /var/cache/apk/* && ...`	
469 bytes	`COPY file:8ff7c7beb9deecfa31f0c6195259b5243a1272f15ac19a764f9f42a90e...`	
227 bytes	`COPY file:64bccf9179ad1e5406ffdb640165365156d9087827c07b6ecd12a1f8cc...`	
58.6 kB	`ADD file:699e394e86a275ff92881a77ebf1c95b1eec06eebb68f369ce3a89ae46e...`	
	`EXPOSE 80/tcp`	
	`ENTRYPOINT ["nginx"]`	
32 bytes	`CMD ["-g" "daemon off;"]`	

By showing the size of each layer and the Dockerfile command 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 is that Microbadger gives you the option of embedding basic statistics about your images in your Git repository or Docker Hub; for example, the following screen shows the Docker Hub page for one of my own images:



PUBLIC | AUTOMATED BUILD

russmckendrick/ab

Last pushed: 5 months ago

Short Description

Runs Apache Bench

Docker Pull Command

```
docker pull russmckendrick/ab
```

Full Description

Apache Bench

5.9MB 7 layers

A Docker build that installs Apache Bench.

Useage

First of all launch a web server;

Owner

russmckendrick

Source Repository

russmckendrick/docker

As you can see, Microbadger is displaying the overall size of the image, which in this example is 5.9MB, as well as the total number of layers the image is made up of, which is 7. The Microbadger service is still in beta 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 you can both manually and automatically build container images using Docker Hub. We discussed the various registries you can use besides Docker Hub, such as the Docker Store and Red Hat's container catalog.

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 you to display information about your remotely hosted container images.

In the next chapter, we are going to look at how we can 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 also logs you into the desktop application?
5. How would you delete two images that share the same IMAGE ID?
6. Which port does the Docker Registry run on by default?

Further reading

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

- Docker Store Publisher Sign-up: `https://store.docker.com/publisher/signup/`
- Docker Trusted Registry (DTR): `https://docs.docker.com/ee/dtr/`
- Docker Registry Documentation: `https://docs.docker.com/registry/`

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

- 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: This uses OpenStack Swift: `https://wiki.openstack.org/wiki/Swift`

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

- Red Hat Container Catalog: `https://access.redhat.com/containers/`
- OpenShift: `https://www.openshift.com/`
- Artifactory by JFrog: `https://www.jfrog.com/artifactory/`
- Quay: `https://quay.io/`

Finally, you can find links to the 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 the first chapter 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:

- Docker container commands:
  - The basics
  - Interacting with your containers
  - Logs and process information
  - Resource limits
  - Container states and miscellaneous commands
  - Removing containers
- Docker networking and volumes

Technical requirements

In this chapter, we will continue to use our local Docker installation. As before, 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 be applicable only to macOS and Linux-based operating systems.

Check out the following video to see the Code in Action:

http://bit.ly/2yupP3n

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

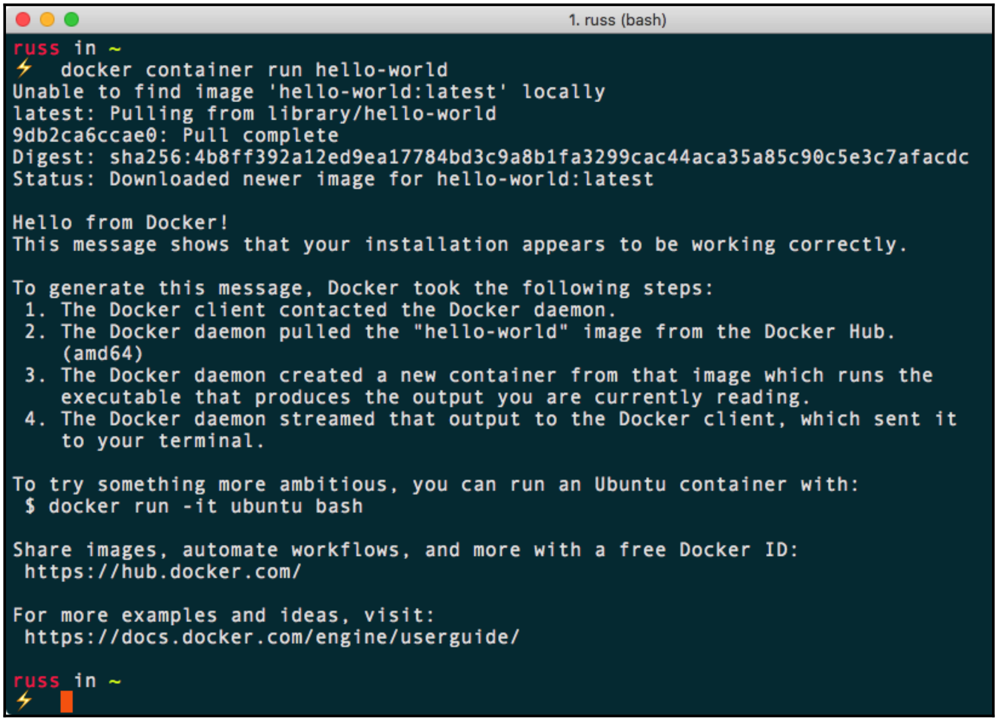
In Chapter 1, *Docker Overview*, we launched the most basic container of all, the hello-world container, using the following command:

```
$ 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 Store page for the image at https://store.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:

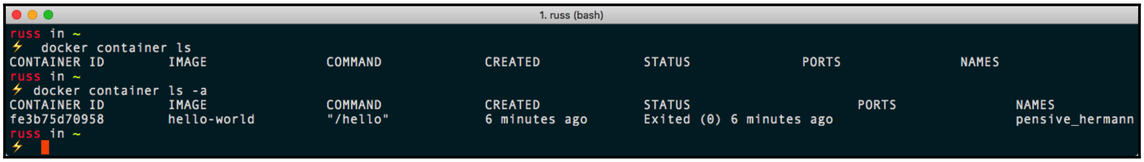


```
russ in ~
⚡  docker container run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
9db2ca6ccae0: Pull complete
Digest: sha256:4b8ff392a12ed9ea17784bd3c9a8b1fa3299cac44aca35a85c90c5e3c7afacdc
Status: Downloaded newer image for hello-world:latest

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/engine/userguide/

russ in ~
⚡
```

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

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

The output of the command is given here:



```
russ in ~
⚡  docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
russ in ~
⚡ docker container ls -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS               NAMES
fe3b75d70958        hello-world         "/hello"            6 minutes ago       Exited (0) 6 minutes ago                       pensive_hermann
russ in ~
⚡
```

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.

We didn't have to name our container as it wasn't 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 `pensive_hermann`.

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. 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).

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!

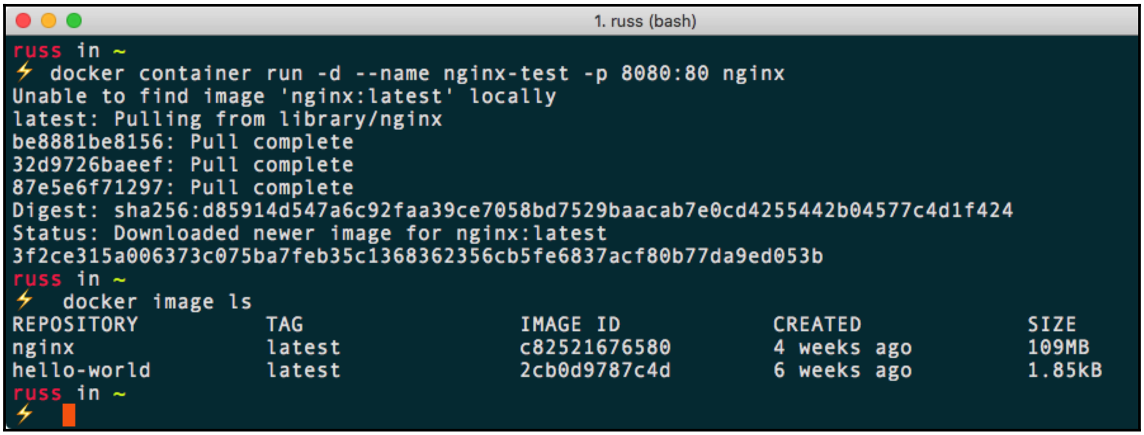
We can remove the container with a status of `exited` by running the following command, making sure that you replace the name of the container with your own container name:

```
$ docker container rm pensive_hermann
```

Also, at the end of `Chapter 1`, *Docker Overview*, we launched a container using the official nginx image, 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`:

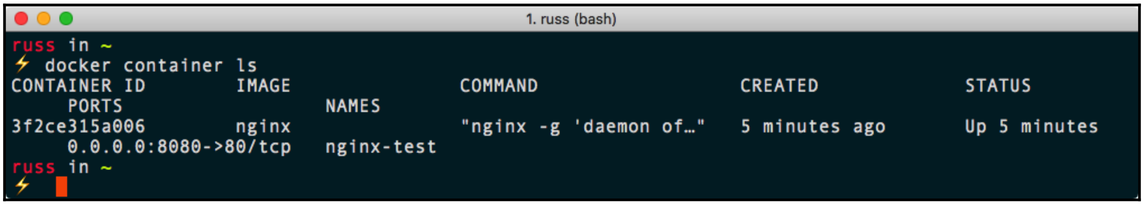


```
russ in ~
⚡ docker container run -d --name nginx-test -p 8080:80 nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
be8881be8156: Pull complete
32d9726baeef: Pull complete
87e5e6f71297: Pull complete
Digest: sha256:d85914d547a6c92faa39ce7058bd7529baacab7e0cd4255442b04577c4d1f424
Status: Downloaded newer image for nginx:latest
3f2ce315a006373c075ba7feb35c1368362356cb5fe6837acf80b77da9ed053b
russ in ~
⚡  docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
nginx               latest              c82521676580        4 weeks ago         109MB
hello-world         latest              2cb0d9787c4d        6 weeks ago         1.85kB
russ in ~
⚡
```

As you can see, running `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:



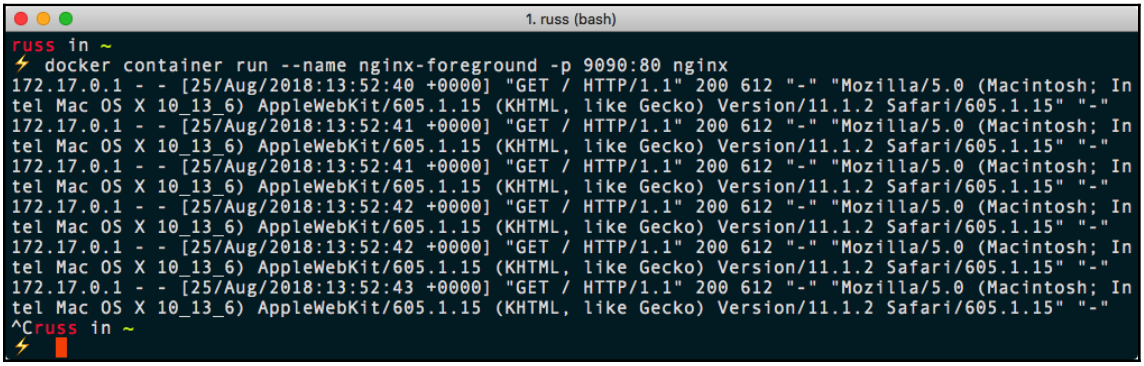
```
russ in ~
⚡ docker container ls
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
     PORTS                  NAMES
3f2ce315a006        nginx               "nginx -g 'daemon of…"   5 minutes ago       Up 5 minutes
     0.0.0.0:8080->80/tcp   nginx-test
russ in ~
⚡
```

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:

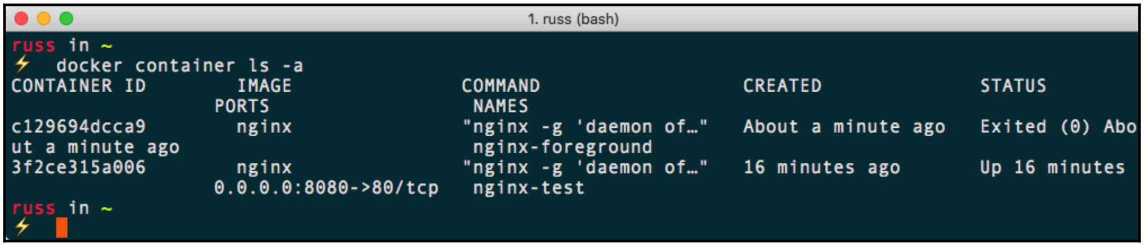
```
$ docker container run --name nginx-foreground -p 9090:80 nginx
```

Once launched, open a browser and go to `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.



```
russ in ~
⚡ docker container run --name nginx-foreground -p 9090:80 nginx
172.17.0.1 - - [25/Aug/2018:13:52:40 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:13:52:41 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:13:52:41 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:13:52:42 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:13:52:42 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:13:52:43 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
^Cruss in ~
⚡
```

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



```
russ in ~
⚡  docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED              STATUS
                  PORTS                  NAMES
c129694dcca9        nginx               "nginx -g 'daemon of…"   About a minute ago   Exited (0) Abo
ut a minute ago                          nginx-foreground
3f2ce315a006        nginx               "nginx -g 'daemon of…"   16 minutes ago       Up 16 minutes
                  0.0.0.0:8080->80/tcp   nginx-test
russ in ~
⚡  
```

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.

> 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 to. Standard error (`stderr`) is where the process writes error messages to.

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
(3f5b355607f24e03f09a60ee688645f223bafe4492f807459e4a2b83571f23f4): 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.

Interacting with your containers

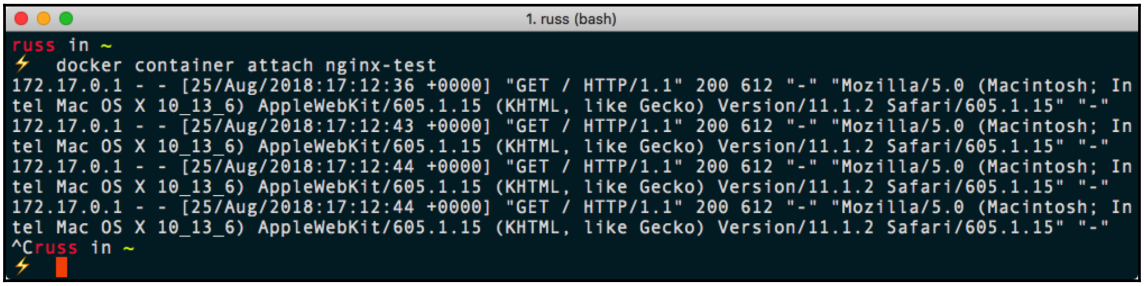
So far, our containers have been running a single process. Docker provides you with a few tools that enable you to both fork additional processes and interact with them.

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 screen, just as 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:



```
russ in ~
⚡  docker container attach nginx-test
172.17.0.1 - - [25/Aug/2018:17:12:36 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:17:12:43 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:17:12:44 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:17:12:44 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
^Cruss in ~
⚡
```

We can start our container back up by running the following command:

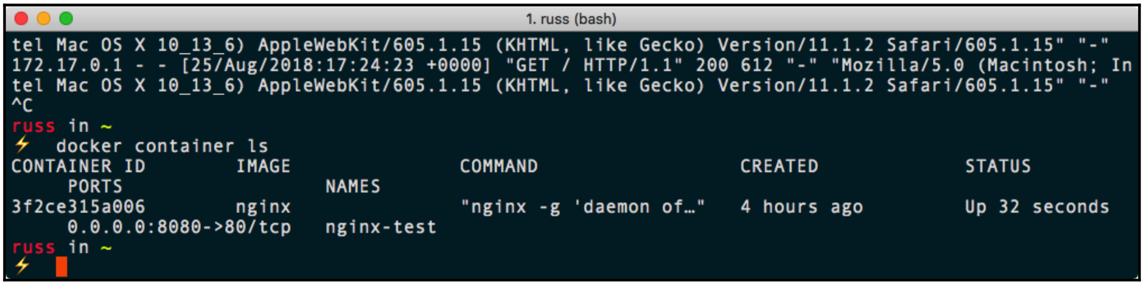
```
$ docker container start nginx-test
```

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`:



```
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:17:24:23 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
^C
russ in ~
⚡  docker container ls
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
     PORTS                  NAMES
3f2ce315a006        nginx               "nginx -g 'daemon of…"   4 hours ago         Up 32 seconds
     0.0.0.0:8080->80/tcp   nginx-test
russ in ~
⚡  
```

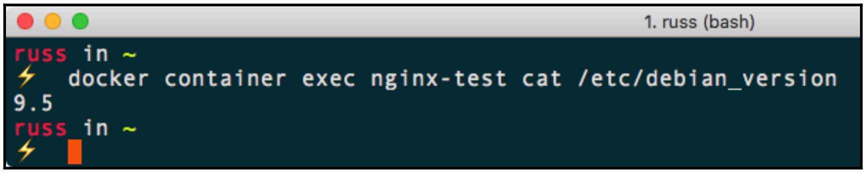
exec

The `attach` command is useful if you need to connect to the process your container is running, but what if you need something 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/debian_version` to `stdout`. The second process will then terminate, leaving our container as it was before the exec command was executed:



```
russ in ~
⚡  docker container exec nginx-test cat /etc/debian_version
9.5
russ in ~
⚡
```

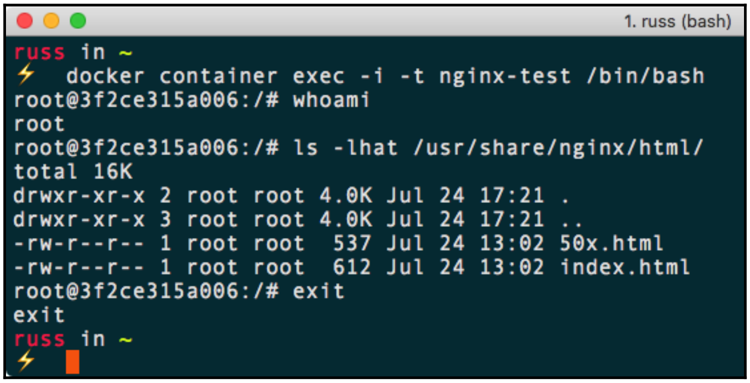
We can take this one step further by running the following command:

```
$ 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.

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:



```
russ in ~
⚡  docker container exec -i -t nginx-test /bin/bash
root@3f2ce315a006:/# whoami
root
root@3f2ce315a006:/# ls -lhat /usr/share/nginx/html/
total 16K
drwxr-xr-x 2 root root 4.0K Jul 24 17:21 .
drwxr-xr-x 3 root root 4.0K Jul 24 17:21 ..
-rw-r--r-- 1 root root  537 Jul 24 13:02 50x.html
-rw-r--r-- 1 root root  612 Jul 24 13:02 index.html
root@3f2ce315a006:/# exit
exit
russ in ~
⚡
```

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 12`, *Docker Workflows*.

Logs and process information

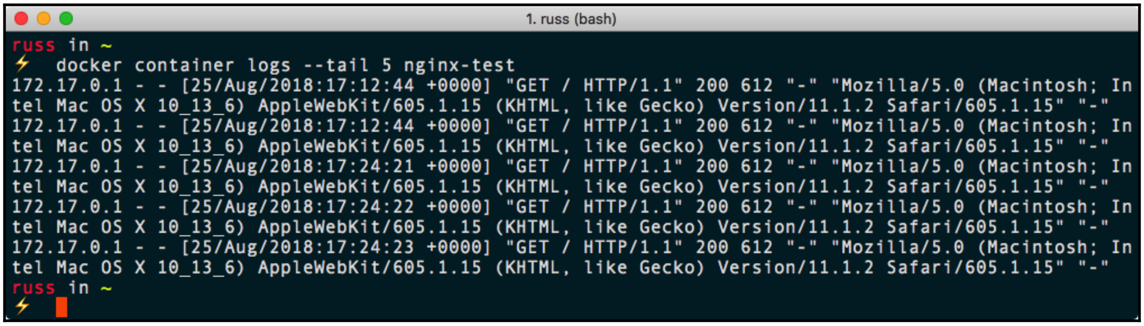
So far, we have been attaching to either the process in our container, or the container itself, to view information. Docker provides a few commands to allow you to view information about your containers without having to use either the `attach` or `exec` commands.

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:



```
russ in ~
⚡  docker container logs --tail 5 nginx-test
172.17.0.1 - - [25/Aug/2018:17:12:44 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:17:12:44 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:17:24:21 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:17:24:22 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:17:24:23 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
russ in ~
⚡
```

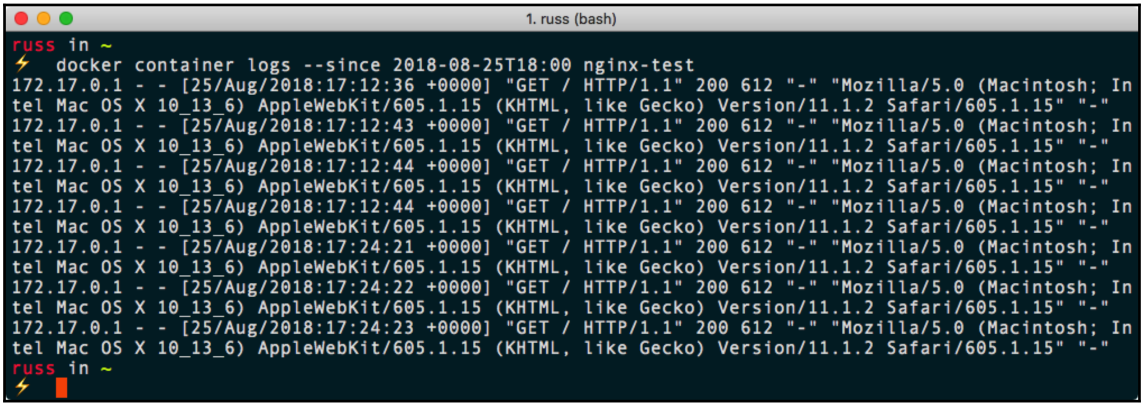
To view the logs in real time, I simply need to run the following:

```
$ docker container logs -f nginx-test
```

The `-f` flag is shorthand for `--follow`. I can also, say, view everything that has been logged since a certain time by running the following:

```
$ docker container logs --since 2018-08-25T18:00 nginx-test
```

The output of the command is shown here:



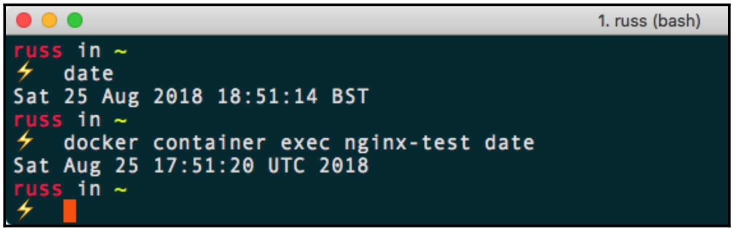
```
russ in ~
⚡  docker container logs --since 2018-08-25T18:00 nginx-test
172.17.0.1 - - [25/Aug/2018:17:12:36 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:17:12:43 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:17:12:44 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:17:12:44 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:17:24:21 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:17:24:22 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
172.17.0.1 - - [25/Aug/2018:17:24:23 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Macintosh; In
tel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1.2 Safari/605.1.15" "-"
russ in ~
⚡
```

You might notice that, in the preceding output, the timestamp in the access log is 17:12, which is before 18:00. Why is that?

The `logs` command shows the timestamps of `stdout` as recorded by Docker, and not the time within the container. You can see this when I run the following commands:

```
$ date
$ docker container exec nginx-test date
```

The output is shown here:



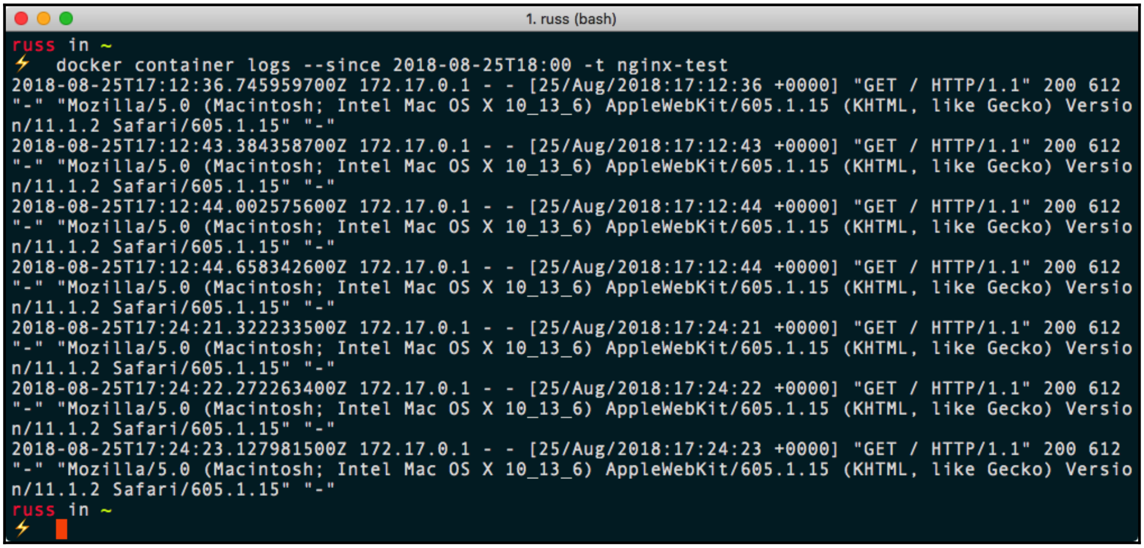
```
russ in ~
⚡  date
Sat 25 Aug 2018 18:51:14 BST
russ in ~
⚡  docker container exec nginx-test date
Sat Aug 25 17:51:20 UTC 2018
russ in ~
⚡
```

There is an hour's time difference between my host machine and the container due to **British Summer Time** (**BST**) being in effect on my host.

Luckily, to save confusion—or add to it, depending on how you look at it—you can add `-t` to your `logs` commands:

```
$ docker container logs --since 2018-08-25T18:00 -t nginx-test
```

The -t flag is short for --timestamp; this option prepends the time the output was captured by Docker:



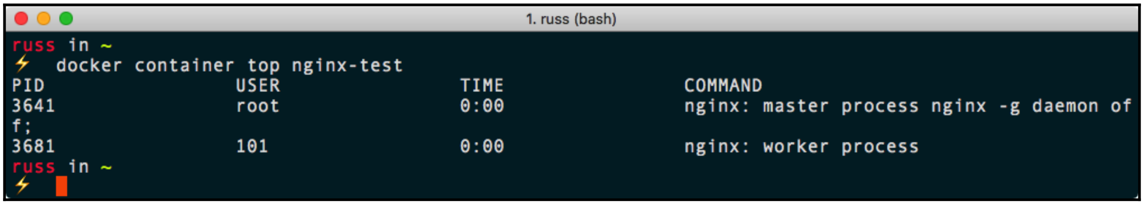
```
russ in ~
⚡  docker container logs --since 2018-08-25T18:00 -t nginx-test
2018-08-25T17:12:36.745959700Z 172.17.0.1 - - [25/Aug/2018:17:12:36 +0000] "GET / HTTP/1.1" 200 612
"-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Versio
n/11.1.2 Safari/605.1.15" "-"
2018-08-25T17:12:43.384358700Z 172.17.0.1 - - [25/Aug/2018:17:12:43 +0000] "GET / HTTP/1.1" 200 612
"-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Versio
n/11.1.2 Safari/605.1.15" "-"
2018-08-25T17:12:44.002575600Z 172.17.0.1 - - [25/Aug/2018:17:12:44 +0000] "GET / HTTP/1.1" 200 612
"-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Versio
n/11.1.2 Safari/605.1.15" "-"
2018-08-25T17:12:44.658342600Z 172.17.0.1 - - [25/Aug/2018:17:12:44 +0000] "GET / HTTP/1.1" 200 612
"-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Versio
n/11.1.2 Safari/605.1.15" "-"
2018-08-25T17:24:21.322233500Z 172.17.0.1 - - [25/Aug/2018:17:24:21 +0000] "GET / HTTP/1.1" 200 612
"-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Versio
n/11.1.2 Safari/605.1.15" "-"
2018-08-25T17:24:22.272263400Z 172.17.0.1 - - [25/Aug/2018:17:24:22 +0000] "GET / HTTP/1.1" 200 612
"-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Versio
n/11.1.2 Safari/605.1.15" "-"
2018-08-25T17:24:23.127981500Z 172.17.0.1 - - [25/Aug/2018:17:24:23 +0000] "GET / HTTP/1.1" 200 612
"-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Versio
n/11.1.2 Safari/605.1.15" "-"
russ in ~
⚡
```

top

The top command is quite a simple one; it lists the processes running within the container you specify, and is used as follows:

```
$ docker container top nginx-test
```

The output of the command is shown here:



```
russ in ~
⚡  docker container top nginx-test
PID                 USER                TIME                COMMAND
3641                root                0:00                nginx: master process nginx -g daemon of
f;
3681                101                 0:00                nginx: worker process
russ in ~
⚡
```

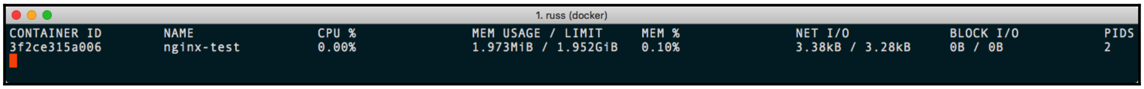
As you can see from the following Terminal output, we have two processes running, both of which are nginx, which is to be expected.

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:

```
$ docker container stats nginx-test
```

As you can see from the following Terminal output, we are given information on the `CPU`, `RAM`, `NETWORK`, `DISK IO`, and `PIDS` for the specified container:



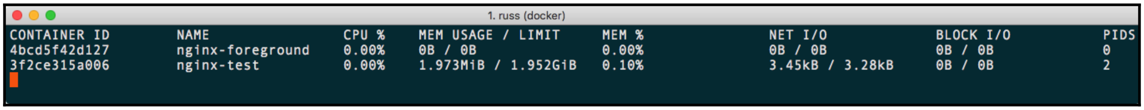
1. russ (docker)

```
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O           PIDS
3f2ce315a006        nginx-test          0.00%               1.973MiB / 1.952GiB   0.10%               3.38kB / 3.28kB     0B / 0B             2
```

We can also pass the `-a` flag; this is short for `--all` and displays all containers, running or not. For example, try running the following command:

```
$ docker container stats -a
```

You should receive something like the following output:



1. russ (docker)

```
CONTAINER ID        NAME                CPU %     MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O           PIDS
4bcd5f42d127        nginx-foreground    0.00%     0B / 0B               0.00%               0B / 0B             0B / 0B             0
3f2ce315a006        nginx-test          0.00%     1.973MiB / 1.952GiB   0.10%               3.45kB / 3.28kB     0B / 0B             2
```

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 for resource utilization in later chapters.

Resource limits

The last 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 requires it. We can put caps 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
3f2ce315a006373c075ba7feb35c1368362356cb5fe6837acf80b77da9ed053b: 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:

```
$ docker container inspect nginx-test
```

As you can see by running the preceding command, there is a lot of configuration data. 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,
    "MemoryReservation": 0,
    "MemorySwap": 0,
    "MemorySwappiness": null,
```

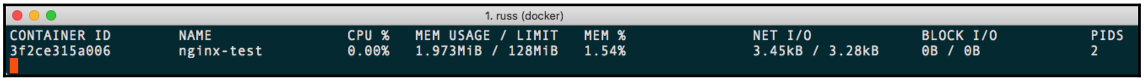
Everything is set to 0, so how can 128M be smaller than 0?

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 the following:

```
$ docker container update --cpu-shares 512 --memory 128M --memory-swap 256M
nginx-test
```

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 the `--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
3f2ce315a006        nginx-test          0.00%     1.973MiB / 128MiB     1.54%               3.45kB / 3.28kB     0B / 0B             2
```

Also, re-running `docker container inspect nginx-test | grep -i memory` will show the changes as follows:

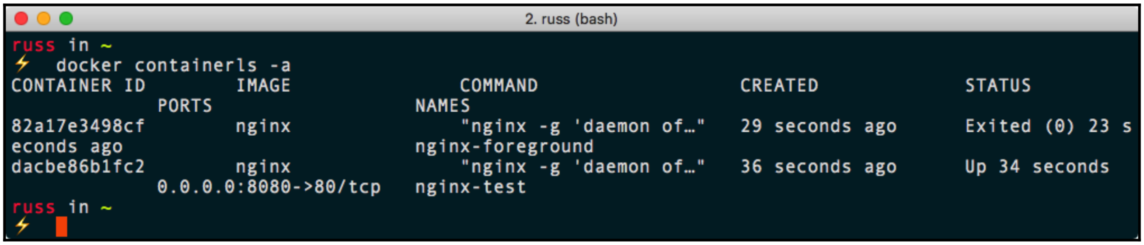
```
"Memory": 134217728,
"KernelMemory": 0,
"MemoryReservation": 0,
"MemorySwap": 268435456,
"MemorySwappiness": null,
```

The values when running `docker container inspect` are all shown in bytes rather 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:

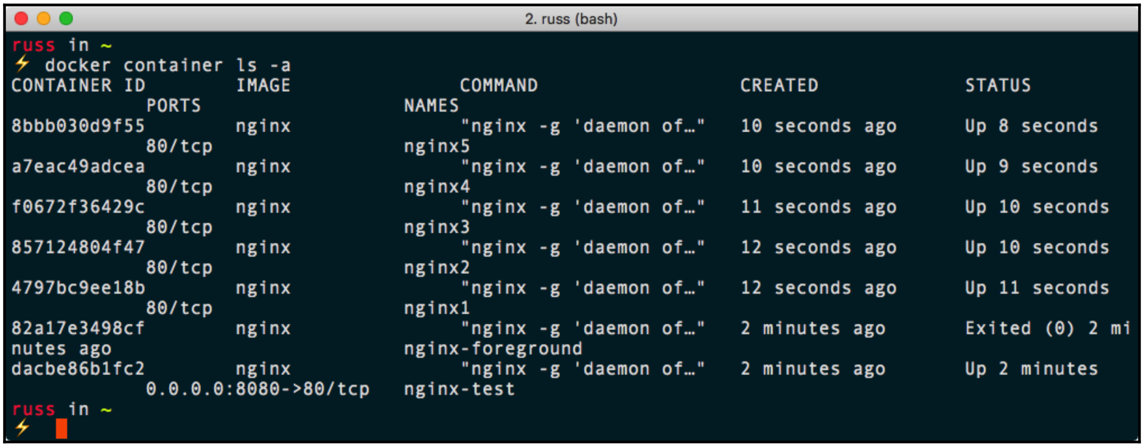


```
russ in ~
⚡  docker containerls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
             PORTS                  NAMES
82a17e3498cf        nginx               "nginx -g 'daemon of…"   29 seconds ago      Exited (0) 23 s
econds ago                          nginx-foreground
dacbe86b1fc2        nginx               "nginx -g 'daemon of…"   36 seconds ago      Up 34 seconds
             0.0.0.0:8080->80/tcp   nginx-test
russ in ~
⚡  
```

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
```

When running `docker container ls -a`, you should see your five new containers, named `nginx1` through to `nginx5`:



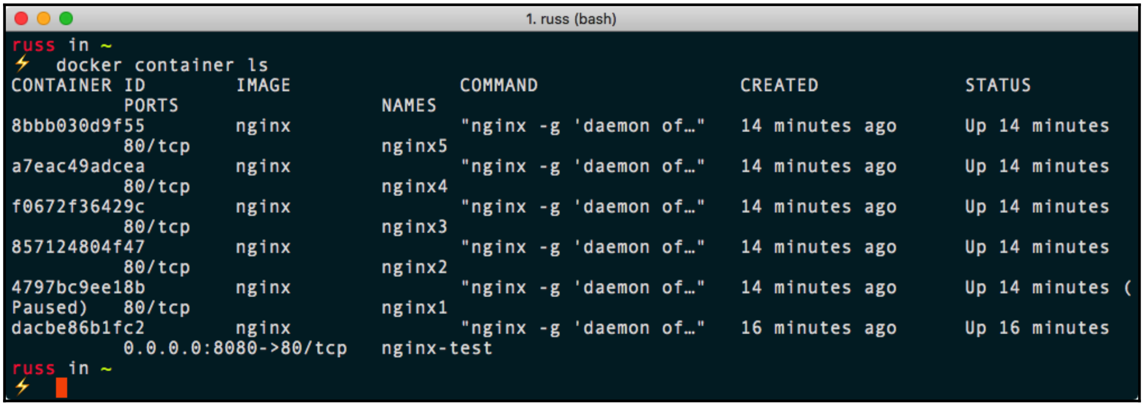
```
russ in ~
⚡ docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
            PORTS                  NAMES
8bbb030d9f55        nginx               "nginx -g 'daemon of…"   10 seconds ago      Up 8 seconds
            80/tcp                 nginx5
a7eac49adcea        nginx               "nginx -g 'daemon of…"   10 seconds ago      Up 9 seconds
            80/tcp                 nginx4
f0672f36429c        nginx               "nginx -g 'daemon of…"   11 seconds ago      Up 10 seconds
            80/tcp                 nginx3
857124804f47        nginx               "nginx -g 'daemon of…"   12 seconds ago      Up 10 seconds
            80/tcp                 nginx2
4797bc9ee18b        nginx               "nginx -g 'daemon of…"   12 seconds ago      Up 11 seconds
            80/tcp                 nginx1
82a17e3498cf        nginx               "nginx -g 'daemon of…"   2 minutes ago       Exited (0) 2 mi
nutes ago                          nginx-foreground
dacbe86b1fc2        nginx               "nginx -g 'daemon of…"   2 minutes ago       Up 2 minutes
            0.0.0.0:8080->80/tcp   nginx-test
russ in ~
⚡ 
```

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`:



```
russ in ~
⚡  docker container ls
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
          PORTS               NAMES
8bbb030d9f55        nginx               "nginx -g 'daemon of…"   14 minutes ago      Up 14 minutes
          80/tcp              nginx5
a7eac49adcea        nginx               "nginx -g 'daemon of…"   14 minutes ago      Up 14 minutes
          80/tcp              nginx4
f0672f36429c        nginx               "nginx -g 'daemon of…"   14 minutes ago      Up 14 minutes
          80/tcp              nginx3
857124804f47        nginx               "nginx -g 'daemon of…"   14 minutes ago      Up 14 minutes
          80/tcp              nginx2
4797bc9ee18b        nginx               "nginx -g 'daemon of…"   14 minutes ago      Up 14 minutes (
Paused)   80/tcp              nginx1
dacbe86b1fc2        nginx               "nginx -g 'daemon of…"   16 minutes ago      Up 16 minutes
          0.0.0.0:8080->80/tcp   nginx-test
russ in ~
⚡
```

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.

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 in 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 a `SIGTERM`. If the process has not terminated itself within a grace period, then a kill signal, called a `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`, 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:

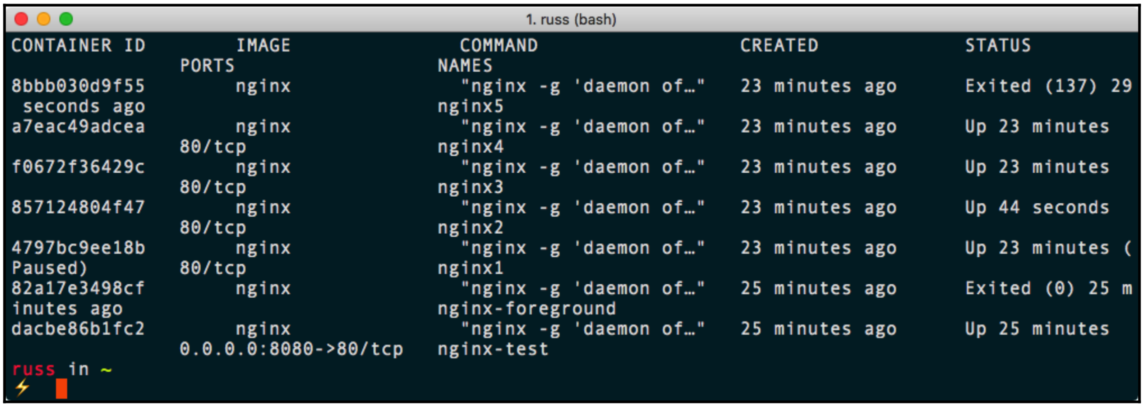
```
$ docker container restart -t 60 nginx4
```

Finally, you also have the option of sending a `SIGKILL` command immediately to the container by running the `kill` command:

```
$ docker container kill nginx5
```

Removing containers

Let's check the containers we have running using 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:

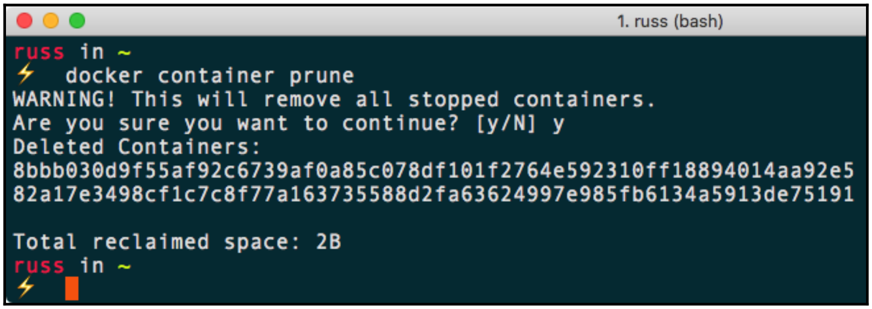


```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
               PORTS                  NAMES
8bbb030d9f55        nginx               "nginx -g 'daemon of…"   23 minutes ago      Exited (137) 29
 seconds ago                          nginx5
a7eac49adcea        nginx               "nginx -g 'daemon of…"   23 minutes ago      Up 23 minutes
               80/tcp                 nginx4
f0672f36429c        nginx               "nginx -g 'daemon of…"   23 minutes ago      Up 23 minutes
               80/tcp                 nginx3
857124804f47        nginx               "nginx -g 'daemon of…"   23 minutes ago      Up 44 seconds
               80/tcp                 nginx2
4797bc9ee18b        nginx               "nginx -g 'daemon of…"   23 minutes ago      Up 23 minutes (
Paused)        80/tcp                 nginx1
82a17e3498cf        nginx               "nginx -g 'daemon of…"   25 minutes ago      Exited (0) 25 m
inutes ago                            nginx-foreground
dacbe86b1fc2        nginx               "nginx -g 'daemon of…"   25 minutes ago      Up 25 minutes
               0.0.0.0:8080->80/tcp   nginx-test
russ in ~
⚡
```

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 in ~
⚡  docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
8bbb030d9f55af92c6739af0a85c078df101f2764e592310ff18894014aa92e5
82a17e3498cf1c7c8f77a163735588d2fa63624997e985fb6134a5913de75191

Total reclaimed space: 2B
russ in ~
⚡
```

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.

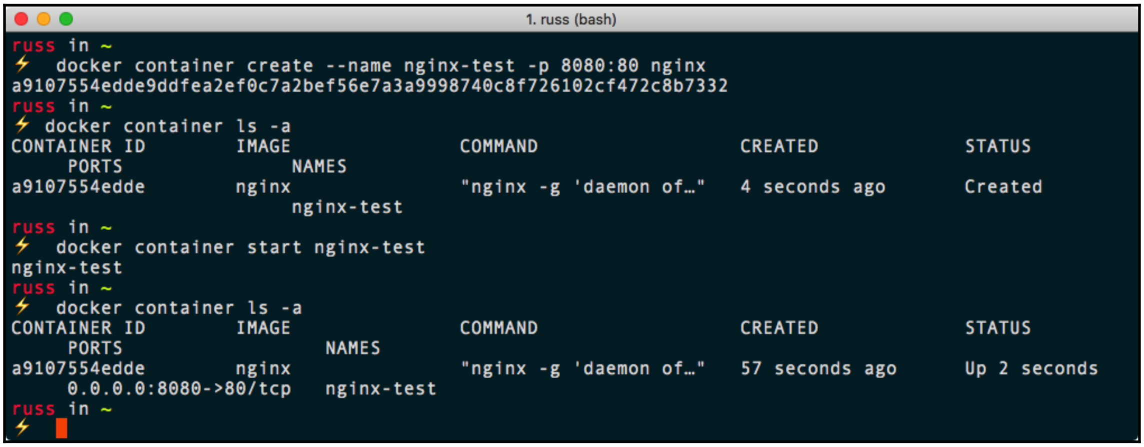
Miscellaneous commands

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:



```
russ in ~
⚡  docker container create --name nginx-test -p 8080:80 nginx
a9107554edde9ddfea2ef0c7a2bef56e7a3a9998740c8f726102cf472c8b7332
russ in ~
⚡ docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
     PORTS               NAMES
a9107554edde        nginx               "nginx -g 'daemon of…"   4 seconds ago       Created
                         nginx-test
russ in ~
⚡  docker container start nginx-test
nginx-test
russ in ~
⚡  docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
     PORTS                  NAMES
a9107554edde        nginx               "nginx -g 'daemon of…"   57 seconds ago      Up 2 seconds
     0.0.0.0:8080->80/tcp   nginx-test
russ in ~
⚡ 
```

The next command we are going to quickly look at is the `port` command; this displays the port 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 final 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 on 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:

```
$ 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/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.

Again, if you are following along, you should remove any running containers launched during this section using the command of your choice.

Docker networking and volumes

Before we finish off this chapter, 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.

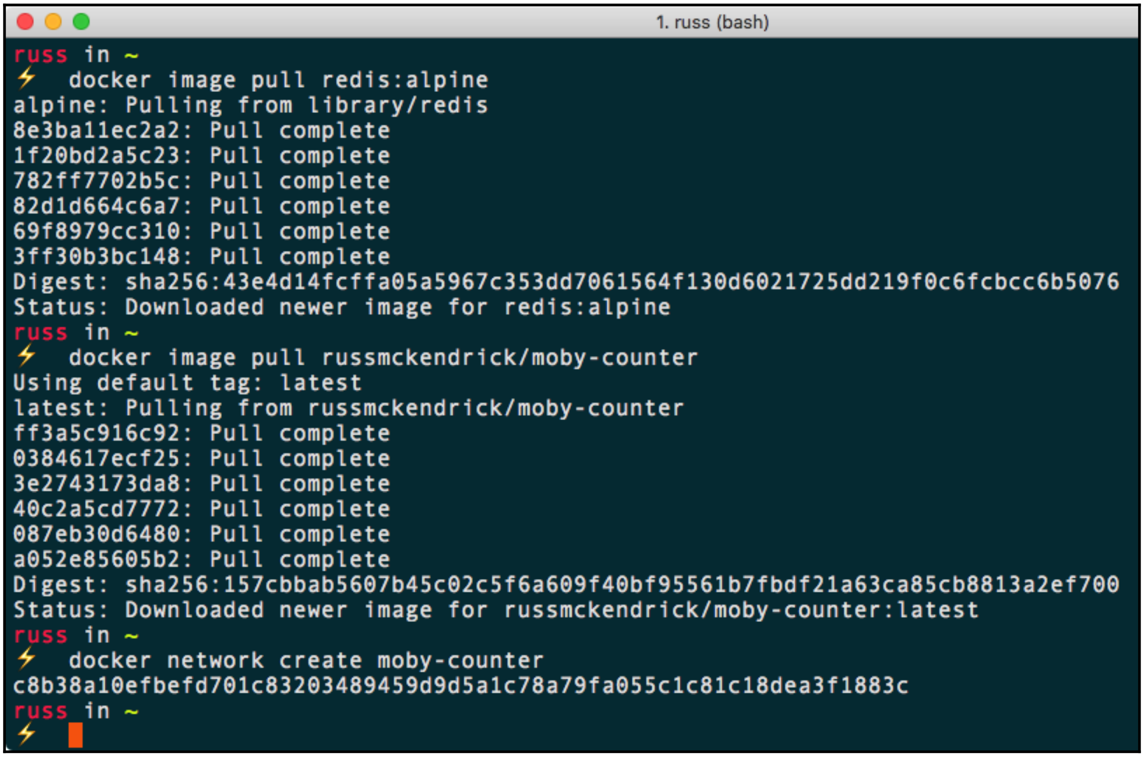
Rather than going into detail now, let's work through an example. We are going to be running a two-container application; the first container will be running Redis, and the second, our application, which uses the Redis container to store a system state.

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:



```
russ in ~
⚡  docker image pull redis:alpine
alpine: Pulling from library/redis
8e3ba11ec2a2: Pull complete
1f20bd2a5c23: Pull complete
782ff7702b5c: Pull complete
82d1d664c6a7: Pull complete
69f8979cc310: Pull complete
3ff30b3bc148: Pull complete
Digest: sha256:43e4d14fcffa05a5967c353dd7061564f130d6021725dd219f0c6fcbcc6b5076
Status: Downloaded newer image for redis:alpine
russ in ~
⚡  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
087eb30d6480: Pull complete
a052e85605b2: Pull complete
Digest: sha256:157cbbab5607b45c02c5f6a609f40bf95561b7fbdf21a63ca85cb8813a2ef700
Status: Downloaded newer image for russmckendrick/moby-counter:latest
russ in ~
⚡  docker network create moby-counter
c8b38a10efbefd701c83203489459d9d5a1c78a79fa055c1c81c18dea3f1883c
russ in ~
⚡
```

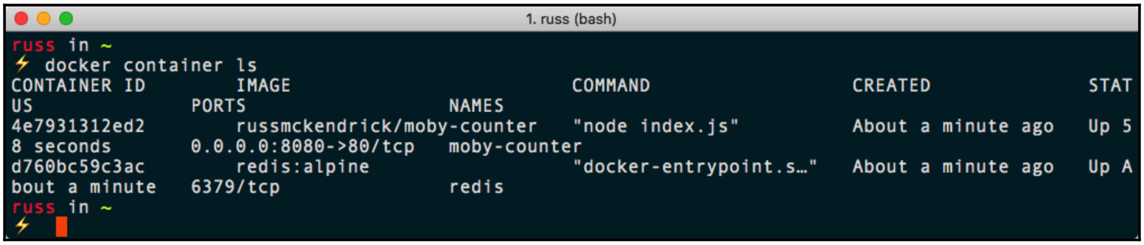
Now that we have our images pulled and our network created, we can launch our containers, starting with the Redis one:

```
$ 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:

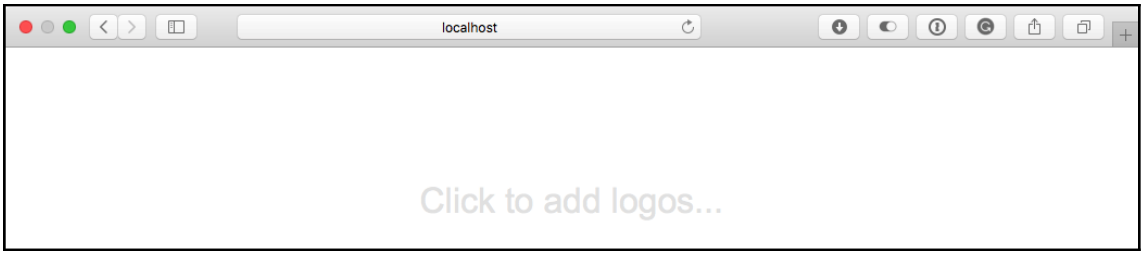
```
$ docker container run -d --name moby-counter --network moby-counter -p
8080:80 russmckendrick/moby-counter
```

Again, we launched the container into 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`:

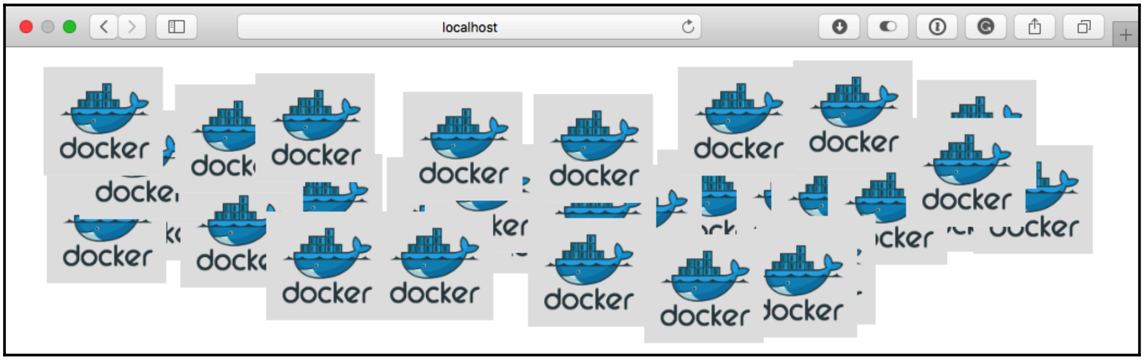


```
russ in ~
⚡ docker container ls
CONTAINER ID        IMAGE                            COMMAND                   CREATED               STAT
US                PORTS                  NAMES
4e7931312ed2        russmckendrick/moby-counter      "node index.js"           About a minute ago    Up 5
8 seconds         0.0.0.0:8080->80/tcp   moby-counter
d760bc59c3ac        redis:alpine                     "docker-entrypoint.s…"    About a minute ago    Up A
bout a minute     6379/tcp               redis
russ in ~
⚡
```

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**:



Clicking anywhere on the page will add Docker logos, so click away:



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 on-screen coordinates of each of the logos that you place on the screen by clicking.

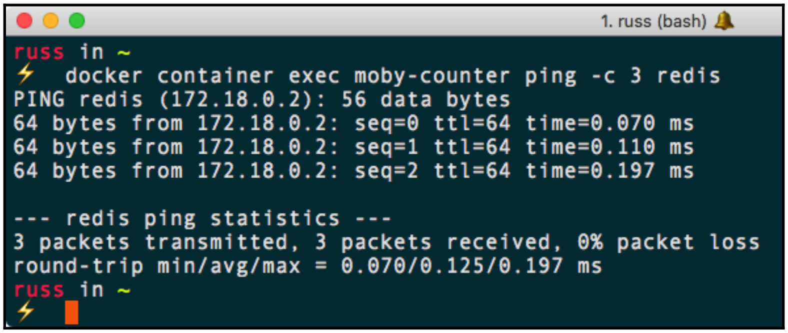
How is the moby-counter application connecting to the redis container? Well, in the server.js file, the following default values are being set:

```
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:



```
russ in ~
⚡  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=0.070 ms
64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.110 ms
64 bytes from 172.18.0.2: seq=2 ttl=64 time=0.197 ms

--- redis ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.125/0.197 ms
russ in ~
⚡
```

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 contents 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 4e7931312ed2
```

Other than the entry at the end, which is actually the IP address resolving to the hostname of the local container, `4e7931312ed2` 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:

```
$ 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
```

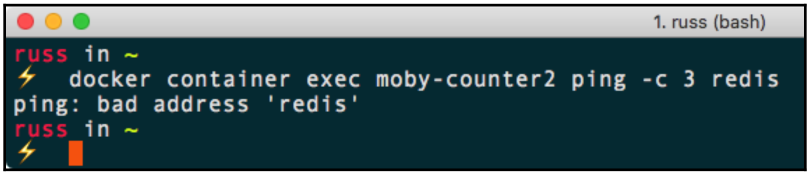
Let's create a second network and launch another application container:

```
$ 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:



```
russ in ~
⚡  docker container exec moby-counter2 ping -c 3 redis
ping: bad address 'redis'
russ in ~
⚡
```

Let's check the `resolv.conf` file to see if 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, so it returns a bad address error:

```
$ docker container exec moby-counter2 nslookup redis 127.0.0.11
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, but Docker has you covered.

While you cannot have two containers with the same names, 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:

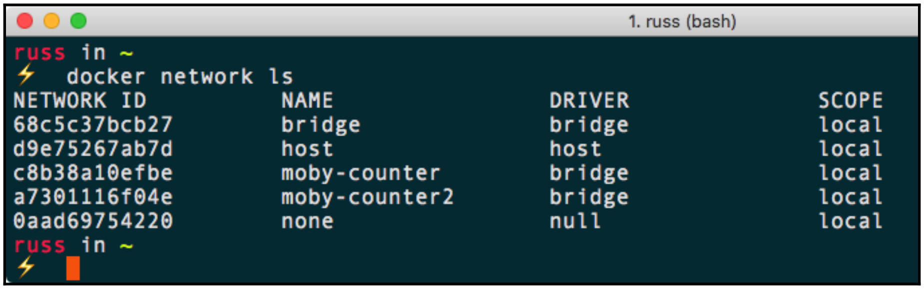
```
$ docker container run -d --name redis2 --network moby-counter2 --network-
alias redis redis:alpine
```

As you can see, we have named the container redis2, but set the --network-alias to be redis; this means that when we perform the lookup, we see the correct IP address returned:

```
$ docker container exec moby-counter2 nslookup redis 127.0.0.1
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:



1. russ (bash)

```
russ in ~
⚡  docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
68c5c37bcb27        bridge              bridge              local
d9e75267ab7d        host                host                local
c8b38a10efbe        moby-counter        bridge              local
a7301116f04e        moby-counter2       bridge              local
0aad69754220        none                null                local
russ in ~
⚡
```

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:

```
[
    {
        "Name": "moby-counter",
        "Id":
"c8b38a10efbefd701c83203489459d9d5a1c78a79fa055c1c81c18dea3f1883c",
        "Created": "2018-08-26T11:51:09.7958001Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
"4e7931312ed299ed9132f3553e0518db79b4c36c43d36e88306aed7f6f9749d8": {
                "Name": "moby-counter",
                "EndpointID":
"dc83770ae0939c98416ee69d939b30a1da391b11d14012c8188be287baa9c325",
                "MacAddress": "02:42:ac:12:00:03",
                "IPv4Address": "172.18.0.3/16",
                "IPv6Address": ""
            },
"d760bc59c3ac5f9ba8b7aa8e9f61fd21ce0b8982f3a85db888a5bcf103bedf6e": {
                "Name": "redis",
                "EndpointID":
"5af2bfd1ce486e38a9c5cddf9e16878fdb91389cc122cfef62d5e575a91b89b9",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

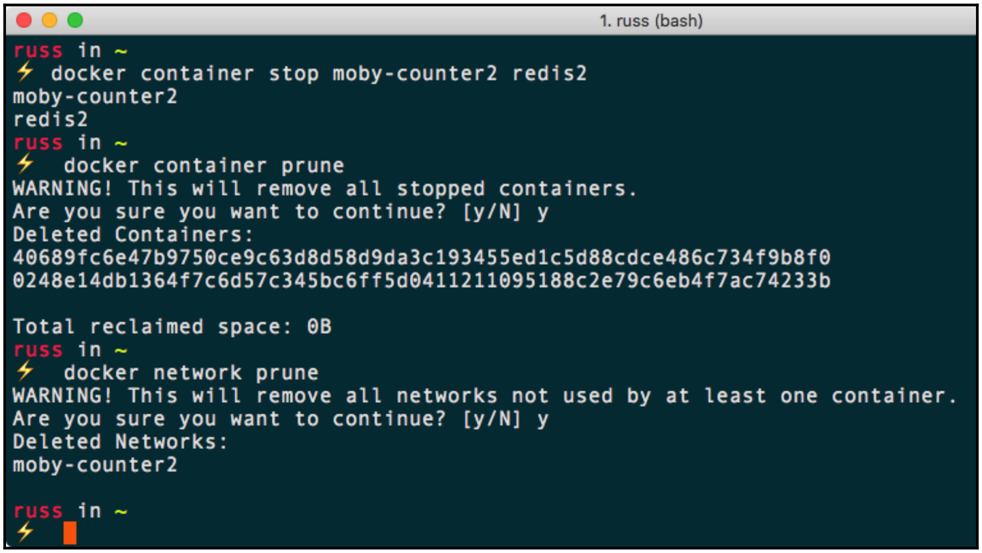
As you can see, it contains information on the network addressing being used in the IPAM section, along with details on each of the two containers running in the network.

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
```

This will remove the containers and network, as shown in the following screenshot:

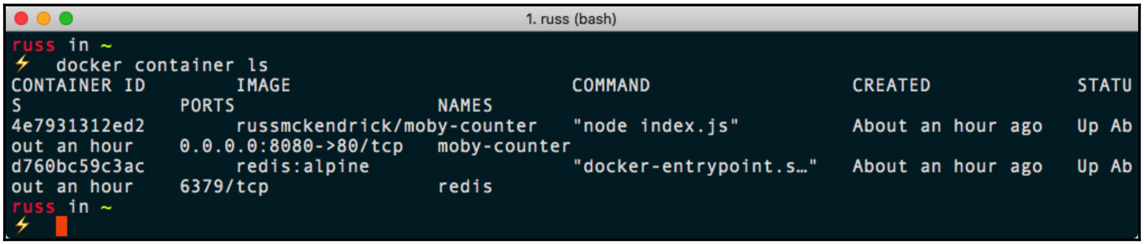


```
russ in ~
⚡ docker container stop moby-counter2 redis2
moby-counter2
redis2
russ in ~
⚡  docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
40689fc6e47b9750ce9c63d8d58d9da3c193455ed1c5d88cdce486c734f9b8f0
0248e14db1364f7c6d57c345bc6ff5d0411211095188c2e79c6eb4f7ac74233b

Total reclaimed space: 0B
russ in ~
⚡  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

russ in ~
⚡
```

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.

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:

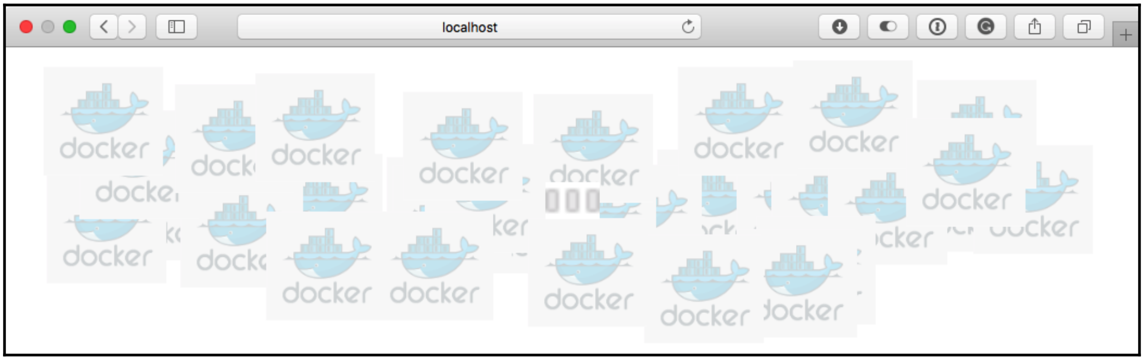


```
russ in ~
⚡  docker container ls
CONTAINER ID        IMAGE                         COMMAND                  CREATED             STATU
S              PORTS                  NAMES
4e7931312ed2        russmckendrick/moby-counter   "node index.js"          About an hour ago   Up Ab
out an hour    0.0.0.0:8080->80/tcp   moby-counter
d760bc59c3ac        redis:alpine                  "docker-entrypoint.s…"   About an hour ago   Up Ab
out an hour    6379/tcp               redis
russ in ~
⚡
```

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
```

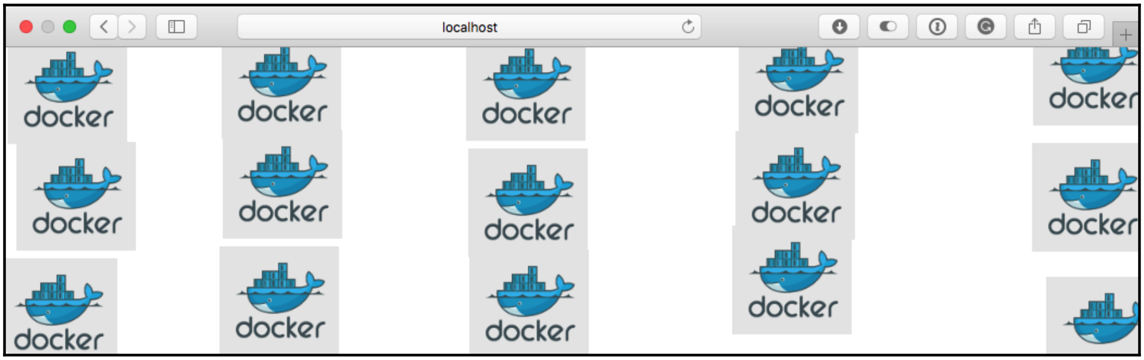
If you have your browser open, you may notice that the Docker icons have faded into the background and there is an animated loader in the center of the screen. This is basically to show that the application is waiting for the connection to the Redis container to be re-established:



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:



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:

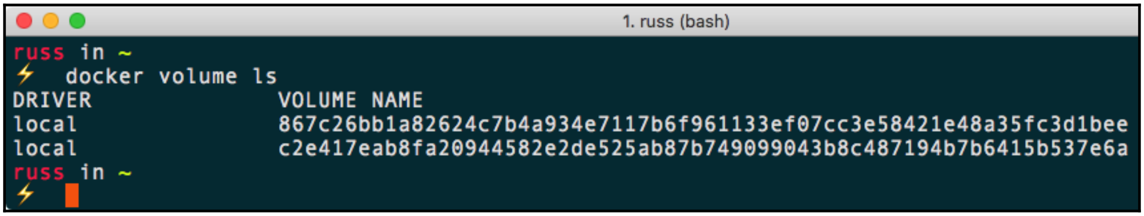
```
FROM alpine:3.8

RUN addgroup -S redis && adduser -S -G redis redis
RUN apk add --no-cache 'su-exec>=0.2'

ENV REDIS_VERSION 4.0.11
ENV REDIS_DOWNLOAD_URL
http://download.redis.io/releases/redis-4.0.11.tar.gz
ENV REDIS_DOWNLOAD_SHA
fc53e73ae7586bcdacb4b63875d1ff04f68c5474c1ddeda78f00e5ae2eed1bbb

RUN set -ex; \
  \
  apk add --no-cache --virtual .build-deps \
    coreutils \
    gcc \
    jemalloc-dev \
    linux-headers \
    make \
    musl-dev \
  ; \
  \
  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; \
  \
  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$'
/usr/src/redis/src/server.h; \
  \
  make -C /usr/src/redis -j "$(nproc)"; \
  make -C /usr/src/redis install; \
  \
  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 --virtual .redis-rundeps $runDeps; \
  apk del .build-deps; \
  \
  redis-server --version

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, toward the end 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 ran `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:



```
russ in ~
⚡  docker volume ls
DRIVER              VOLUME NAME
local               867c26bb1a82624c7b4a934e7117b6f961133ef07cc3e58421e48a35fc3d1bee
local               c2e417eab8fa20944582e2de525ab87b749099043b8c487194b7b6415b537e6a
russ in ~
⚡
```

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
c2e417eab8fa20944582e2de525ab87b749099043b8c487194b7b6415b537e6a:/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 12
drwxr-xr-x 1 root root 4.0K Aug 26 13:30 ..
drwxr-xr-x 2 redis redis 4.0K Aug 26 12:44 .
-rw-r--r-- 1 redis redis 392 Aug 26 12:44 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.

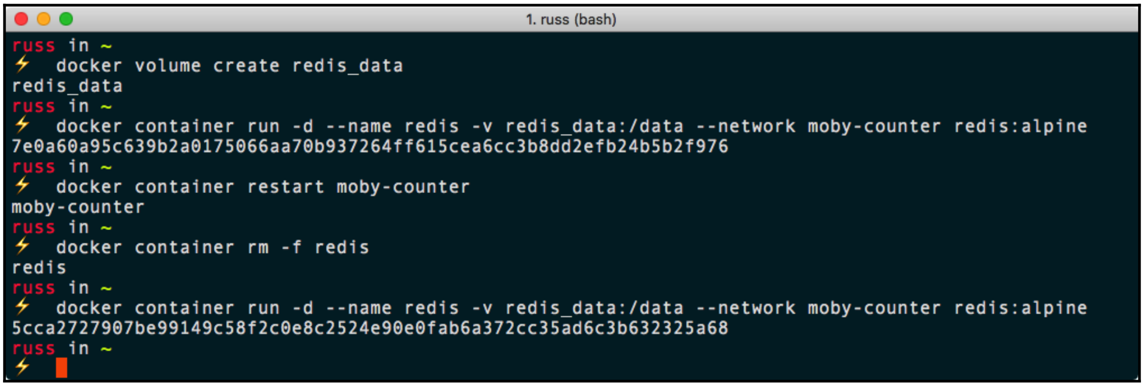
Finally, you can override the volume with your own. To create a volume, we need to use the `volume` command:

```
$ 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 screen below shows the volume being created, attached to a container which is then removed and finally reattached to a new container:



```
russ in ~
⚡  docker volume create redis_data
redis_data
russ in ~
⚡  docker container run -d --name redis -v redis_data:/data --network moby-counter redis:alpine
7e0a60a95c639b2a0175066aa70b937264ff615cea6cc3b8dd2efb24b5b2f976
russ in ~
⚡  docker container restart moby-counter
moby-counter
russ in ~
⚡  docker container rm -f redis
redis
russ in ~
⚡  docker container run -d --name redis -v redis_data:/data --network moby-counter redis:alpine
5cca2727907be99149c58f2c0e8c2524e90e0fab6a372cc35ad6c3b632325a68
russ in ~
⚡  
```

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 code will produce something like the following output:

```
[
    {
        "CreatedAt": "2018-08-26T13:39:33Z",
        "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.

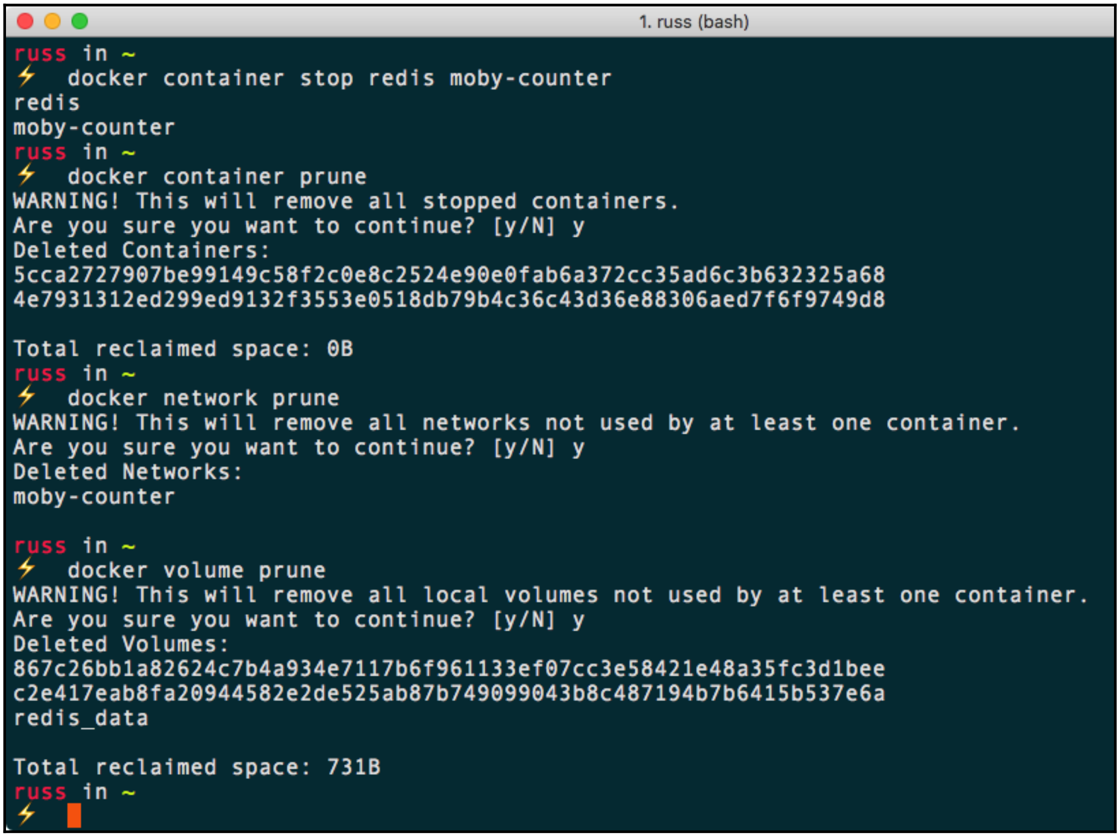
Don't worry though; we will be looking at Docker volumes and how you can interact with data in later chapters. For now, we should tidy up. First of all, remove the two containers and network:

```
$ 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:



```
russ in ~
⚡  docker container stop redis moby-counter
redis
moby-counter
russ in ~
⚡  docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
5cca2727907be99149c58f2c0e8c2524e90e0fab6a372cc35ad6c3b632325a68
4e7931312ed299ed9132f3553e0518db79b4c36c43d36e88306aed7f6f9749d8

Total reclaimed space: 0B
russ in ~
⚡  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

russ in ~
⚡  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:
867c26bb1a82624c7b4a934e7117b6f961133ef07cc3e58421e48a35fc3d1bee
c2e417eab8fa20944582e2de525ab87b749099043b8c487194b7b6415b537e6a
redis_data

Total reclaimed space: 731B
russ in ~
⚡
```

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 using Docker Compose.

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. We will break the chapter down into the following sections:

- Docker Compose introduction
- Our first Docker Compose application
- Docker Compose YAML files
- Docker Compose commands
- Docker App

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.

A full copy of the code used in this chapter can be found at: `https://github.com/PacktPublishing/Mastering-Docker-Third-Edition/tree/master/chapter05`.

Check out the following video to see the Code in Action:

`http://bit.ly/2q7MJZU`

Introducing Docker Compose

In Chapter 1, *Docker Overview*, we discussed a few of the problems that Docker has been designed to solve. We explained how it addresses challenges such as running two applications side by side by isolating processes into a single container, meaning that you can run two entirely different versions of the same software stack, say PHP 5.6 and PHP 7, on the same host, as we did in `Chapter 2`, *Building Container Images*.

Towards 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 location of the Docker logos on screen.

This meant that we had to launch two containers, one for the 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
```

Okay, I could get away with losing the first two commands as the image will be pulled during the run if they haven't already pulled it, but as the application starts 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 is DevOps 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, they saw this as a scenario that the technology is meant to stop us from finding ourselves in. Thanks to Docker, people no longer have to worry about inconsistencies in the environment they are launching their applications in as they can now be shipped in images.

For this reason, back in July 2014, Docker purchased a small British start-up who offered two container-based products called Orchard Laboratories.

The first of the two products was a Docker-based hosting platform: think of it as a hybrid of Docker Machine, which we will be looking at in a later chapter, and Docker itself. 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 would have launched a Docker host on Orchard's platform and then a Redis container.

The second product was an open source project called **Fig**. Fig lets you use a `YAML` file to define how you would like your multi-container application to be structured. It would then take the `YAML` file and automate the launch of 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.

Of these two products, Docker purchased Orchard Laboratories for Fig. After a short while, the Orchard service was discontinued, and, in February 2015, Fig became Docker Compose.

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 and bring up the two-container application we launched manually at the end of the last chapter using just Docker Compose.

Our first Docker Compose application

As already mentioned, Docker Compose uses a YAML file, typically named `dockercompose.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:

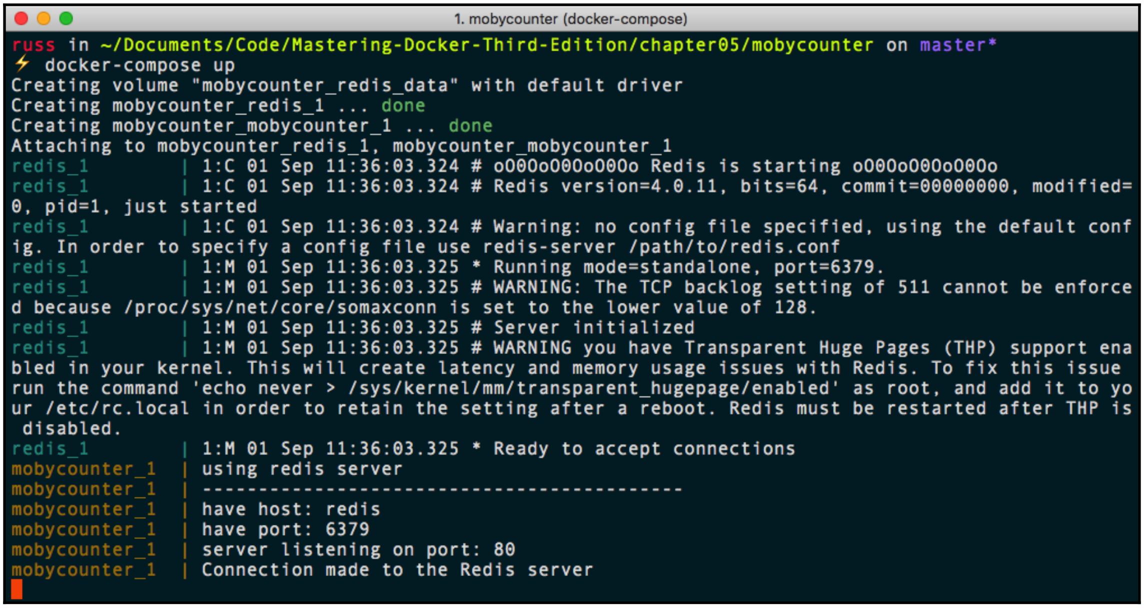
```
version: "3"

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. 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, a lot happened when it launched:



```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/mobycounter on master*
⚡ docker-compose up
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 01 Sep 11:36:03.324 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
redis_1          | 1:C 01 Sep 11:36:03.324 # Redis version=4.0.11, bits=64, commit=00000000, modified=
0, pid=1, just started
redis_1          | 1:C 01 Sep 11:36:03.324 # Warning: no config file specified, using the default conf
ig. In order to specify a config file use redis-server /path/to/redis.conf
redis_1          | 1:M 01 Sep 11:36:03.325 * Running mode=standalone, port=6379.
redis_1          | 1:M 01 Sep 11:36:03.325 # WARNING: The TCP backlog setting of 511 cannot be enforce
d because /proc/sys/net/core/somaxconn is set to the lower value of 128.
redis_1          | 1:M 01 Sep 11:36:03.325 # Server initialized
redis_1          | 1:M 01 Sep 11:36:03.325 # WARNING you have Transparent Huge Pages (THP) support ena
bled 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 yo
ur /etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after THP is
 disabled.
redis_1          | 1:M 01 Sep 11:36:03.325 * 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
```

As you can see, from the first few lines, Docker Compose did the following:

- It created a volume called mobycounter_redis_data, using the default driver as 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. More on this in a minute.
- It launched two containers, one called mobycounter_redis_1 , and the second called mobycounter_mobycounter_1.

You may have also spotted 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 the heart of Docker Compose.

YAML is a recursive acronym that stands for **YAML Ain't Markup Language**. It is used by a lot of different applications for both configuration and also 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.

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:

```
version: "3"
```

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
--> container name:
----> container options
```

In our example, we defined two containers. I have separated them 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
```

The syntax for defining the service is 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 being 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 this:

- `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:

```
volume:
 redis_data:
```

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.

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. Ignore the other files, for now; 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:

```
version: "3"

services:
```

As you can see, it starts simply enough by defining the version and then it starts to list the services. 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, we are actually building an image from scratch by using `build` instead of the `image` command:

```
  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` when it 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 the second called `back-tier`.

The `front-tier` network will have the containers that have to have ports 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. Like the vote container, the image is built locally using a Dockerfile that can be found in the ./result folder:

```
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:

```
worker:
  build:
    context: ./worker
  depends_on:
    - "redis"
  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 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`:

```
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:

```
volumes:
  db-data:
```

Then, finally, here are the two networks we have been speaking about:

```
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 following command `echo "104.16.16.35 registry.npmjs.org" >> /etc/hosts` as a user with privileges to write to `/etc/hosts`.

We start by creating the networks and getting the volume ready for our containers to use:

```
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
```

We then build the `vote` container image:

```
Building vote
Step 1/7 : FROM python:2.7-alpine
2.7-alpine: Pulling from library/python
8e3ba11ec2a2: Pull complete
ea489525e565: Pull complete
f0d8a8560df7: Pull complete
8971431029b9: Pull complete
Digest:
sha256:c9f17d63ea49a186d899cb9856a5cc1c601783f2c9fa9b776b4582a49ceac548
Status: Downloaded newer image for python:2.7-alpine
 ---> 5082b69714da
Step 2/7 : WORKDIR /app
 ---> Running in 663db929990a
Removing intermediate container 663db929990a
 ---> 45fe48ea8e4c
Step 3/7 : ADD requirements.txt /app/requirements.txt
 ---> 2df3b3211688
Step 4/7 : RUN pip install -r requirements.txt
 ---> Running in 23ad90b81e6b
[lots of python build output here]
Step 5/7 : ADD . /app
 ---> cebab4f80850
Step 6/7 : EXPOSE 80
 ---> Running in b28d426e3516
Removing intermediate container b28d426e3516
 ---> bb951ea7dffc
```

```
Step 7/7 : CMD ["gunicorn", "app:app", "-b", "0.0.0.0:80", "--log-file", "-
", "--access-logfile", "-", "--workers", "4", "--keep-alive", "0"]
 ---> Running in 2e97ca847f8a
Removing intermediate container 2e97ca847f8a
 ---> 638c74fab05e
Successfully built 638c74fab05e
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 this `vote` image has been built, the `worker` image is constructed:

```
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:f4ea9cdf980bb9512523a3fb88e30f2b83cce4b0cddd2972bc36685461081e2f
Status: Downloaded newer image for microsoft/dotnet:2.0.0-sdk
 ---> fde8197d13f4
Step 2/5 : WORKDIR /code
 ---> Running in 1ca2374cff99
Removing intermediate container 1ca2374cff99
 ---> 37f9b05325f9
Step 3/5 : ADD src/Worker /code/src/Worker
 ---> 9d393c6bd48c
Step 4/5 : RUN dotnet restore -v minimal src/Worker && dotnet publish -c
Release -o "./" "src/Worker/"
 ---> Running in ab9fe7820062
  Restoring packages for /code/src/Worker/Worker.csproj...
  [lots of .net build output here]
  Restore completed in 8.86 sec for /code/src/Worker/Worker.csproj.
Microsoft (R) Build Engine version 15.3.409.57025 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.
  Worker -> /code/src/Worker/bin/Release/netcoreapp2.0/Worker.dll
  Worker -> /code/src/Worker/
Removing intermediate container ab9fe7820062
 ---> cf369fbb11dd
Step 5/5 : CMD dotnet src/Worker/Worker.dll
 ---> Running in 232416405e3a
Removing intermediate container 232416405e3a
```

```
 ---> d355a73a45c9
Successfully built d355a73a45c9
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`.
```

Then the `redis` image is pulled:

```
Pulling redis (redis:alpine)...
alpine: Pulling from library/redis
8e3ba11ec2a2: Already exists
1f20bd2a5c23: Pull complete
782ff7702b5c: Pull complete
82d1d664c6a7: Pull complete
69f8979cc310: Pull complete
3ff30b3bc148: Pull complete
Digest:
sha256:43e4d14fcffa05a5967c353dd7061564f130d6021725dd219f0c6fcbcc6b5076
Status: Downloaded newer image for redis:alpine
```

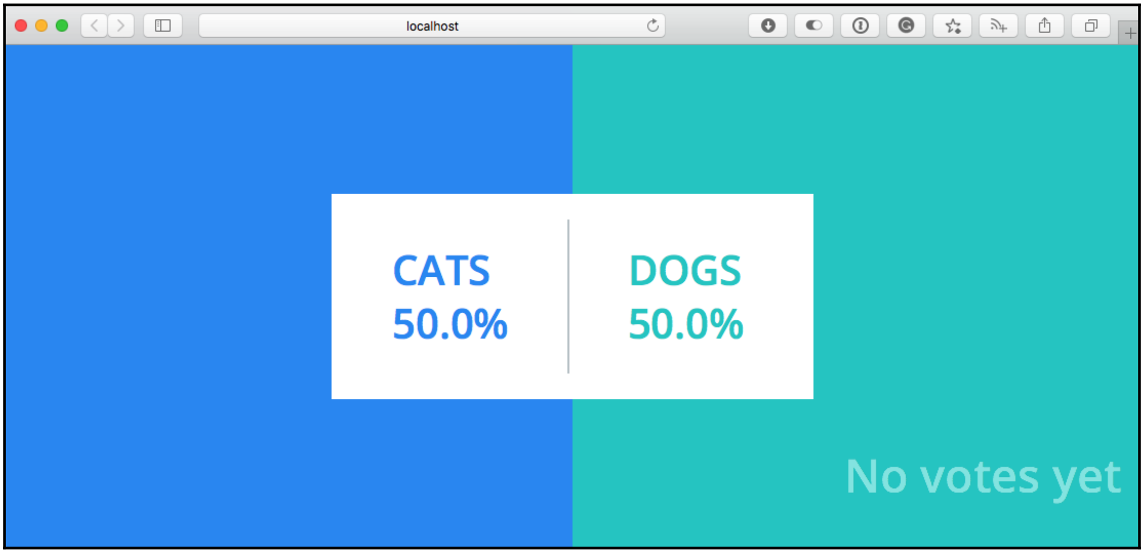
This is followed by the PostgreSQL image for the `db` container:

```
Pulling db (postgres:9.4)...
9.4: Pulling from library/postgres
be8881be8156: Pull complete
01d7a10e8228: Pull complete
f8968e0fd5ca: Pull complete
69add08e7e51: Pull complete
954fe1f9e4e8: Pull complete
9ace39987bb3: Pull complete
9020931bcc5d: Pull complete
71f421dd7dcd: Pull complete
a909f41228ab: Pull complete
cb62befcd007: Pull complete
4fea257fde1a: Pull complete
f00651fb0fbf: Pull complete
0ace3ceac779: Pull complete
b64ee32577de: Pull complete
Digest:
sha256:7430585790921d82a56c4cbe62fdf50f03e00b89d39cbf881afa1ef82eefd61c
Status: Downloaded newer image for postgres:9.4
```

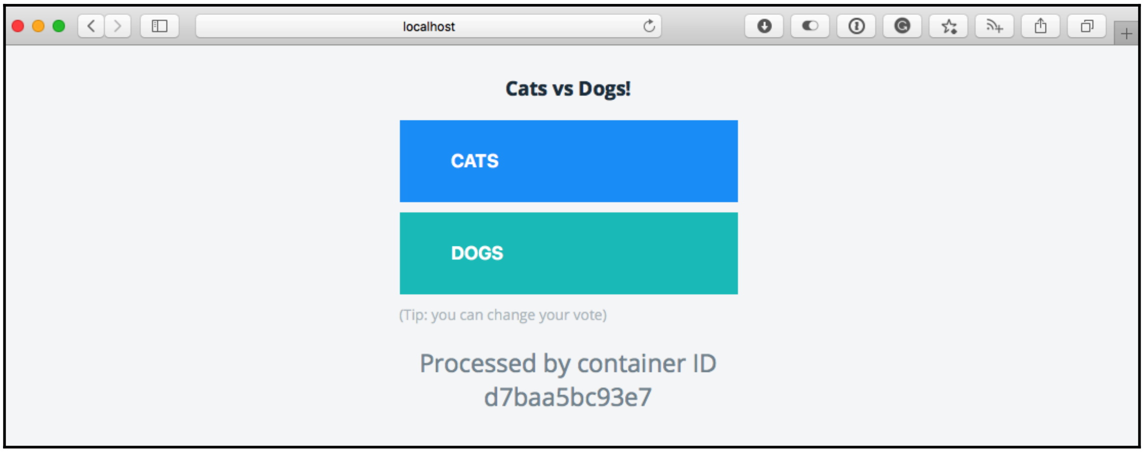
Now it is time for the big one; the building of the `result` image. Node.js is quite verbose, so you will get quite a bit of output being printed to the screen as the `npm` sections of the `Dockerfile` are executed; in fact, there are over 250 lines of output:

```
Building result
Step 1/11 : FROM node:8.9-alpine
8.9-alpine: Pulling from library/node
605ce1bd3f31: Pull complete
79b85b1676b5: Pull complete
20865485d0c2: Pull complete
Digest:
sha256:6bb963d58da845cf66a22bc5a48bb8c686f91d30240f0798feb0d61a2832fc46
Status: Downloaded newer image for node:8.9-alpine
 ---> 406f227b21f5
Step 2/11 : RUN mkdir -p /app
 ---> Running in 4af9c85c67ee
Removing intermediate container 4af9c85c67ee
 ---> f722dde47fcf
Step 3/11 : WORKDIR /app
 ---> Running in 8ad29a42f32f
Removing intermediate container 8ad29a42f32f
 ---> 32a05580f2ec
Step 4/11 : RUN npm install -g nodemon
[lots and lots of nodejs output]
Step 8/11 : COPY . /app
 ---> 725966c2314f
Step 9/11 : ENV PORT 80
 ---> Running in 6f402a073bf4
Removing intermediate container 6f402a073bf4
 ---> e3c426b5a6c8
Step 10/11 : EXPOSE 80
 ---> Running in 13db57b3c5ca
Removing intermediate container 13db57b3c5ca
 ---> 1305ea7102cf
Step 11/11 : CMD ["node", "server.js"]
 ---> Running in a27700087403
Removing intermediate container a27700087403
 ---> 679c16721a7f
Successfully built 679c16721a7f
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`.
```

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:



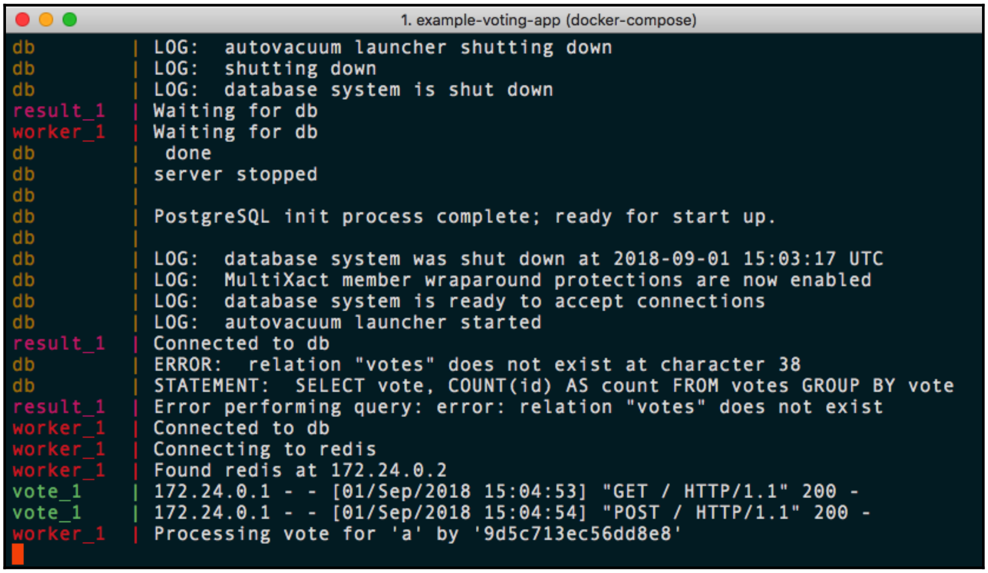
The `vote` part of the application can be found at `http://localhost:5000`:



(Tip: you can change your vote)

Processed by container ID d7baa5bc93e7

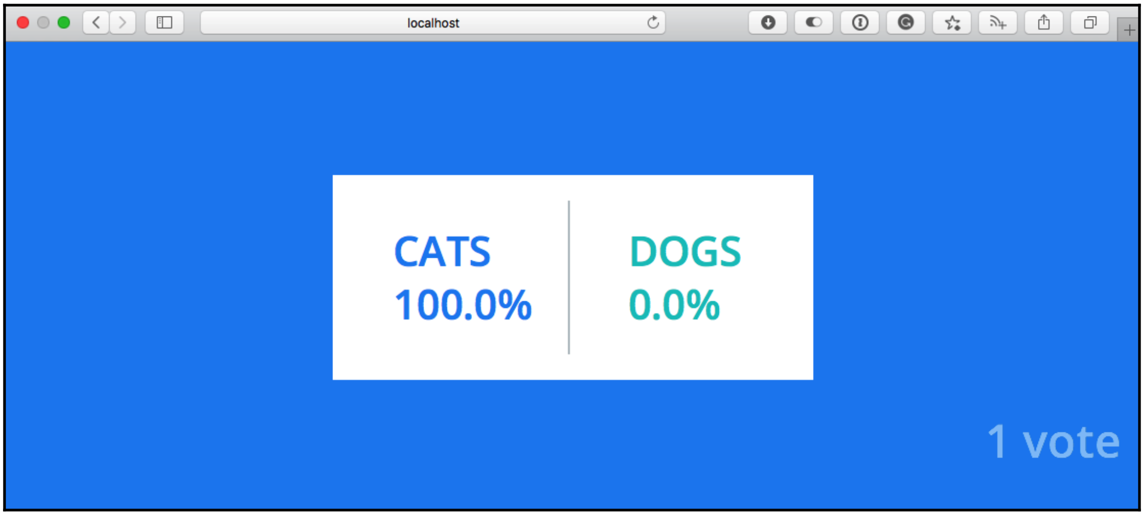
Clicking on either **CATS** or **DOGS** will register a vote; you should be able to see this logged in the Docker Compose output in your Terminal:



1. example-voting-app (docker-compose)

```
db          | LOG:  autovacuum launcher shutting down
db          | LOG:  shutting down
db          | LOG:  database system is shut down
result_1    | Waiting for db
worker_1    | Waiting for db
db          |  done
db          | server stopped
db          |
db          | PostgreSQL init process complete; ready for start up.
db          |
db          | LOG:  database system was shut down at 2018-09-01 15:03:17 UTC
db          | LOG:  MultiXact member wraparound protections are now enabled
db          | LOG:  database system is ready to accept connections
db          | LOG:  autovacuum launcher started
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    | Connecting to redis
worker_1    | Found redis at 172.24.0.2
vote_1      | 172.24.0.1 - - [01/Sep/2018 15:04:53] "GET / HTTP/1.1" 200 -
vote_1      | 172.24.0.1 - - [01/Sep/2018 15:04:54] "POST / HTTP/1.1" 200 -
worker_1    | Processing vote for 'a' by '9d5c713ec56dd8e8'
```

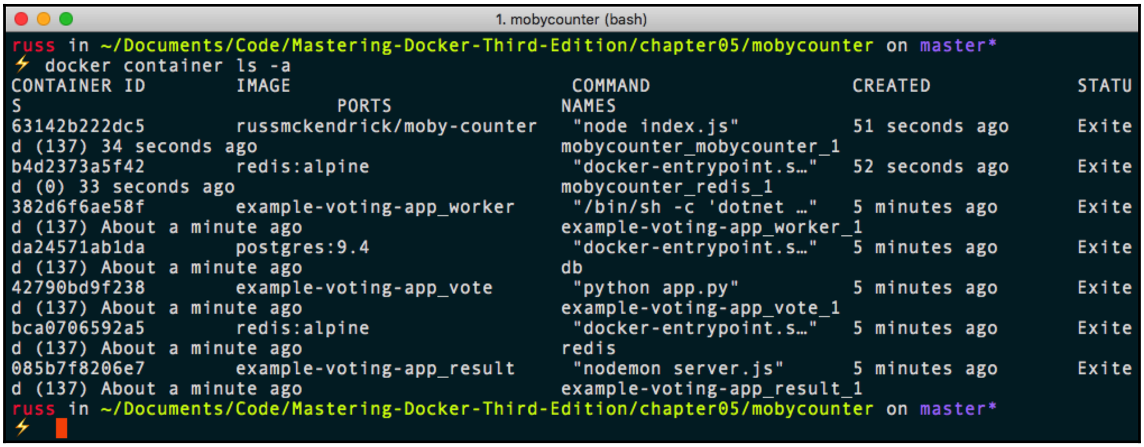
There are a few errors, as the Redis table structure is only created when the vote application registers the first vote; once a vote has been cast, the Redis table structure will be created and the worker container will take that vote and process it by writing to the `db` container. Once the vote has been cast, the `result` container will update in real time:



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 Kubenetes clusters. For now, let's get back to Docker Compose and look at some of the commands we can run.

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 in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/mobycounter on master*
⚡ docker container ls -a
CONTAINER ID        IMAGE                         COMMAND                  CREATED             STATU
S                            PORTS               NAMES
63142b222dc5        russmckendrick/moby-counter   "node index.js"          51 seconds ago      Exite
d (137) 34 seconds ago                           mobycounter_mobycounter_1
b4d2373a5f42        redis:alpine                  "docker-entrypoint.s…"   52 seconds ago      Exite
d (0) 33 seconds ago                             mobycounter_redis_1
382d6f6ae58f        example-voting-app_worker     "/bin/sh -c 'dotnet …"   5 minutes ago       Exite
d (137) About a minute ago                       example-voting-app_worker_1
da24571ab1da        postgres:9.4                  "docker-entrypoint.s…"   5 minutes ago       Exite
d (137) About a minute ago                       db
42790bd9f238        example-voting-app_vote       "python app.py"          5 minutes ago       Exite
d (137) About a minute ago                       example-voting-app_vote_1
bca0706592a5        redis:alpine                  "docker-entrypoint.s…"   5 minutes ago       Exite
d (137) About a minute ago                       redis
085b7f8206e7        example-voting-app_result     "nodemon server.js"      5 minutes ago       Exite
d (137) About a minute ago                       example-voting-app_result_1
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/mobycounter on master*
⚡
```

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.

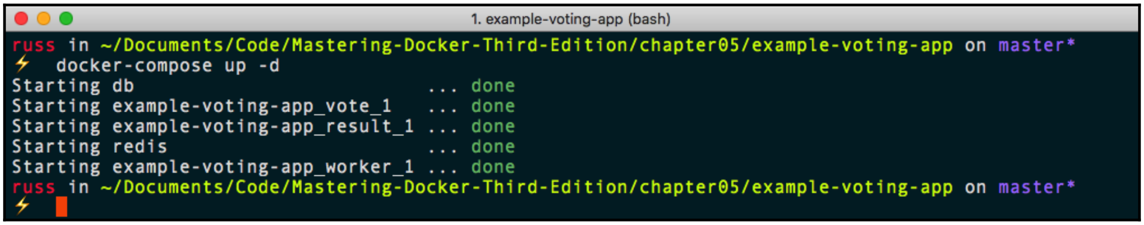
Choose one of the Docker Compose applications and change to the folder that contains the `docker-compose.yml` file, and we will work through some more Docker Compose commands. I will be using the **Example Vote** application.

Up and PS

The first one 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:

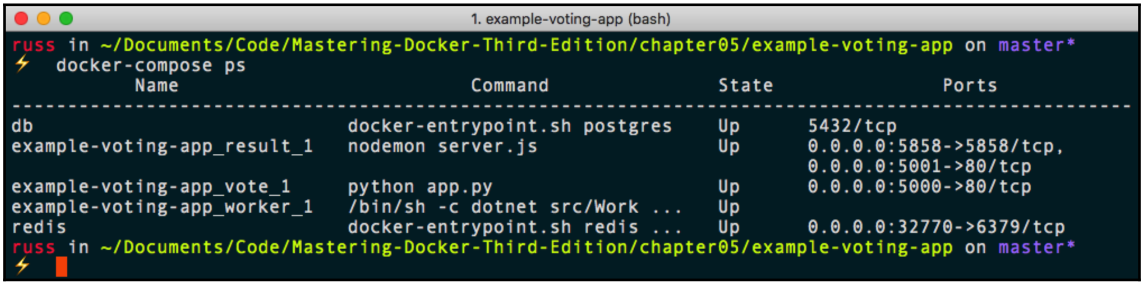


```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/example-voting-app on master*
⚡  docker-compose up -d
Starting db                           ... done
Starting example-voting-app_vote_1    ... done
Starting example-voting-app_result_1 ... done
Starting redis                        ... done
Starting example-voting-app_worker_1 ... done
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/example-voting-app on master*
⚡
```

Once control of your Terminal is returned, 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 in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/example-voting-app on master*
⚡  docker-compose ps
           Name                          Command               State                Ports
------------------------------------------------------------------------------------------------------
db                            docker-entrypoint.sh postgres    Up      5432/tcp
example-voting-app_result_1   nodemon server.js                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      0.0.0.0:32770->6379/tcp
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/example-voting-app on master*
⚡
```

When running these commands, Docker Compose will only be aware of the containers defined in the service 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 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` command 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. In the same way that the `docker container create` command does, they will have an exited state until you start them. The `create` command has a few useful flags you can pass:

- `--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 exactly in 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
```

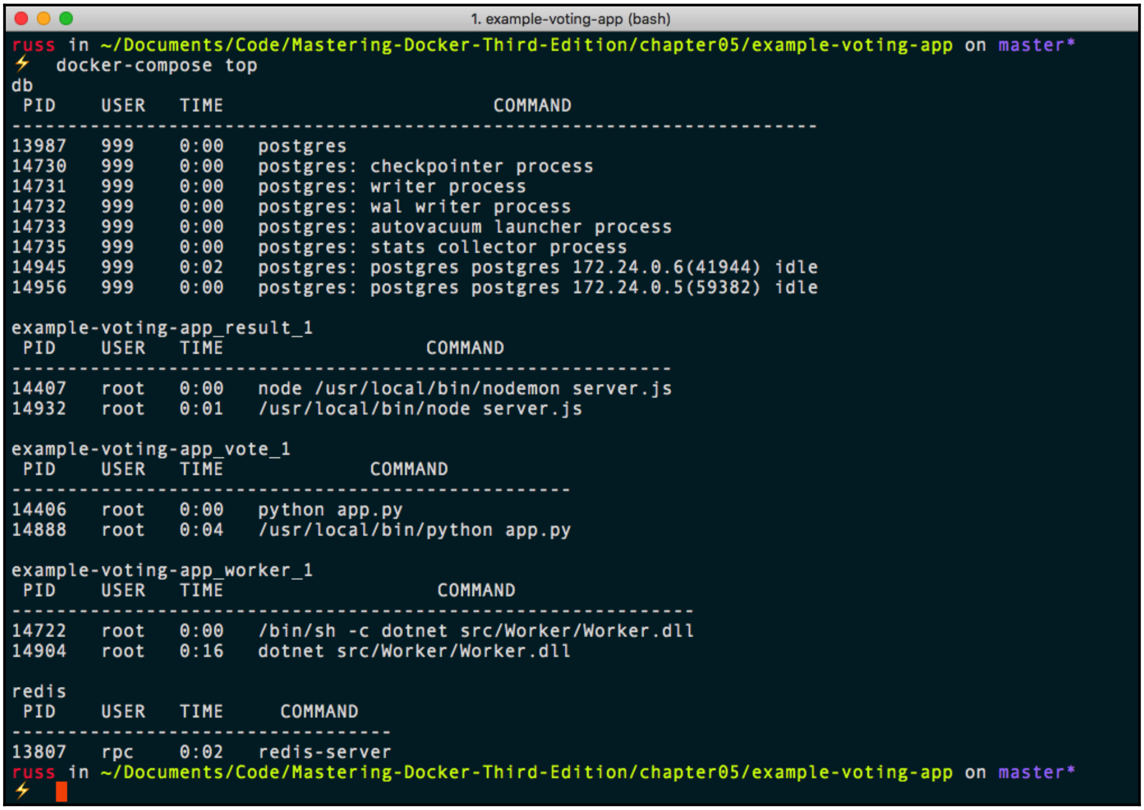
Top, logs, and events

The next three commands all give us feedback on what is happening within our running containers and Docker Compose.

The following command, like 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 in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/example-voting-app on master*
⚡  docker-compose top
db
 PID    USER   TIME                        COMMAND
------------------------------------------------------------------------
13987   999    0:00   postgres
14730   999    0:00   postgres: checkpointer process
14731   999    0:00   postgres: writer process
14732   999    0:00   postgres: wal writer process
14733   999    0:00   postgres: autovacuum launcher process
14735   999    0:00   postgres: stats collector process
14945   999    0:02   postgres: postgres postgres 172.24.0.6(41944) idle
14956   999    0:00   postgres: postgres postgres 172.24.0.5(59382) idle

example-voting-app_result_1
 PID    USER   TIME                  COMMAND
-----------------------------------------------------------
14407   root   0:00   node /usr/local/bin/nodemon server.js
14932   root   0:01   /usr/local/bin/node server.js

example-voting-app_vote_1
 PID    USER   TIME             COMMAND
--------------------------------------------------
14406   root   0:00   python app.py
14888   root   0:04   /usr/local/bin/python app.py

example-voting-app_worker_1
 PID    USER   TIME                   COMMAND
-------------------------------------------------------------
14722   root   0:00   /bin/sh -c dotnet src/Worker/Worker.dll
14904   root   0:16   dotnet src/Worker/Worker.dll

redis
 PID    USER   TIME     COMMAND
----------------------------------
13807   rpc    0:02   redis-server
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/example-voting-app on master*
⚡
```

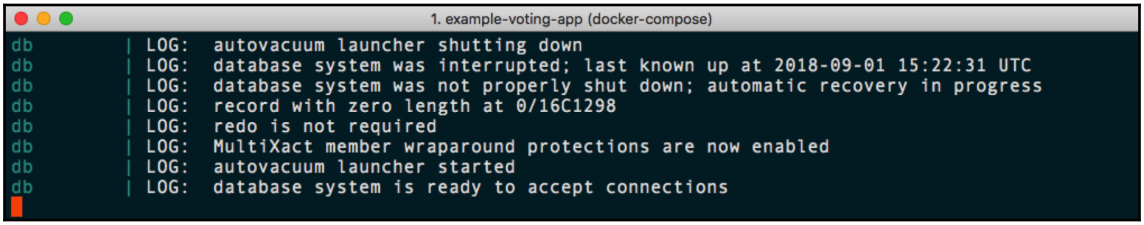
If you would like to see just one of the services, you simply have to pass its name when running the command:

```
$ docker-compose top db
```

The next command streams the `logs` from each of the running containers to screen:

```
$ docker-compose logs
```

Like 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:

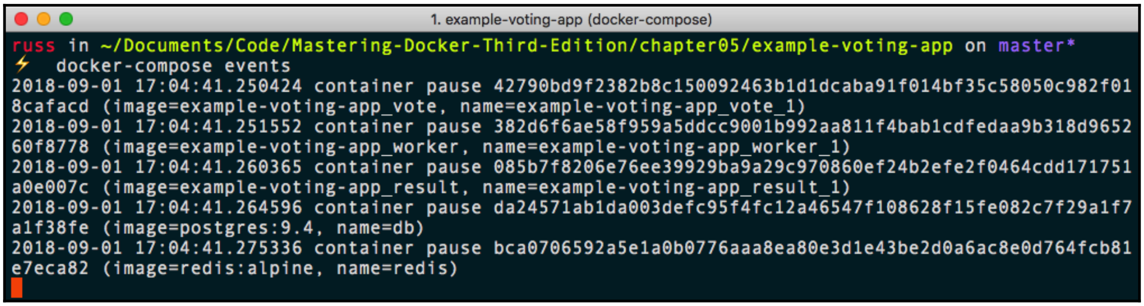


```
db          | LOG:  autovacuum launcher shutting down
db          | LOG:  database system was interrupted; last known up at 2018-09-01 15:22:31 UTC
db          | LOG:  database system was not properly shut down; automatic recovery in progress
db          | LOG:  record with zero length at 0/16C1298
db          | LOG:  redo is not required
db          | LOG:  MultiXact member wraparound protections are now enabled
db          | LOG:  autovacuum launcher started
db          | LOG:  database system is ready to accept connections
```

The `events` command again works like the docker container version; 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:

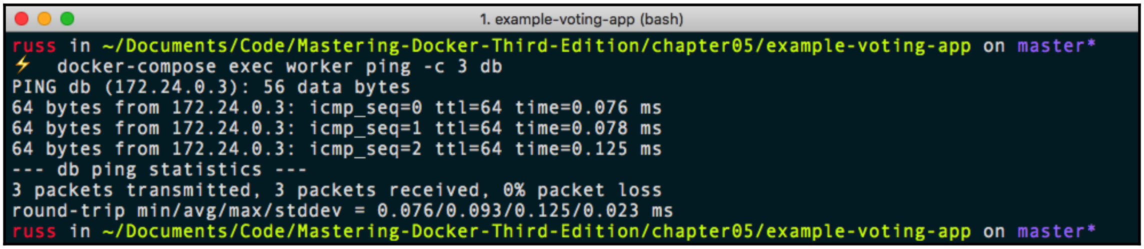


```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/example-voting-app on master*
⚡  docker-compose events
2018-09-01 17:04:41.250424 container pause 42790bd9f2382b8c150092463b1d1dcaba91f014bf35c58050c982f01
8cafacd (image=example-voting-app_vote, name=example-voting-app_vote_1)
2018-09-01 17:04:41.251552 container pause 382d6f6ae58f959a5ddcc9001b992aa811f4bab1cdfedaa9b318d9652
60f8778 (image=example-voting-app_worker, name=example-voting-app_worker_1)
2018-09-01 17:04:41.260365 container pause 085b7f8206e76ee39929ba9a29c970860ef24b2efe2f0464cdd171751
a0e007c (image=example-voting-app_result, name=example-voting-app_result_1)
2018-09-01 17:04:41.264596 container pause da24571ab1da003defc95f4fc12a46547f108628f15fe082c7f29a1f7
a1f38fe (image=postgres:9.4, name=db)
2018-09-01 17:04:41.275336 container pause bca0706592a5e1a0b0776aaa8ea80e3d1e43be2d0a6ac8e0d764fcb81
e7eca82 (image=redis:alpine, name=redis)
```

These two commands run similar 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 in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/example-voting-app on master*
⚡  docker-compose exec worker ping -c 3 db
PING db (172.24.0.3): 56 data bytes
64 bytes from 172.24.0.3: icmp_seq=0 ttl=64 time=0.076 ms
64 bytes from 172.24.0.3: icmp_seq=1 ttl=64 time=0.078 ms
64 bytes from 172.24.0.3: icmp_seq=2 ttl=64 time=0.125 ms
--- db ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.076/0.093/0.125/0.023 ms
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/example-voting-app on master*
```

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` to `/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:

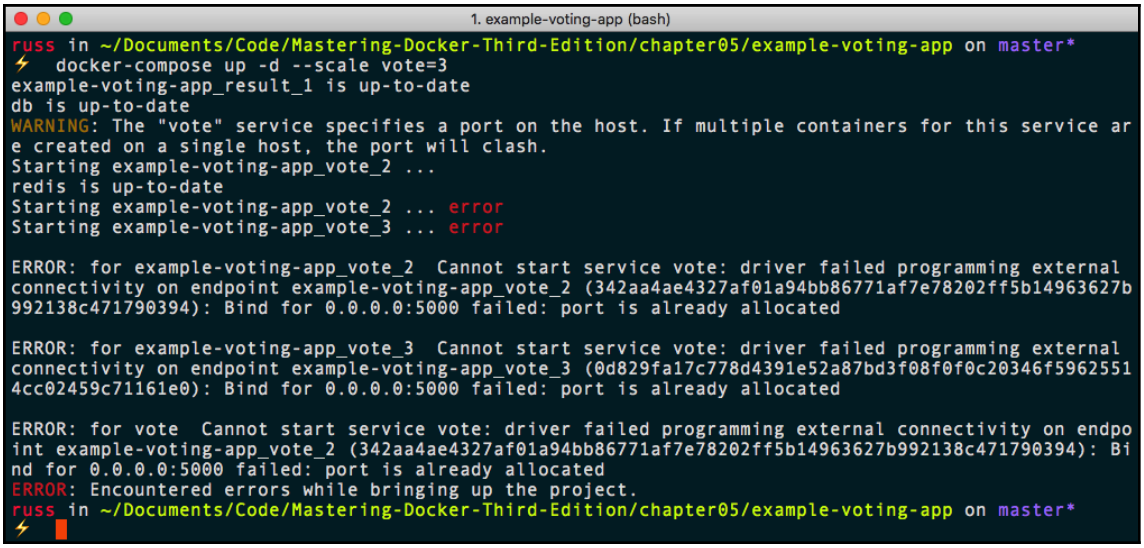
```
$ docker-compose up -d --scale worker=3
```

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 in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/example-voting-app on master*
⚡  docker-compose up -d --scale vote=3
example-voting-app_result_1 is up-to-date
db 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_2 ...
redis is up-to-date
Starting example-voting-app_vote_2 ... error
Starting 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 (342aa4ae4327af01a94bb86771af7e78202ff5b14963627b
992138c471790394): 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 (0d829fa17c778d4391e52a87bd3f08f0f0c20346f5962551
4cc02459c71161e0): 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 (342aa4ae4327af01a94bb86771af7e78202ff5b14963627b992138c471790394): Bi
nd for 0.0.0.0:5000 failed: port is already allocated
ERROR: Encountered errors while bringing up the project.
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/example-voting-app on master*
⚡
```

That is because we cannot have three individual containers all trying to map to the same port. There is a workaround for this and we will look at that in more detail in a later chapter.

Kill, rm, and down

The three Docker Compose commands we are finally going to look at 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:

```
$ 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`:

```
$ docker-compose rm
```

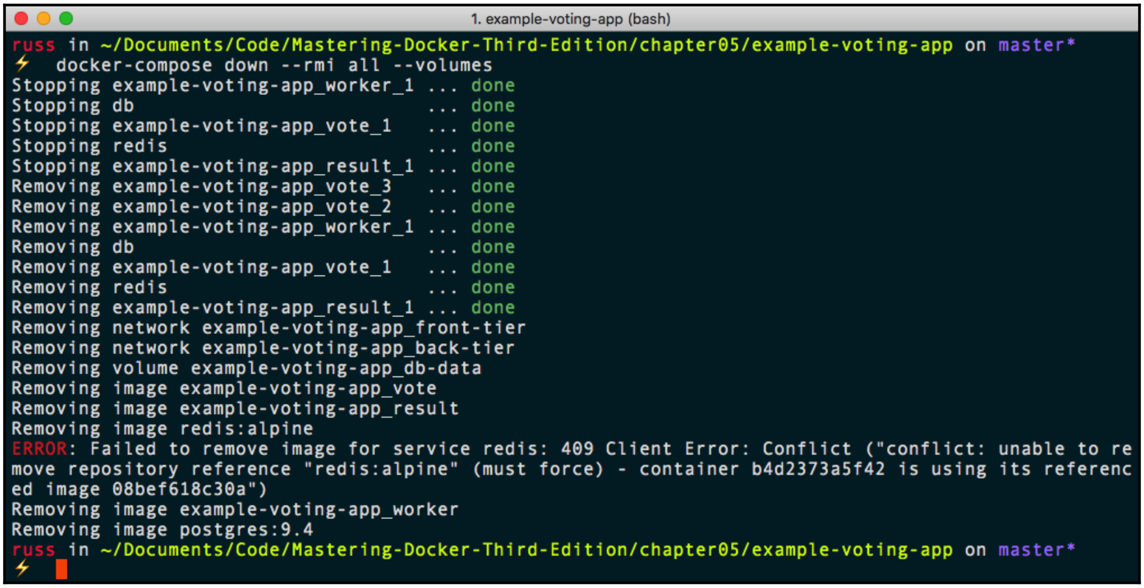
Finally, we have the `down` command. This, as you might have already guessed, has the opposite effect of running `docker-compose up`:

```
$ 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 they will not be removed:



```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/example-voting-app on master*
⚡  docker-compose down --rmi all --volumes
Stopping example-voting-app_worker_1 ... done
Stopping db                          ... done
Stopping example-voting-app_vote_1   ... done
Stopping redis                       ... done
Stopping example-voting-app_result_1 ... 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_vote_1   ... done
Removing redis                       ... done
Removing example-voting-app_result_1 ... 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 re
move repository reference "redis:alpine" (must force) - container b4d2373a5f42 is using its referenc
ed image 08bef618c30a")
Removing image example-voting-app_worker
Removing image postgres:9.4
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/example-voting-app on master*
⚡
```

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` , and the ones downloaded from Docker Hub.

Docker App

Before we start this section, I should issue the following warning:

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 this, I am only going to cover the installation of the macOS version. However, before we install it, lets 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 a self-contained binary that helps you to create an application bundle that can be shared via Docker Hub or a Docker Enterprise Registry.

I would recommend checking the GitHub projects **Releases** page (you can find the link in the *Further reading* section) to make sure you are a using the latest version. If the version is later than 0.4.1, you will need to replace the version number in the following command.

To install Docker App on macOS, you can run the following commands, starting with setting the version to download:

```
$ VERSION=v0.4.1
```

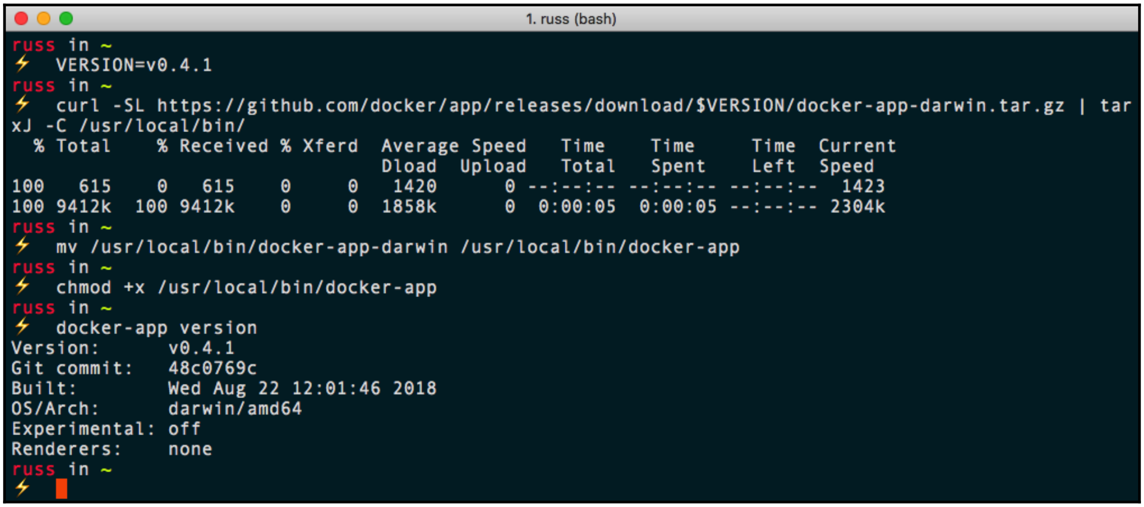
Now that you have the correct version, you can download it and put it in place using the following:

```
$ curl -SL
https://github.com/docker/app/releases/download/$VERSION/docker-app-darwin.
tar.gz | tar xJ -C /usr/local/bin/
$ mv /usr/local/bin/docker-app-darwin /usr/local/bin/docker-app
$ chmod +x /usr/local/bin/docker-app
```

Once in place, you should be able to run the following command that will print some basic information about binary on screen:

```
$ docker-app version
```

The full output of the preceding commands can be seen here for those not following along:



```
russ in ~
⚡  VERSION=v0.4.1
russ in ~
⚡  curl -SL https://github.com/docker/app/releases/download/$VERSION/docker-app-darwin.tar.gz | tar
xJ -C /usr/local/bin/
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   615    0   615    0     0   1420      0 --:--:-- --:--:-- --:--:--  1423
100 9412k  100 9412k    0     0  1858k      0  0:00:05  0:00:05 --:--:-- 2304k
russ in ~
⚡  mv /usr/local/bin/docker-app-darwin /usr/local/bin/docker-app
russ in ~
⚡  chmod +x /usr/local/bin/docker-app
russ in ~
⚡  docker-app version
Version:      v0.4.1
Git commit:   48c0769c
Built:        Wed Aug 22 12:01:46 2018
OS/Arch:      darwin/amd64
Experimental: off
Renderers:    none
russ in ~
⚡
```

There is a slight change to the `docker-compose.yml` file we will be using. The version needs to be updated to `3.6` rather than just `3`. Not doing this will result in the following error:

```
Error: unsupported Compose file version: 3
```

The command we need to run, and which also generates the preceding error, is as follows:

```
$ docker-app init --single-file mobycounter
```

This command takes our `docker-compose.yml` file and embeds it in a `.dockerapp` file. Initially, there will be quite a few comments in the file that detail what changes you need to make before moving on to the next steps. I have left an unaltered version of the file in the repository, in the `chapter5/mobycounter-app` folder called `mobycounter.dockerapp.original`.

An edited version of the `mobycounter.dockerapp` file can be found here:

```
version: latest
name: mobycounter
description: An example Docker App file which packages up the Moby Counter
application
namespace: masteringdockerthirdedition
maintainers:
  - name: Russ McKendrick
    email: russ@mckendrick.io
```

```
---
version: "3.6"

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:

---

{ "port":"8080" }
```

As you can see, it split into three sections; the first contains metadata about the application, as follows:

- `Version`: This the version of the app that will be published on Docker Hub
- `Name`: The name of the application as it will appear on Docker Hub
- `Description`: A short description of the application
- `Namespace`: This is typically your Docker Hub username or an organisation you have access to
- `Maintainers`: A list of maintainers for the application

The second section contains our Docker Compose file. You may notice that a few of the options have been replaced with variables. In our example, I have replaced port `8080` with `${port}` . The default value for the `port` variable is defined in the final section.

Once the `.dockerapp` file is complete, you can run the following command to save the Docker App as an image:

```
$ docker-app save
```

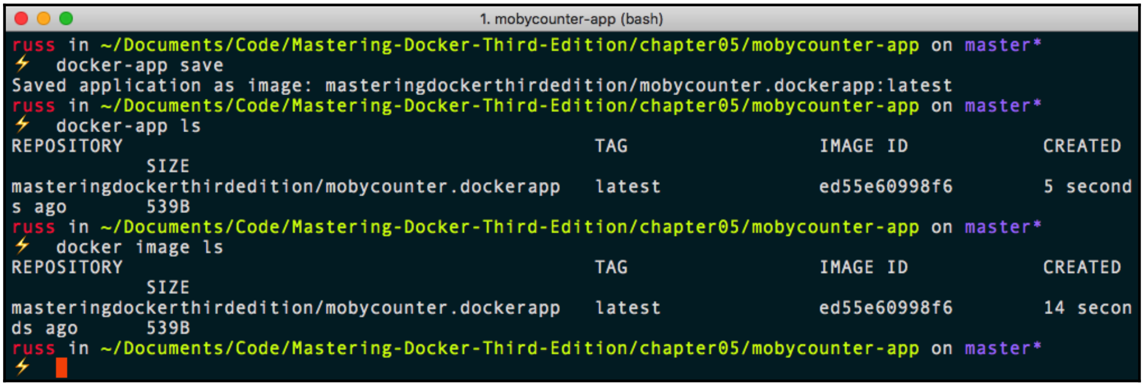
You can view just the Docker Apps you have active on your host by running this:

```
$ docker-app ls
```

As the Docker App is mostly just a bunch of metadata wrapped in a standard Docker image, you can also see it by running the following:

```
$ docker image ls
```

If you are not following along with this part, you can see the results in the terminal output here:

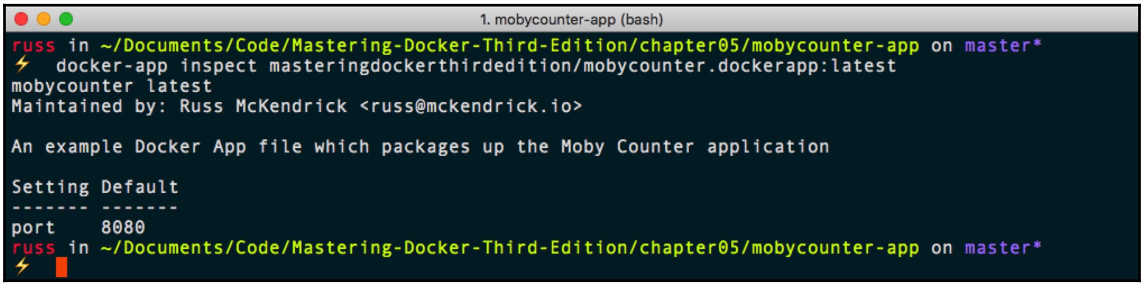


```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/mobycounter-app on master*
⚡  docker-app save
Saved application as image: masteringdockerthirdedition/mobycounter.dockerapp:latest
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/mobycounter-app on master*
⚡  docker-app ls
REPOSITORY                                         TAG                 IMAGE ID            CREATED
            SIZE
masteringdockerthirdedition/mobycounter.dockerapp   latest              ed55e60998f6        5 second
s ago       539B
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/mobycounter-app on master*
⚡  docker image ls
REPOSITORY                                         TAG                 IMAGE ID            CREATED
            SIZE
masteringdockerthirdedition/mobycounter.dockerapp   latest              ed55e60998f6        14 secon
ds ago      539B
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/mobycounter-app on master*
⚡
```

Running the following command gives an overview of the Docker App, in much the same way you can use `docker image inspect` to find out details on how the image was built:

```
$ docker-app inspect
masteringdockerthirdedition/mobycounter.dockerapp:latest
```

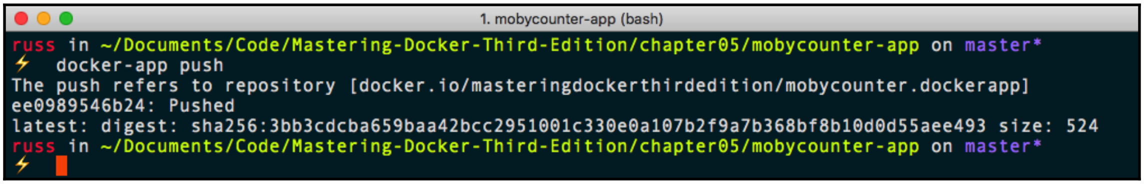
As you can see from the following terminal output, running the command using `docker-app inspect` rather than `docker image inspect` gives a much more friendly output:



```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/mobycounter-app on master*
⚡  docker-app inspect masteringdockerthirdedition/mobycounter.dockerapp:latest
mobycounter latest
Maintained by: Russ McKendrick <russ@mckendrick.io>

An example Docker App file which packages up the Moby Counter application

Setting Default
------- -------
port    8080
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/mobycounter-app on master*
⚡
```

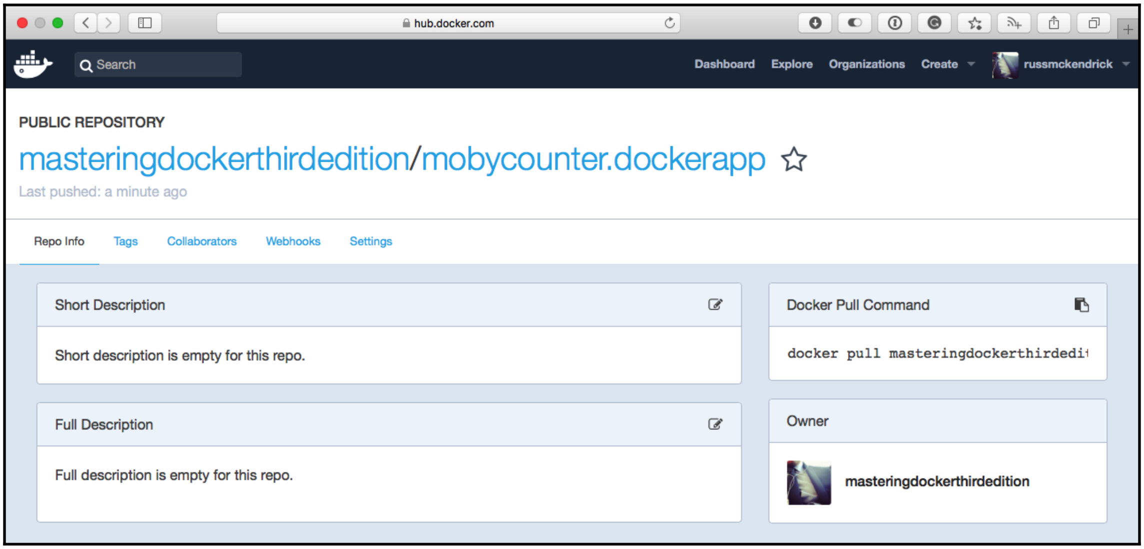
Now that we have our finished application, we need to push it to Docker Hub. To do this, simply run the following command:

```
$ docker-app push
```



```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/mobycounter-app on master*
⚡  docker-app push
The push refers to repository [docker.io/masteringdockerthirdedition/mobycounter.dockerapp]
ee0989546b24: Pushed
latest: digest: sha256:3bb3cdcba659baa42bcc2951001c330e0a107b2f9a7b368bf8b10d0d55aee493 size: 524
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter05/mobycounter-app on master*
⚡
```

This means that our application is now published on Docker Hub:



PUBLIC REPOSITORY

masteringdockerthirdedition/mobycounter.dockerapp

Last pushed: a minute ago

Short Description

Short description is empty for this repo.

Full Description

Full description is empty for this repo.

Docker Pull Command

`docker pull masteringdockerthirdedi`

Owner

masteringdockerthirdedition

So how do you get the Docker App? First of all, we need to remove the local image. To do this, run the following command:

```
$ docker image rm masteringdockerthirdedition/mobycounter.dockerapp:latest
```

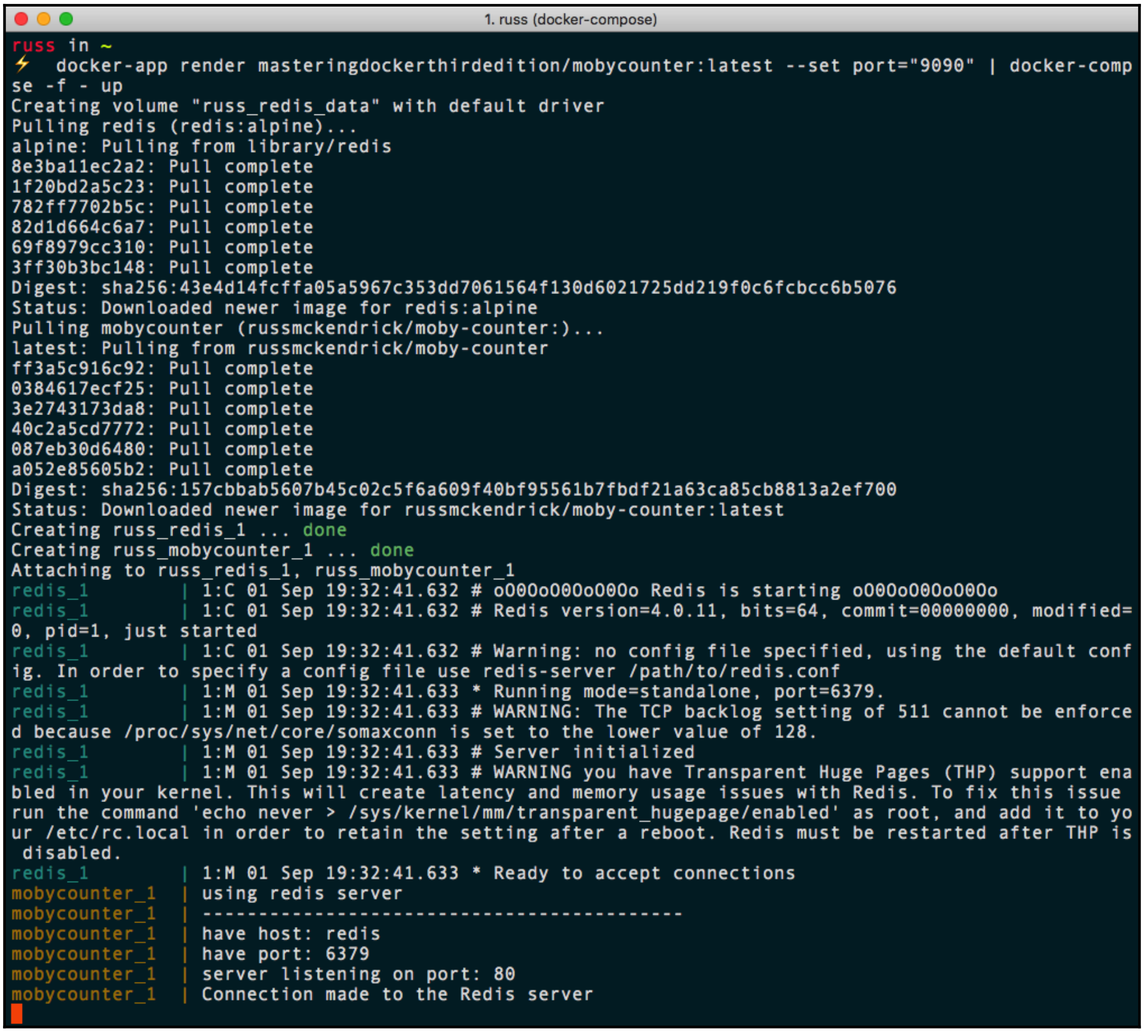
Once gone, move to a different directory:

```
$ cd ~/
```

Now, let's download the Docker App, make a change to the port, and start it up:

```
$ docker-app render masteringdockerthirdedition/mobycounter:latest --set
port="9090" | docker-compose -f - up
```

Again, for those not following along, the terminal output of the preceding command can be found here:



```
russ in ~
⚡  docker-app render masteringdockerthirdedition/mobycounter:latest --set port="9090" | docker-comp
se -f - up
Creating volume "russ_redis_data" with default driver
Pulling redis (redis:alpine)...
alpine: Pulling from library/redis
8e3ba11ec2a2: Pull complete
1f20bd2a5c23: Pull complete
782ff7702b5c: Pull complete
82d1d664c6a7: Pull complete
69f8979cc310: Pull complete
3ff30b3bc148: Pull complete
Digest: sha256:43e4d14fcffa05a5967c353dd7061564f130d6021725dd219f0c6fcbcc6b5076
Status: Downloaded newer image for redis:alpine
Pulling mobycounter (russmckendrick/moby-counter:)...
latest: Pulling from russmckendrick/moby-counter
ff3a5c916c92: Pull complete
0384617ecf25: Pull complete
3e2743173da8: Pull complete
40c2a5cd7772: Pull complete
087eb30d6480: Pull complete
a052e85605b2: Pull complete
Digest: sha256:157cbbab5607b45c02c5f6a609f40bf95561b7fbdf21a63ca85cb8813a2ef700
Status: Downloaded newer image for russmckendrick/moby-counter:latest
Creating russ_redis_1 ... done
Creating russ_mobycounter_1 ... done
Attaching to russ_redis_1, russ_mobycounter_1
redis_1         | 1:C 01 Sep 19:32:41.632 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
redis_1         | 1:C 01 Sep 19:32:41.632 # Redis version=4.0.11, bits=64, commit=00000000, modified=
0, pid=1, just started
redis_1         | 1:C 01 Sep 19:32:41.632 # Warning: no config file specified, using the default conf
ig. In order to specify a config file use redis-server /path/to/redis.conf
redis_1         | 1:M 01 Sep 19:32:41.633 * Running mode=standalone, port=6379.
redis_1         | 1:M 01 Sep 19:32:41.633 # WARNING: The TCP backlog setting of 511 cannot be enforce
d because /proc/sys/net/core/somaxconn is set to the lower value of 128.
redis_1         | 1:M 01 Sep 19:32:41.633 # Server initialized
redis_1         | 1:M 01 Sep 19:32:41.633 # WARNING you have Transparent Huge Pages (THP) support ena
bled 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 yo
ur /etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after THP is
 disabled.
redis_1         | 1:M 01 Sep 19:32:41.633 * 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
```

As you can see, without having to even manually download the Docker App image, we have our application up and running. Going to `http://localhost:9090/` should present you with the screen that invites you to click to add logos.

As per a normal foregrounded Docker Compose app, press *Ctrl* + *C* to return to your terminal.

You can run the following commands to interact and terminate your app:

```
$ docker-app render masteringdockerthirdedition/mobycounter:latest --set
port="9090" | docker-compose -f - ps
$ docker-app render masteringdockerthirdedition/mobycounter:latest --set
port="9090" | docker-compose -f - down --rmi all --volumes
```

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 9`, *Docker and Kubernetes*.

As mentioned at the top of this section, this feature is in its early stages of development and it is possible that the commands and functionality we have discussed so far may change in the future. But, even at this early 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 in 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 9`, *Docker and Kubernetes*.

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 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://www.orchardup.com/`
- Orchard Laboratories joins Docker: `https://blog.docker.com/2014/07/welcoming-the-orchard-and-fig-team`

For more information on the Docker App project, see the following:

- GitHub Repository: `http://github.com/docker/app/`
- Releases page – `https://github.com/docker/app/releases`

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
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.

In this chapter, we will cover 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

As per 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—yes, even though we are going to be running Windows containers, you can still use your macOS client. More on that later.

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, 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.

A full copy of the code used in this chapter can be found at `https://github.com/PacktPublishing/Mastering-Docker-Third-Edition/tree/master/chapter06/`.

Check out the following video to see the Code in Action:

`http://bit.ly/2PfjuSR`

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 that "Microsoft ❤ Linux", and it hasn't looked back since:

- Linux is a first-class citizen in Microsoft Azure
- .NET Core is cross-platform, meaning that you can run your .NET applications on Linux and Windows
- SQL Server is now available on Linux
- You can run Linux shells, such as Ubuntu, on Windows 10 Professional machines
- PowerShell has been ported to Linux
- It has developed cross-platform tools, such as Visual Studio Code, and open sourced them
- It is acquiring GitHub for $7.5 billion!!

It is clear that the Microsoft of old, where former 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, which was made in October 2014 months after Microsoft publicly declared its love of 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 to 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, 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 host operating systems without the worries and overhead of having to manage multiple virtual machines running older unsupported versions of Windows.

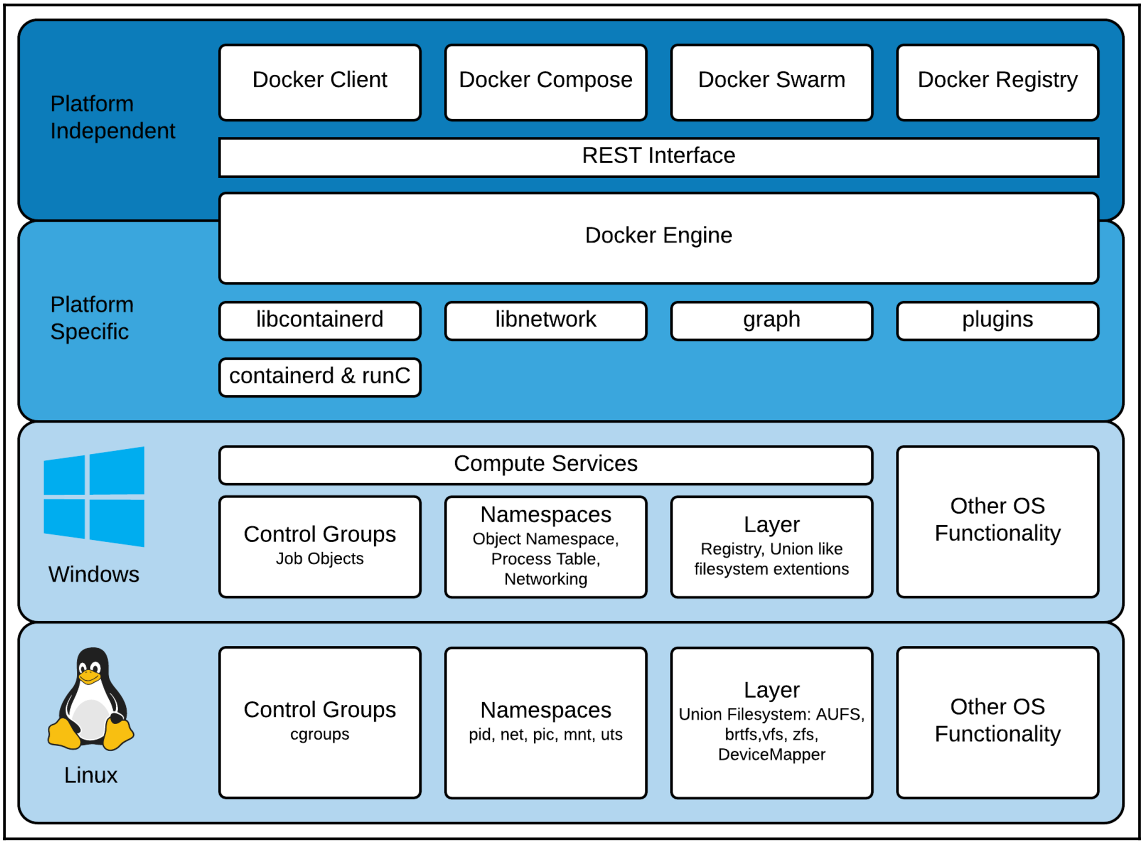
There is also another layer of isolation provided by Windows Containers. Hyper-V isolation runs the container processes within a minimal hypervisor when the container is started. This further isolates the container processes from the host machine. However, there is a cost of a small amount of additional resources 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 management tool?

Docker had already established itself as the go-to tool for managing containers with a set of proven APIs and a large community. Also, it was open source, 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:



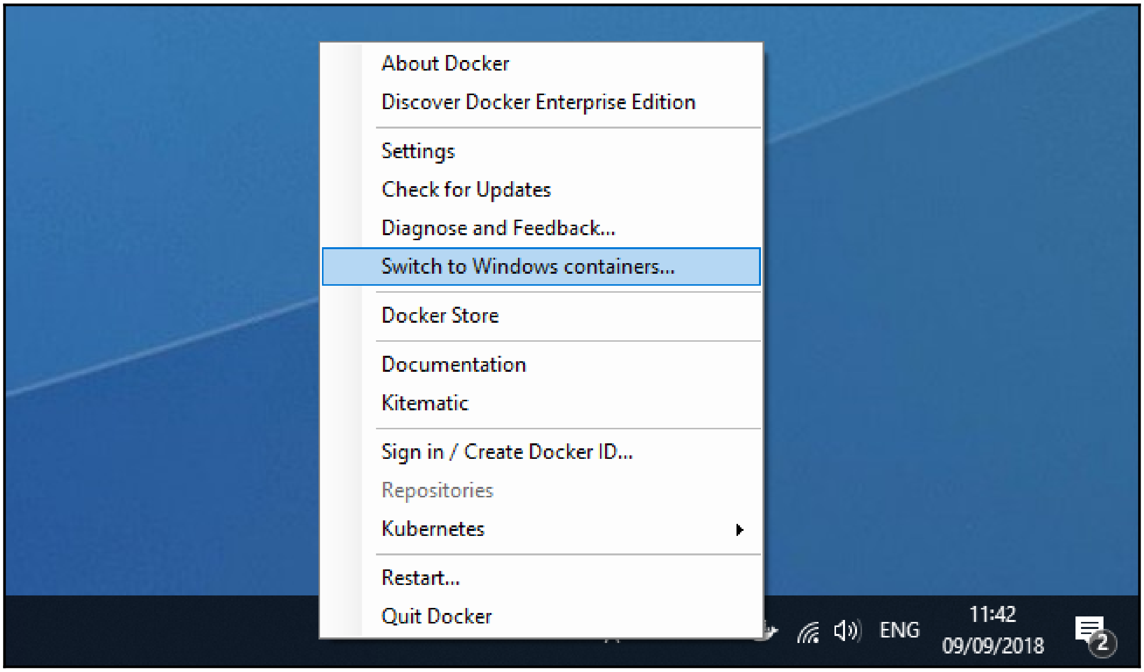
Notice that I said Docker on Windows, not Docker for Windows; they are very different products. Docker on Windows is the native version of the Docker Engine and client that interacts with the Windows kernel to provide Windows containers. Docker for Windows is a native as possible experience for developers to run both Linux and Windows containers on their desktops.

Setting up your Docker host for Windows containers

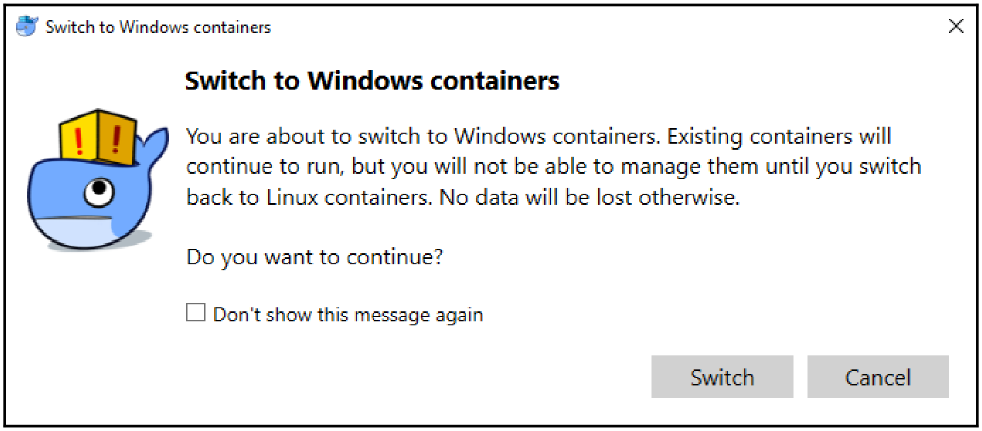
As you may have guessed, you are going to need access to a Windows host running Docker. Don't worry too much if you are not running a Windows 10 Professional machine—there are ways in which you can achieve this on macOS and Linux. Before we talk about those, let's look at how you can run Windows containers on Windows 10 Professional with your Docker for Windows installation.

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:



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.

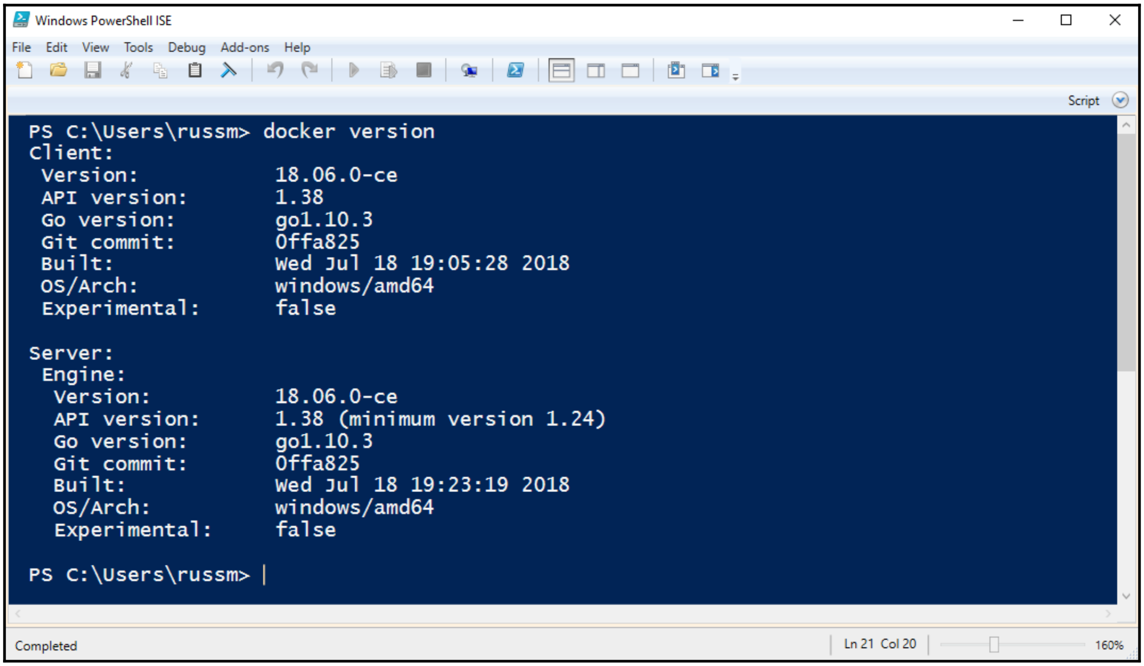
Do you want to continue?

□ Don't show this message again

Hit the **Switch** button and, after a few seconds, you will now be managing Windows containers. You can see this by opening up a prompt and running the following command:

```
$ docker version
```

This can be seen from the following output:



```
PS C:\Users\russm> docker version
Client:
 Version:           18.06.0-ce
 API version:       1.38
 Go version:        go1.10.3
 Git commit:        0ffa825
 Built:             Wed Jul 18 19:05:28 2018
 OS/Arch:           windows/amd64
 Experimental:      false

Server:
 Engine:
  Version:          18.06.0-ce
  API version:      1.38 (minimum version 1.24)
  Go version:       go1.10.3
  Git commit:       0ffa825
  Built:            Wed Jul 18 19:23:19 2018
  OS/Arch:          windows/amd64
  Experimental:     false

PS C:\Users\russm> |
```

The Docker Engine has an `OS/Arch` of `windows/amd64`, rather than the `linux/amd64` we have been used to seeing up until now. So that covers Windows 10 Professional. But what about people like me who prefer macOS and Linux?

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 `chapter06` folder of the repository that accompanies this book, there is a forked version of Stefan's Windows –`docker-machine repo`, which contains all of the files you need to get up and running with Windows containers on macOS.

Before we start, you will need the following tools – Vagrant by Hashicorp, and Virtualbox by Oracle. You can download these from:

- `https://www.vagrantup.com/downloads.html`
- `https://www.virtualbox.org/wiki/Downloads`

Once downloaded and installed, open a Terminal, go to the `chapter06/docker-machine` repository folder, and run the following command:

```
$ vagrant up --provider virtualbox 2016-box
```

This will download a VirtualBox Windows Server 2016 core eval image that contains everything needed to get you up and running with Windows containers. The download is just over 10 GB, so please make sure that you have the bandwidth and disk space needed to run the image.

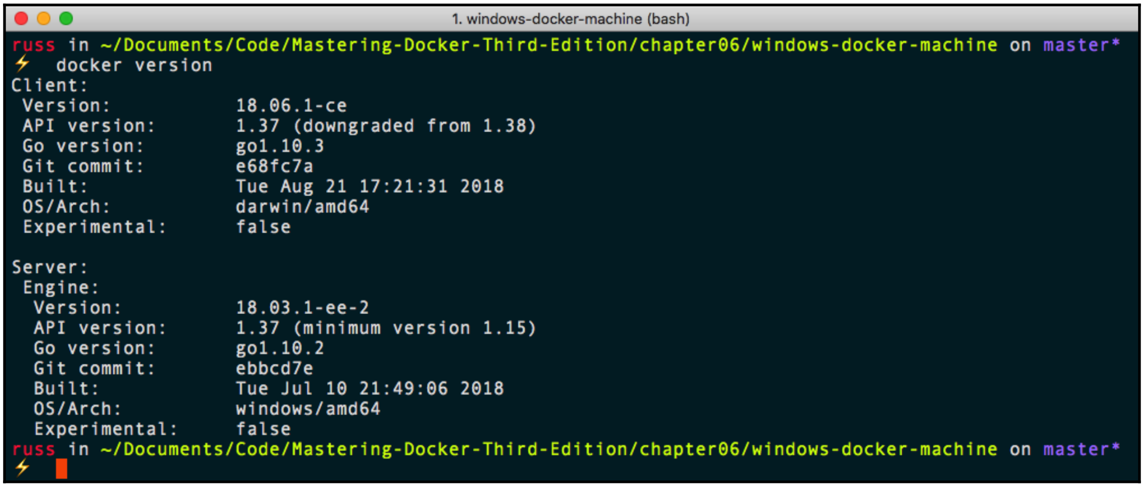
Vagrant will launch the image, configure Docker on the VM, and copy the certificate files needed for your local Docker client to interact with the host to your machine. To switch over to using the newly launched Docker Windows host, just run the following command:

```
$ eval $(docker-machine env 2016-box)
```

We will be going into more detail on Docker Machine in the next chapter. However, what the preceding command has done is reconfigure your local Docker client to speak to the Docker Windows host. You can see this by running the following command:

```
$ docker version
```

If you are not following along you can see the expected output below:



```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter06/windows-docker-machine on master*
⚡  docker version
Client:
 Version:           18.06.1-ce
 API version:       1.37 (downgraded from 1.38)
 Go version:        go1.10.3
 Git commit:        e68fc7a
 Built:             Tue Aug 21 17:21:31 2018
 OS/Arch:           darwin/amd64
 Experimental:      false

Server:
 Engine:
  Version:          18.03.1-ee-2
  API version:      1.37 (minimum version 1.15)
  Go version:       go1.10.2
  Git commit:       ebbcd7e
  Built:            Tue Jul 10 21:49:06 2018
  OS/Arch:          windows/amd64
  Experimental:     false
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter06/windows-docker-machine on master*
⚡  
```

As you can see, we are now connected to a Docker Engine running on `windows/amd64`. To switch back, you can either restart your terminal session or run the following command:

```
$ eval $(docker-machine env -unset)
```

Once you are finished with the Docker Windows host, you can run the following command to stop it:

```
$ vagrant halt
```

Alternatively, to remove it altogether, run the following command:

```
$ vagrant destroy
```

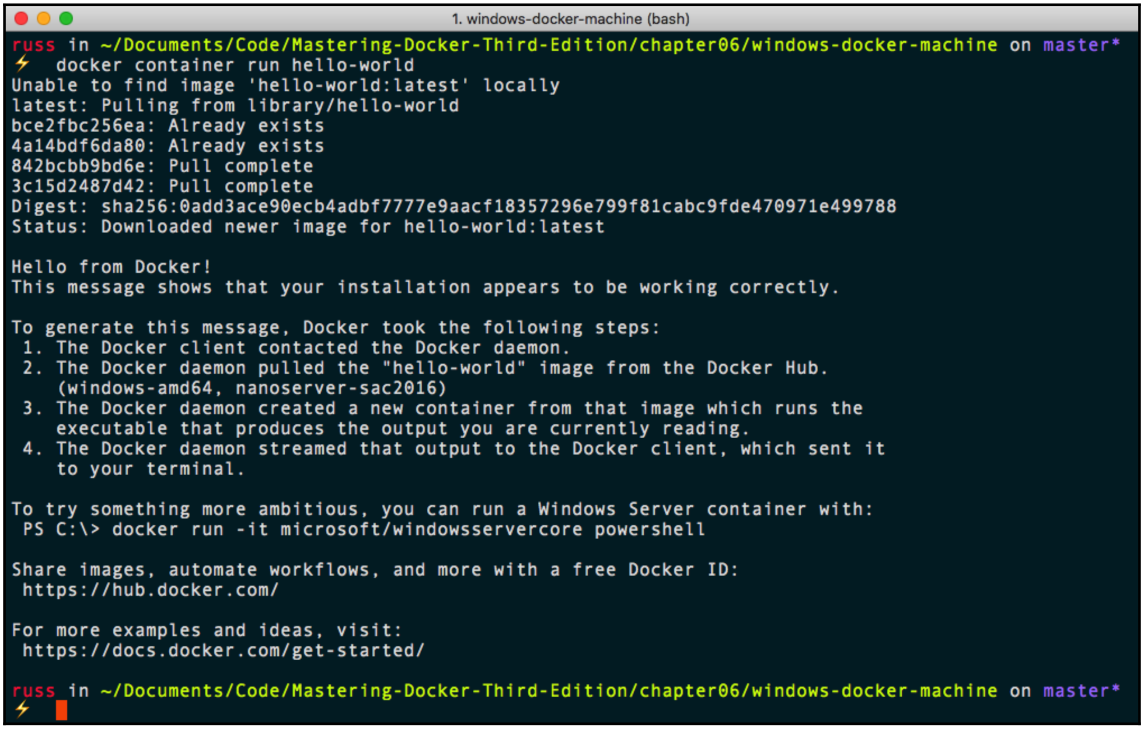
The preceding commands must be run from within the `chapter06/docker-machine` repository folder.

Running Windows containers

As already hinted at by the first part of this chapter, launching and interacting with Windows containers using the Docker command-line client is no different to what we have been running so far. Let's test this by running the `hello-world` container:

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

Just as before, this will download the `hello-world` container and return a message:



1. windows-docker-machine (bash)

```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter06/windows-docker-machine on master*
⚡  docker container run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
bce2fbc256ea: Already exists
4a14bdf6da80: Already exists
842bcbb9bd6e: Pull complete
3c15d2487d42: Pull complete
Digest: sha256:0add3ace90ecb4adbf7777e9aacf18357296e799f81cabc9fde470971e499788
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-sac2016)
 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 microsoft/windowsservercore 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/

russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter06/windows-docker-machine on master*
⚡
```

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.

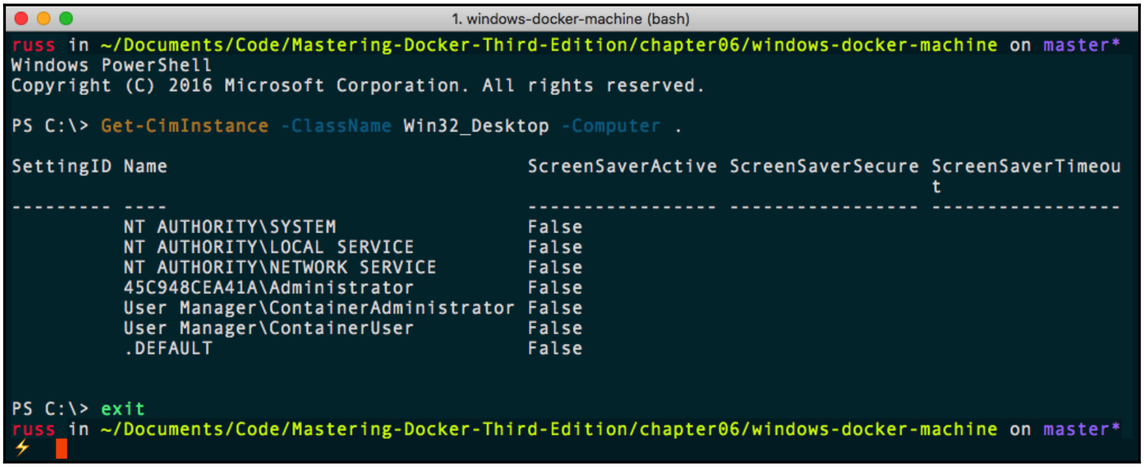
Now, let's look at running a container in the foreground, this time running PowerShell:

```
$ docker container run -it microsoft/windowsservercore 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 output of the commands above in the terminal output below:

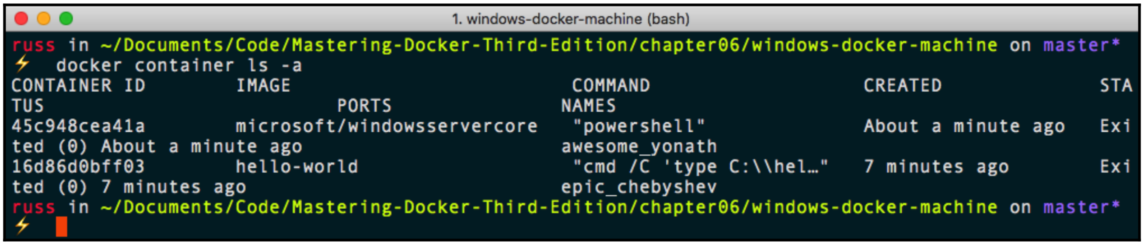


```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter06/windows-docker-machine on master*
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\> Get-CimInstance -ClassName Win32_Desktop -Computer .

SettingID Name                               ScreenSaverActive ScreenSaverSecure ScreenSaverTimeou
                                                                                 t
--------- ----                               ----------------- ----------------- -----------------
          NT AUTHORITY\SYSTEM                False
          NT AUTHORITY\LOCAL SERVICE         False
          NT AUTHORITY\NETWORK SERVICE       False
          45C948CEA41A\Administrator         False
          User Manager\ContainerAdministrator False
          User Manager\ContainerUser         False
          .DEFAULT                           False


PS C:\> exit
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter06/windows-docker-machine on master*
⚡
```

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 screen below:



```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter06/windows-docker-machine on master*
⚡  docker container ls -a
CONTAINER ID        IMAGE                                     COMMAND                   CREATED              STA
TUS                      PORTS                                NAMES
45c948cea41a        microsoft/windowsservercore                "powershell"              About a minute ago   Exi
ted (0) About a minute ago                                    awesome_yonath
16d86d0bff03        hello-world                                "cmd /C 'type C:\\hel…"   7 minutes ago        Exi
ted (0) 7 minutes ago                                         epic_chebyshev
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter06/windows-docker-machine on master*
⚡
```

Now, let's take a look at building an image that does something.

A Windows container Dockerfile

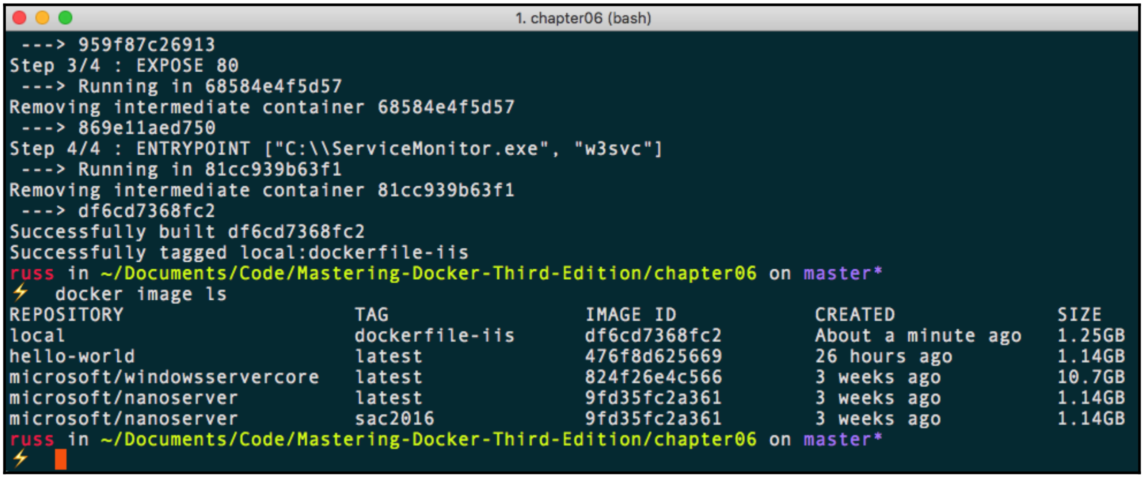
Windows container images use the same format Dockerfile commands as Linux containers. The following Dockerfile will download, install, and enable the IIS web server on the container:

```
# escape=`
FROM microsoft/nanoserver:sac2016

RUN powershell -NoProfile -Command `
    New-Item -Type Directory C:\install; `
    Invoke-WebRequest
https://az880830.vo.msecnd.net/nanoserver-ga-2016/Microsoft-NanoServer-IIS-
Package_base_10-0-14393-0.cab -OutFile C:\install\Microsoft-NanoServer-IIS-
Package_base_10-0-14393-0.cab; `
    Invoke-WebRequest
https://az880830.vo.msecnd.net/nanoserver-ga-2016/Microsoft-NanoServer-IIS-
Package_English_10-0-14393-0.cab -OutFile C:\install\Microsoft-NanoServer-
IIS-Package_English_10-0-14393-0.cab; `
    dism.exe /online /add-package /packagepath:c:\install\Microsoft-
NanoServer-IIS-Package_base_10-0-14393-0.cab & `
    dism.exe /online /add-package /packagepath:c:\install\Microsoft-
NanoServer-IIS-Package_English_10-0-14393-0.cab & `
    dism.exe /online /add-package /packagepath:c:\install\Microsoft-
NanoServer-IIS-Package_base_10-0-14393-0.cab & ;`
    powershell -NoProfile -Command `
    Remove-Item -Recurse C:\install\ ; `
    Invoke-WebRequest
https://dotnetbinaries.blob.core.windows.net/servicemonitor/2.0.1.3/Service
Monitor.exe -OutFile C:\ServiceMonitor.exe; `
    Start-Service Was; `
    While ((Get-ItemProperty
HKLM:\SYSTEM\CurrentControlSet\Services\WAS\Parameters\ -Name NanoSetup -
ErrorAction Ignore) -ne $null) {Start-Sleep 1}

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:



```
 ---> 959f87c26913
Step 3/4 : EXPOSE 80
 ---> Running in 68584e4f5d57
Removing intermediate container 68584e4f5d57
 ---> 869e11aed750
Step 4/4 : ENTRYPOINT ["C:\\ServiceMonitor.exe", "w3svc"]
 ---> Running in 81cc939b63f1
Removing intermediate container 81cc939b63f1
 ---> df6cd7368fc2
Successfully built df6cd7368fc2
Successfully tagged local:dockerfile-iis
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter06 on master*
⚡  docker image ls
REPOSITORY                    TAG                  IMAGE ID            CREATED               SIZE
local                         dockerfile-iis       df6cd7368fc2        About a minute ago    1.25GB
hello-world                   latest               476f8d625669        26 hours ago          1.14GB
microsoft/windowsservercore   latest               824f26e4c566        3 weeks ago           10.7GB
microsoft/nanoserver          latest               9fd35fc2a361        3 weeks ago           1.14GB
microsoft/nanoserver          sac2016              9fd35fc2a361        3 weeks ago           1.14GB
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter06 on master*
⚡
```

The one immediate thing you will notice about Windows container images is that they are big. This is something that is being worked on with the release of Server 2019.

Running the container with the following command will start the IIS image:

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

You can see your newly launched container in action by opening your browser. However, instead of going to `http://localhost:8080/`, you will need to access it via the NAT IP of the container. If you are using Windows 10 Professional, you can find the NAT IP by running the following command:

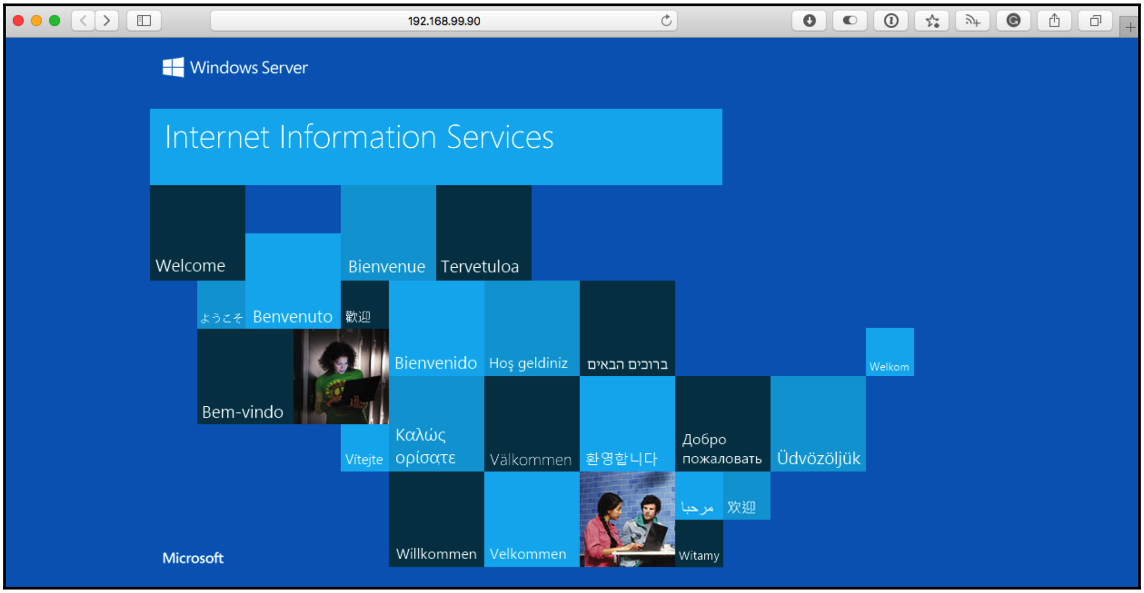
```
$ docker inspect --format="{{.NetworkSettings.Networks.nat.IPAddress}}"
dockerfile-iis
```

This will give you an IP address, simply augmented with `8080/` at the end; for example, `http://172.31.20.180:8080/`.

macOS users can run the following command to open their browsers using the IP address of the Vagrant VM we launched:

```
$ open http://$(docker-machine ip 2016-box):8080/
```

Whichever operating system you have launched your IIS container on, you should see the following default holding page:



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 to 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 the previous chapter. In the `chapter06` folder in the repository, there is a fork of the `dotnet-album-viewer` application from the Docker Examples repository as this ships with a `docker-compose.yml` file.

The Docker Compose file looks like the following:

```
version: '2.1'

services:
  db:
    image: microsoft/mssql-server-windows-express
    environment:
      sa_password: "DockerCon!!!"
      ACCEPT_EULA: "Y"
    healthcheck:
      test: [ "CMD", "sqlcmd", "-U", "sa", "-P", "DockerCon!!!", "-Q",
"select 1" ]
      interval: 2s
      retries: 10

  app:
    image: dockersamples/dotnet-album-viewer
    build:
      context: .
      dockerfile: docker/app/Dockerfile
    environment:
      - "Data:useSqLite=false"
      - "Data:SqlServerConnectionString=Server=db;Database=AlbumViewer;User
Id=sa;Password=DockerCon!!!;MultipleActiveResultSets=true;App=AlbumViewer"
    depends_on:
      db:
        condition: service_healthy
    ports:
      - "80:80"

networks:
  default:
    external:
      name: nat
```

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.

To build the required images, simply run the following command:

```
$ docker-compose build
```

Then, once built, launch using the following command:

```
$ docker-compose up -d
```

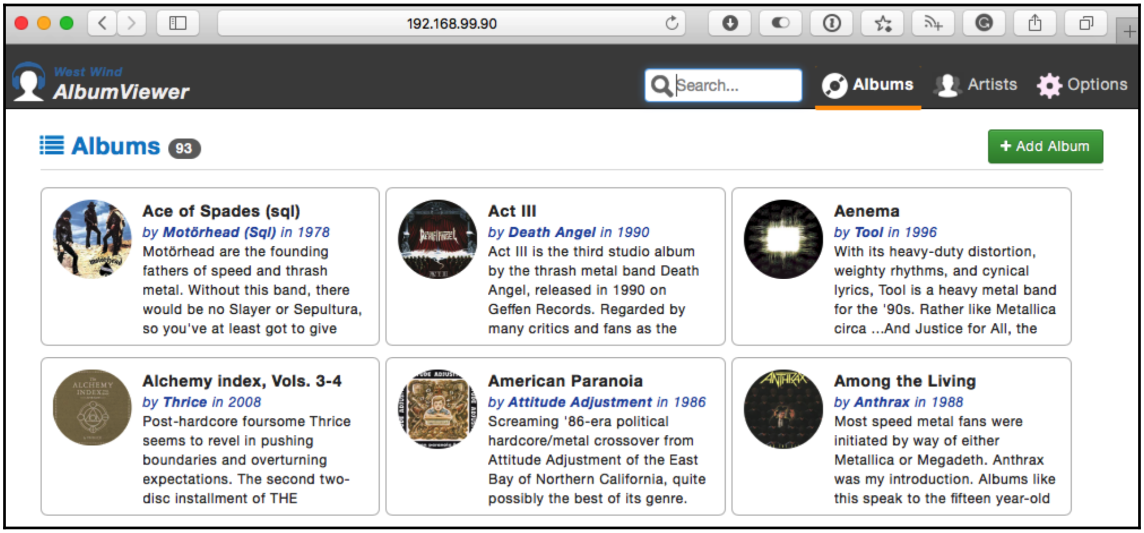
As before, you can then use this command to find out the IP address on Windows:

```
$ docker inspect -f "{{ .NetworkSettings.Networks.nat.IPAddress }}"
musicstore_web_1
```

To open the application you just need to put the IP address of your Docker host in your browser. If you are running using macOS, run the following command:

```
$ open http://$(docker-machine ip 2016-box)/
```

You should see the following page:



Ace of Spades (sql)
*by **Motörhead (Sql)** in 1978*
Motörhead are the founding fathers of speed and thrash metal. Without this band, there would be no Slayer or Sepultura, so you've at least got to give

Act III
*by **Death Angel** in 1990*
Act III is the third studio album by the thrash metal band Death Angel, released in 1990 on Geffen Records. Regarded by many critics and fans as the

Aenema
*by **Tool** in 1996*
With its heavy-duty distortion, weighty rhythms, and cynical lyrics, Tool is a heavy metal band for the '90s. Rather like Metallica circa ...And Justice for All, the

Alchemy index, Vols. 3-4
*by **Thrice** in 2008*
Post-hardcore foursome Thrice seems to revel in pushing boundaries and overturning expectations. The second two-disc installment of THE

American Paranoia
*by **Attitude Adjustment** in 1986*
Screaming '86-era political hardcore/metal crossover from Attitude Adjustment of the East Bay of Northern California, quite possibly the best of its genre.

Among the Living
*by **Anthrax** in 1988*
Most speed metal fans were initiated by way of either Metallica or Megadeth. Anthrax was my introduction. Albums like this speak to the fifteen year-old

Once you have finished with the application, you can run the following command to remove it:

```
$ docker-compose down --rmi all --volumes
```

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 more detailed look at Docker Machine.

Questions

1. Docker on Windows introduces what 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 and Containers to Windows: `https://www.youtube.com/watch?v=85nCF5S8Qok`
- Stefan Scherer on GitHub: `https://github.com/stefanScherer/`
- The `dotnet-album-viewer` repository: `https://github.com/dockersamples/dotnet-album-viewer`

7
Docker Machine

In this chapter, we will take a deeper look at Docker Machine, which we touched upon in the previous chapter. It can be used to easily launch and bootstrap Docker hosts targeting various platforms, including locally or in a cloud environment. You can control your Docker hosts with it as well. Let's take a look at what we will be covering in this chapter:

- An introduction to Docker Machine
- Using Docker Machine to set up local Docker hosts
- Launching Docker hosts in the cloud
- Using other base operating systems

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 locally using VirtualBox as well as in public clouds, so you will need an account with Digital Ocean if you would like to follow along with the example in this chapter.

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:

`http://bit.ly/2Ansb5v`

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 occupies in the overall Docker ecosystem.

Docker Machine's biggest strength is that it provides a consistent interface to several public cloud providers, such as Amazon Web Services, DigitalOcean, Microsoft Azure, and Google Cloud, as well as self-hosted virtual machine/cloud platforms, including OpenStack, and VMware vSphere. Finally, the following locally-hosted hypervisors are supported, such as Oracle VirtualBox and VMware Workstation or Fusion.

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 Amazon Web Services one day and then DigitialOcean the next—you know you are going to get a consistent experience.

As it is a command-line tool, it is also very easy to pass instructions to colleagues or even script the launch and tear down on Docker hosts: imagine starting work with your environment built fresh for you each morning and then, to save costs, it is torn down each evening.

Deploying local Docker hosts with Docker Machine

Before we journey out into the cloud, we are going to look at the basics of Docker Machine locally by launching it, using Oracle VirtualBox to provide the virtual machine.

> VirtualBox is a free virtualization product from Oracle. It allows you to install virtual machines across many different platforms and 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 to store all of its files and virtual machines:

```
Creating CA: /Users/russ/.docker/machine/certs/ca.pem
Creating client certificate: /Users/russ/.docker/machine/certs/cert.pem
Running pre-create checks...
(docker-local) Image cache directory does not exist, creating it at
/Users/russ/.docker/machine/cache...
```

It then checks for the presence of the image it will use for the virtual machine. If it is not there, the image will be downloaded:

```
(docker-local) No default Boot2Docker ISO found locally, downloading the
latest release...
(docker-local) Latest release for github.com/boot2docker/boot2docker is
v18.06.1-ce
(docker-local) Downloading
/Users/russ/.docker/machine/cache/boot2docker.iso from
https://github.com/boot2docker/boot2docker/releases/download/v18.06.1-ce/bo
ot2docker.iso...
(docker-local)
0%....10%....20%....30%....40%....50%....60%....70%....80%....90%....100%
```

Once the checks have passed, it creates the virtual machine using the selected driver:

```
Creating machine...
(docker-local) Copying /Users/russ/.docker/machine/cache/boot2docker.iso to
/Users/russ/.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 SSH key for the virtual machine. This means that you will be able to access the virtual machine over SSH, but more on that later. Once the virtual machine has booted, Docker Machine then makes a connection to the virtual machine:

```
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...
```

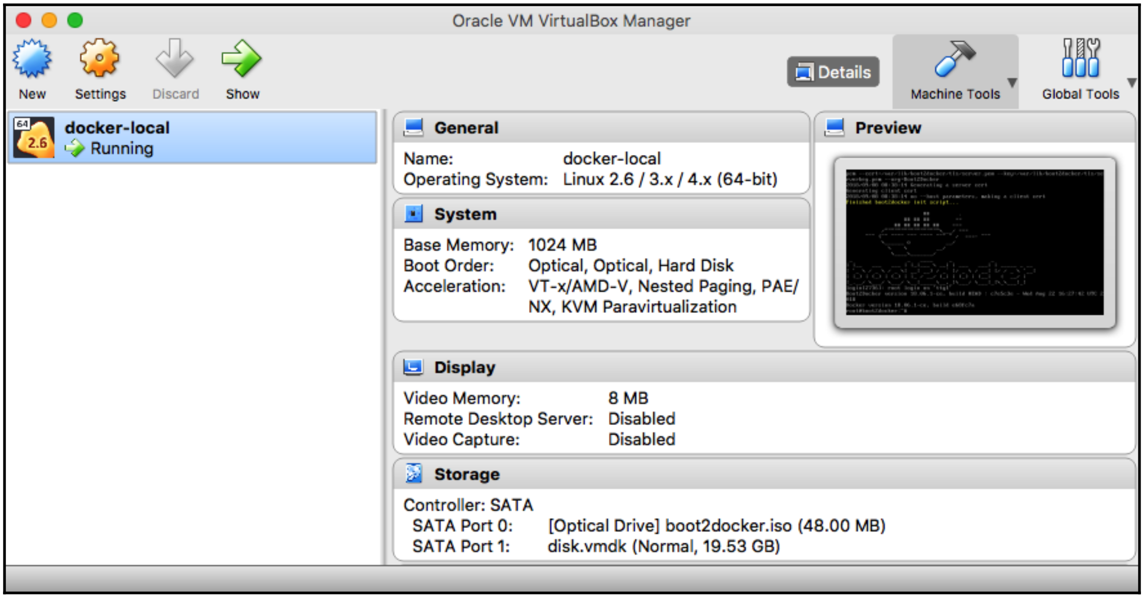
As you can see, Docker Machine detects the operating system being used and chooses the appropriate bootstrap script to deploy Docker. Once Docker is installed, Docker Machine generates and shares certificates between your local host and the Docker host. It then configures the remote Docker installation 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 local host 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:

```
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on
this virtual machine, 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 virtual machine:



docker-local
Running

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
Remote Desktop Server: Disabled
Video Capture: Disabled

Storage

Controller: SATA
SATA Port 0: [Optical Drive] boot2docker.iso (48.00 MB)
SATA Port 1: disk.vmdk (Normal, 19.53 GB)

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:

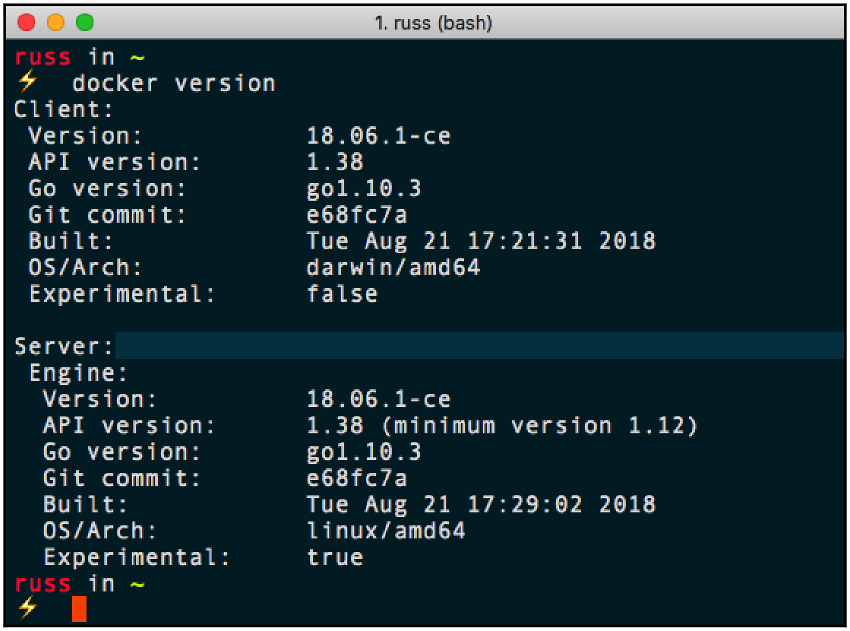
```
$ docker-machine env docker-local
```

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/.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:

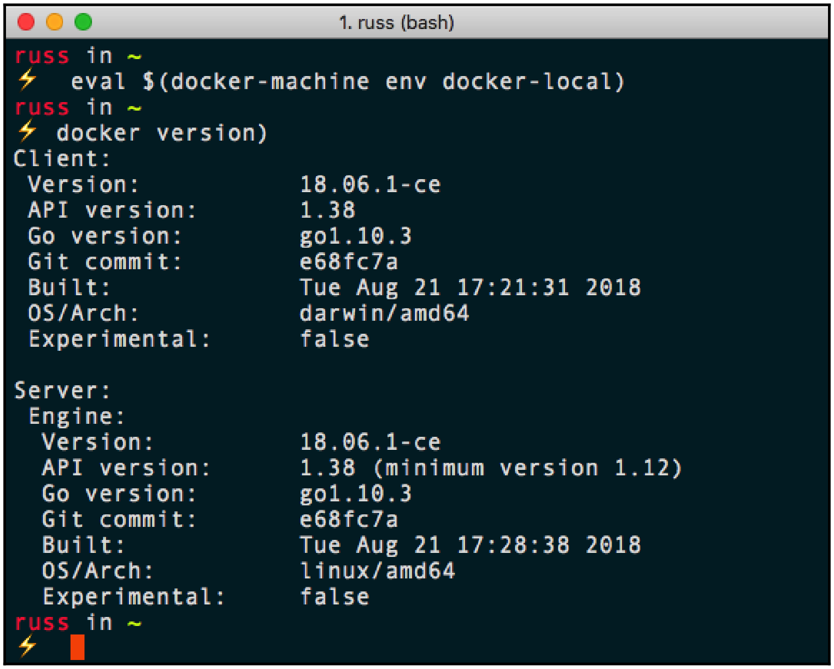


```
russ in ~
⚡  docker version
Client:
 Version:           18.06.1-ce
 API version:       1.38
 Go version:        go1.10.3
 Git commit:        e68fc7a
 Built:             Tue Aug 21 17:21:31 2018
 OS/Arch:           darwin/amd64
 Experimental:      false

Server:
 Engine:
  Version:          18.06.1-ce
  API version:      1.38 (minimum version 1.12)
  Go version:       go1.10.3
  Git commit:       e68fc7a
  Built:            Tue Aug 21 17:29:02 2018
  OS/Arch:          linux/amd64
  Experimental:     true
russ in ~
⚡
```

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 here:



```
russ in ~
⚡  eval $(docker-machine env docker-local)
russ in ~
⚡ docker version)
Client:
 Version:           18.06.1-ce
 API version:       1.38
 Go version:        go1.10.3
 Git commit:        e68fc7a
 Built:             Tue Aug 21 17:21:31 2018
 OS/Arch:           darwin/amd64
 Experimental:      false

Server:
 Engine:
  Version:          18.06.1-ce
  API version:      1.38 (minimum version 1.12)
  Go version:       go1.10.3
  Git commit:       e68fc7a
  Built:            Tue Aug 21 17:28:38 2018
  OS/Arch:          linux/amd64
  Experimental:     false
russ in ~
⚡
```

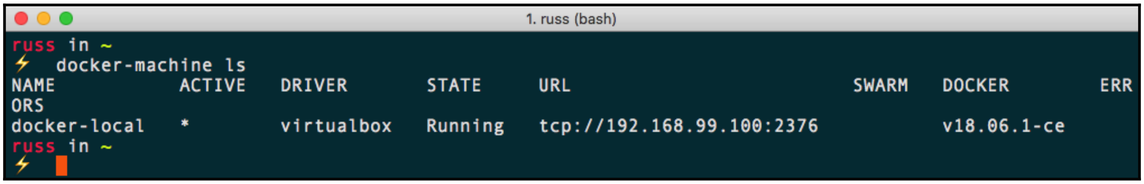
As you can see, the server launched by Docker Machine is pretty much in line with what we have installed locally; in fact, the only difference is the build time. As you can see, the Docker Engine binary on my Docker for Mac installation was built one minute after the Docker Machine version.

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 lists the currently configured Docker hosts:

```
$ docker-machine ls
```

The output of the command is given here:



```
russ in ~
⚡  docker-machine ls
NAME           ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER        ERR
ORS
docker-local   *        virtualbox   Running   tcp://192.168.99.100:2376           v18.06.1-ce
russ in ~
⚡
```

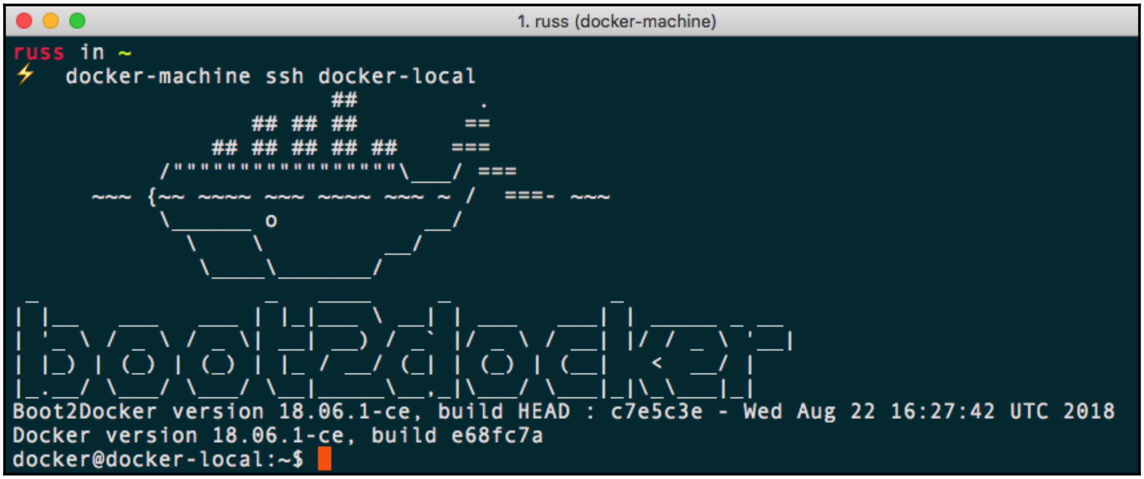
As you can see, it lists the details on the machine name, the driver used and the Docker endpoint URL, as well as the version of Docker the hosts are running.

You will also notice that there is an `*` 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:

```
$ docker-machine ssh docker-local
```

The output of the command is given here:



1. russ (docker-machine)

```
russ in ~
⚡  docker-machine ssh docker-local
                        ##         .
                  ## ## ##        ==
               ## ## ## ## ##    ===
           /"""""""""""""""""\___/ ===
      ~~~ {~~ ~~~~ ~~~ ~~~~ ~~~ ~ /  ===- ~~~
           \______ o           __/
             \    \         __/
              \____\_______/
 _                 _   ____     _            _
| |__   ___   ___ | |_|___ \ __| | ___   ___| | _____ _ __
| '_ \ / _ \ / _ \| __| __) / _` |/ _ \ / __| |/ / _ \ '__|
| |_) | (_) | (_) | |_ / __/ (_| | (_) | (__|   <  __/ |
|_.__/ \___/ \___/ \__|_____\__,_|\___/ \___|_|\_\___|_|
Boot2Docker version 18.06.1-ce, build HEAD : c7e5c3e - Wed Aug 22 16:27:42 UTC 2018
Docker version 18.06.1-ce, build e68fc7a
docker@docker-local:~$
```

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, as 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 a lot later in the chapter. There are also commands for establishing more details about your Docker host:

```
$ 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 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:

```
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

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. The only thing we need to do this is an API access token. Rather than explaining how to generate one here, you can follow the instructions at `https://www.digitalocean.com/help/api/`.

> 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 we are going to do is set our token as an environment variable so we don't have to keep using it. To do this, run the following command, making sure you replace the API token with your own:

```
$ DOTOKEN=0cb54091fecfe743920d0e6d28a29fe325b9fc3f2f6fccba80ef4b26d41c7224
```

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.

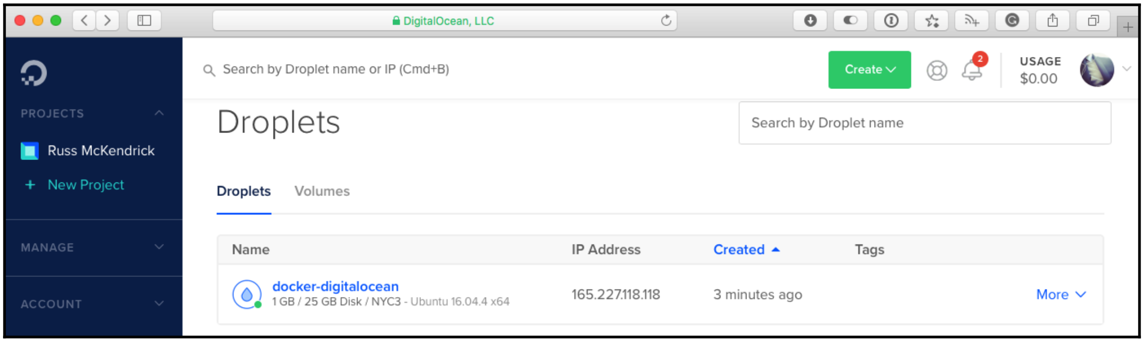
To launch a Docker host called `docker-digtialocean`, we need to run the following command:

```
$ 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 virtual machine, run: docker-machine env docker-digitalocean
```

Once launched, you should be able to see the Docker host in your DigitalOcean control panel:



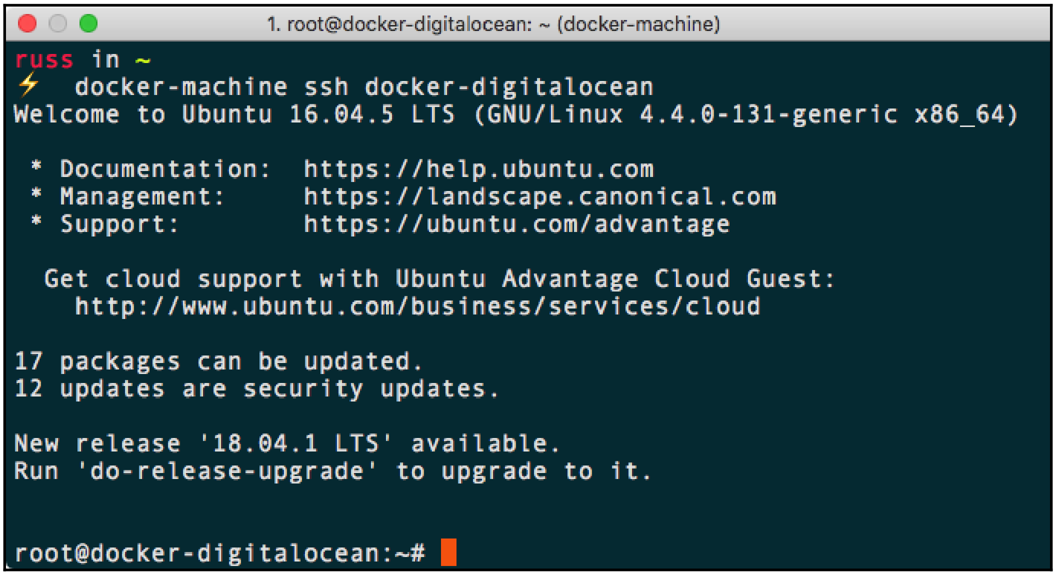
Name	IP Address	Created	Tags
docker-digitalocean 1 GB / 25 GB Disk / NYC3 - Ubuntu 16.04.4 x64	165.227.118.118	3 minutes ago	

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:



```
russ in ~
⚡  docker-machine ssh docker-digitalocean
Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-131-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

17 packages can be updated.
12 updates are security updates.

New release '18.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


root@docker-digitalocean:~#
```

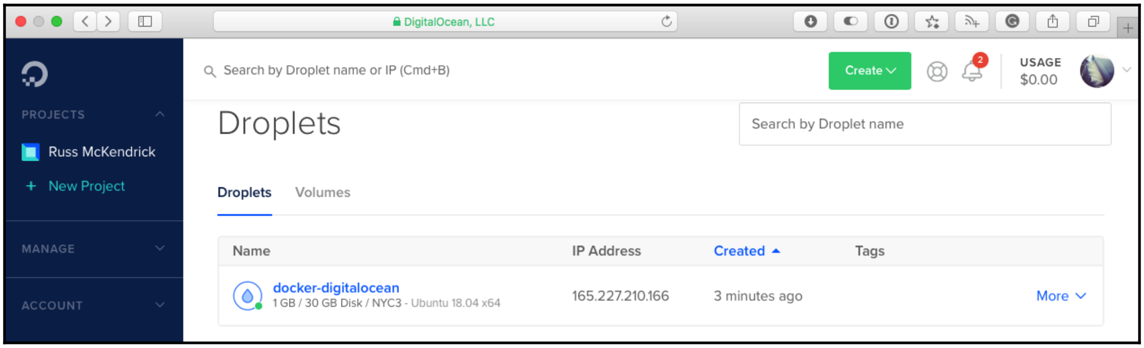
You can exit the remote shell by running `exit`. As you can see, we didn't have to tell Docker Machine which operating system to use, the size of the Docker host, or even where to launch it. That is because each driver has some pretty sound defaults. Adding these defaults to our command makes it look 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 false \
  --digitalocean-private-networking false \
  --digitalocean-backups false \
  --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. 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:



Name	IP Address	Created ▲	Tags	
docker-digitalocean 1 GB / 30 GB Disk / NYC3 - Ubuntu 18.04 x64	165.227.210.166	3 minutes ago		More

You can remove the DigitalOcean Docker host by running the following command:

```
$ docker-machine rm docker-digitalocean
```

Using other base operating systems

You don't have to use the default operating systems with Docker Machine; it does come with provisioners for other base operating systems, including ones that are geared toward running containers. Before we finish the chapter, we are going to take a look at launching one of these, CoreOS.

The distribution we are going to look at has just enough of an operating system to run a kernel, networking stack, and containers, just like Docker's own MobyOS, which is used as the base for Docker for Mac and Docker for Windows.

While CoreOS supports its own container runtime, called RKT (pronounced Rocket), it also ships with Docker. However, as we will see, the version of Docker currently shipping with the stable version of CoreOS is a little out of date.

To launch the DigitalOcean-managed `coreos-stable` version, run the following command:

```
$ docker-machine create \
  --driver digitalocean \
  --digitalocean-access-token $DOTOKEN \
  --digitalocean-image coreos-stable \
  --digitalocean-size 1GB \
  --digitalocean-ssh-user core \
  docker-coreos
```

As with launching our other Docker hosts on public clouds, the output is pretty much the same. You will notice that Docker Machine uses the CoreOS provisioner:

```
Running pre-create checks...
Creating machine...
(docker-coreos) Creating SSH key...
(docker-coreos) Creating Digital Ocean droplet...
(docker-coreos) 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 coreOS...
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 virtual machine, run: docker-machine env docker-coreos
```

Once launched, you can run the following:

```
$ docker-machine ssh docker-coreos cat /etc/*release
```

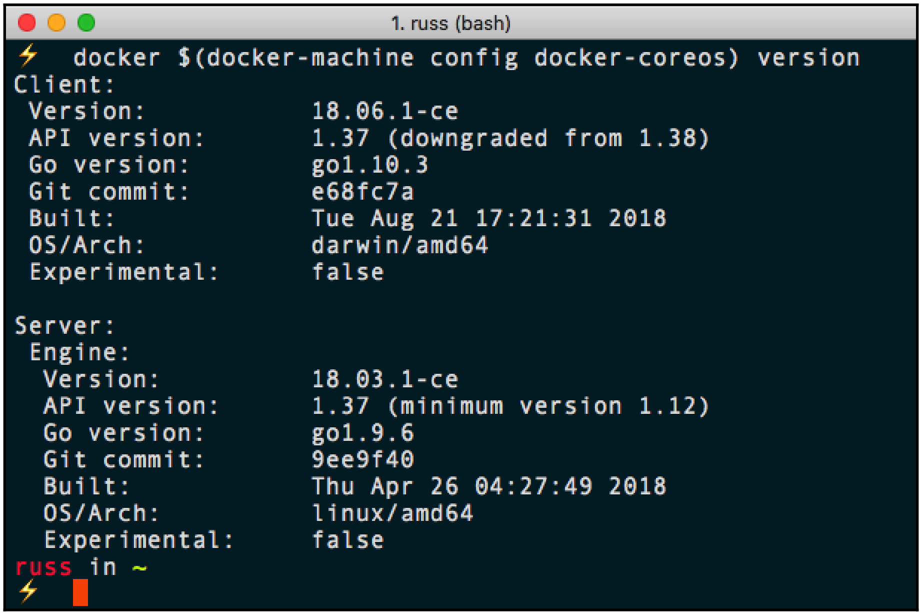
This will return the content of the `release` file:

```
DISTRIB_ID="Container Linux by CoreOS"
DISTRIB_RELEASE=1800.7.0
DISTRIB_CODENAME="Rhyolite"
DISTRIB_DESCRIPTION="Container Linux by CoreOS 1800.7.0 (Rhyolite)"
NAME="Container Linux by CoreOS"
ID=coreos
VERSION=1800.7.0
VERSION_ID=1800.7.0
BUILD_ID=2018-08-15-2254
PRETTY_NAME="Container Linux by CoreOS 1800.7.0 (Rhyolite)"
ANSI_COLOR="38;5;75"
HOME_URL="https://coreos.com/"
BUG_REPORT_URL="https://issues.coreos.com"
COREOS_BOARD="amd64-usr"
```

Running the following will show you more information on the version of Docker that is running on the CoreOS host:

```
$ docker $(docker-machine config docker-coreos) version
```

You can see this from the following output; also, as already mentioned, it is behind the current release:



```
⚡  docker $(docker-machine config docker-coreos) version
Client:
 Version:           18.06.1-ce
 API version:       1.37 (downgraded from 1.38)
 Go version:        go1.10.3
 Git commit:        e68fc7a
 Built:             Tue Aug 21 17:21:31 2018
 OS/Arch:           darwin/amd64
 Experimental:      false

Server:
 Engine:
  Version:          18.03.1-ce
  API version:      1.37 (minimum version 1.12)
  Go version:       go1.9.6
  Git commit:       9ee9f40
  Built:            Thu Apr 26 04:27:49 2018
  OS/Arch:          linux/amd64
  Experimental:     false
russ in ~
⚡
```

This means that not all of the commands we are using in this book may work. To remove the CoreOS host, run the following command:

```
$ docker-machine rm docker-coreos
```

Summary

In this chapter, we looked at how to use Docker Machine to create the 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 your cloud environments, namely DigitalOcean. Finally, we took a very quick look at how to launch a different container-optimized Linux operating system, which was CoreOS.

I am sure you will agree that using Docker Machine made running these tasks, which typically have very different approaches, a very consistent experience, and which, in the long run, will save a lot of time as well as explaining.

In the next chapter, we are going to move away from interacting with single Docker hosts to launching and running a Docker Swarm cluster.

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 you local Docker client to interact with `my-host`?
3. Explain the basic principle behind Docker Machine.

Further reading

For information on the various platforms supported by Docker Machine, refer to the following:

- Amazon Web Services: `https://aws.amazon.com/`
- Microsoft Azure: `https://azure.microsoft.com/`
- DigitalOcean: `https://www.digitalocean.com/`
- Exoscale: `https://www.exoscale.ch/`
- Google Compute Engine: `https://cloud.google.com/`
- Rackspace: `https://www.rackspace.com/`
- IBM SoftLayer: `https://www.softlayer.com/`
- Microsoft Hyper-V: `https://www.microsoft.com/en-gb/cloud-platform/server-virtualization/`
- OpenStack: `https://www.openstack.org/`
- VMware vSphere: `https://www.vmware.com/uk/products/vsphere.html`
- Oracle VirtualBox: `https://www.virtualbox.org/`
- VMware Fusion: `https://www.vmware.com/uk/products/fusion.html`
- VMware Workstation: `https://www.vmware.com/uk/products/workstation.html`
- CoreOS: `https://coreos.com/`

8
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
- Roles within a Docker Swarm cluster
- Creating and managing a Swarm
- Docker Swarm services and stacks
- Docker Swarm load balancing 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:

`http://bit.ly/2yWA4gl`

Introducing Docker Swarm

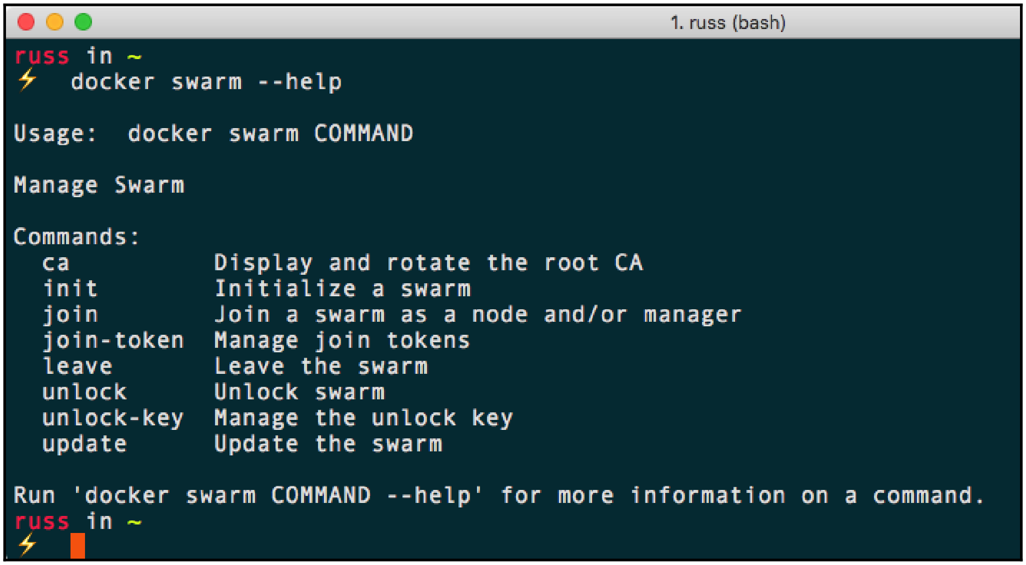
Before we go any further, I should mention that there are two very different versions of Docker Swarm. There was a standalone version of Docker Swarm; this was supported up until Docker 1.12 and is no longer being actively developed; however, you may find some old documentation mentions it. Installation 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 into the core Docker engine, along with a significant number of additional features. As we are covering Docker 18.06 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:



```
russ in ~
⚡  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.
russ in ~
⚡
```

If you get an error, ensure that you are running Docker 18.06 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 decided 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.

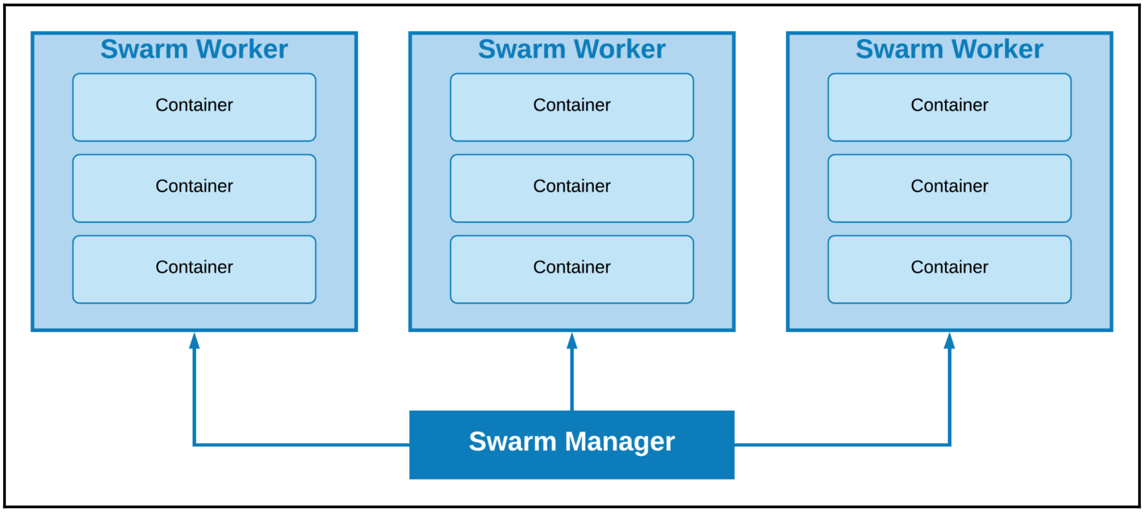
Swarm manager

The **Swarm manager** is a host that is the central management point for all Swarm hosts. 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 worker

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:



This is an illustration of all the Docker Swarm components. We see that the Docker Swarm manager talks to each Swarm host that has the role of Docker Swarm workers. 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 a cluster

Let's start by creating a cluster, which starts with a Swarm manager. Since we are going to be creating a multi-node cluster on our local machine, we should use Docker Machine to launch a host by running this command:

```
$ docker-machine create \
    -d virtualbox \
    swarm-manager
```

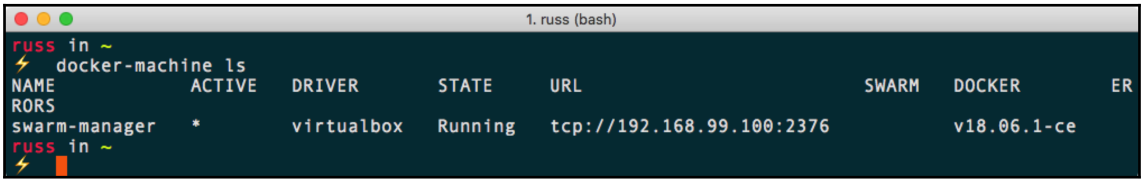
An abridged version of the output you get is shown here:

```
(swarm-manager) Creating VirtualBox VM...
(swarm-manager) Starting the VM...
(swarm-manager) Check network to re-create if needed...
(swarm-manager) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on
this virtual machine, run: docker-machine env swarm-manager
```

The Swarm manager node is now up and running using VirtualBox. We can confirm this by running the following command:

```
$ docker-machine ls
```

You should see something similar to the following output:



```
russ in ~
⚡  docker-machine ls
NAME          ACTIVE    DRIVER        STATE     URL                         SWARM    DOCKER         ER
RORS
swarm-manager  *        virtualbox    Running   tcp://192.168.99.100:2376           v18.06.1-ce
russ in ~
⚡
```

Now, let's point Docker Machine at the new Swarm manager. From the preceding output when we created the Swarm manager, we can see it is telling us how to point to the node:

```
$ docker-machine env swarm-manager
```

This will show you the commands needed to configure your local Docker client to talk to our newly launched Docker host. The following block of code shows the configuration returned when I ran the command:

```
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/russ/.docker/machine/machines/swarm-
manager"
export DOCKER_MACHINE_NAME="swarm-manager"
# Run this command to configure your shell:
# eval $(docker-machine env swarm-manager)
```

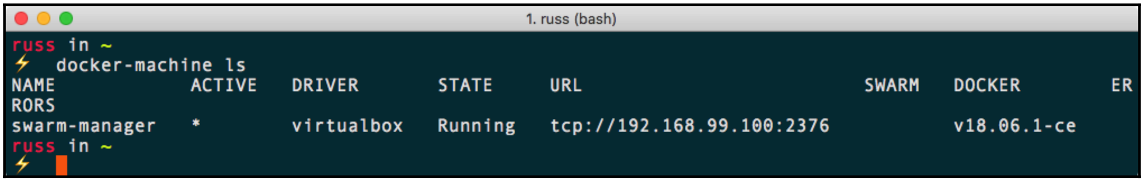
Upon running the previous command, we are told to run the following command to point to the Swarm manager:

```
$ eval $(docker-machine env swarm-manager)
```

Now, if we look at which machines are on our host, we can see that we have the Swarm master host, as well as it now being set to `ACTIVE`, which means we can now run commands on it:

```
$ docker-machine ls
```

It should show you something like the following:



```
russ in ~
⚡  docker-machine ls
NAME            ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER        ER
RORS
swarm-manager   *        virtualbox   Running   tcp://192.168.99.100:2376           v18.06.1-ce
russ in ~
⚡
```

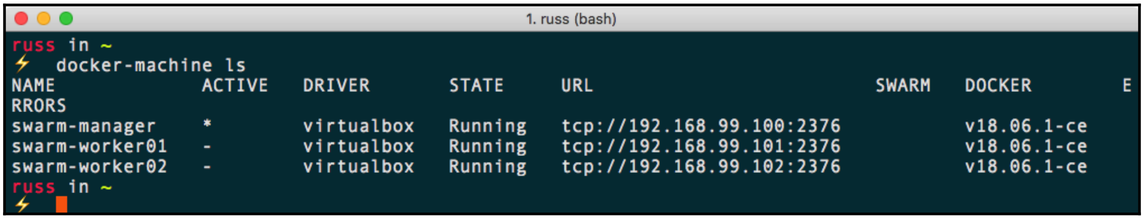
Now that we have the first host up and running, we should add the two worker nodes. To do this, simply run the following command to launch two more Docker hosts:

```
$ docker-machine create \
    -d virtualbox \
    swarm-worker01
$ docker-machine create \
    -d virtualbox \
    swarm-worker02
```

Once you have launched the two additional hosts, you can get the list of hosts using this command:

```
$ docker-machine ls
```

It should show you something like the following:



```
russ in ~
⚡  docker-machine ls
NAME             ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER        E
RRORS
swarm-manager    *        virtualbox   Running   tcp://192.168.99.100:2376           v18.06.1-ce
swarm-worker01   -        virtualbox   Running   tcp://192.168.99.101:2376           v18.06.1-ce
swarm-worker02   -        virtualbox   Running   tcp://192.168.99.102:2376           v18.06.1-ce
russ in ~
⚡
```

It is worth pointing out that, so far, we have not done anything to create our Swarm cluster; we have only launched the hosts it will be running on.

> You may have noticed that one of the columns when running the `docker-machine ls` command is `SWARM`. This only contains information if you have launched your Docker hosts using the standalone Docker Swarm command, which is built into Docker Machine.

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. The command to run in order to create our manager is as follows:

```
$ docker $(docker-machine config swarm-manager) swarm init \
    --advertise-addr $(docker-machine ip swarm-manager):2377 \
    --listen-addr $(docker-machine ip swarm-manager):2377
```

You should receive a message similar to this one:

```
Swarm initialized: current node (uxgvqhw6npr9glhp0zpabn4ha) is now a
manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token
SWMTKN-1-1uulmpx4j4hub2qmd8q2ozxmonzcehxcomt7cw92xarg3yrkx2-
dfiqnfisl75bwwh8yk9pv3msh 192.168.99.100:2377

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 `SWMTKN-1-1uulmpx4j4hub2qmd8q2ozxmonzcehxcomt7cw92xarg3yrkx2-dfiqnfisl75bwwh8yk9pv3msh`. This token will be needed for the worker nodes to authenticate themselves and join our cluster.

Joining Swarm workers to the cluster

To add our two workers to the cluster, run the following commands. First, let's set an environment variable to hold our token, making sure you replace the token with the one you received when initializing your own manager:

```
$ SWARM_TOKEN=SWMTKN-1-1uulmpx4j4hub2qmd8q2ozxmonzcehxcomt7cw92xarg3yrkx2-
dfiqnfisl75bwwh8yk9pv3msh
```

Now we can run the following command to add `swarm-worker01` to the cluster:

```
$ docker $(docker-machine config swarm-worker01) swarm join \
     --token $SWARM_TOKEN \
     $(docker-machine ip swarm-manager):2377
```

For `swarm-worker02`, you need to run the following command:

```
$ docker $(docker-machine config swarm-worker02) swarm join \
     --token $SWARM_TOKEN \
     $(docker-machine ip swarm-manager):2377
```

Both times, you should get confirmation that your node has joined the cluster:

```
This node joined a swarm as a worker.
```

Listing nodes

You can check the Swarm by running the following command:

```
$ docker-machine ls
```

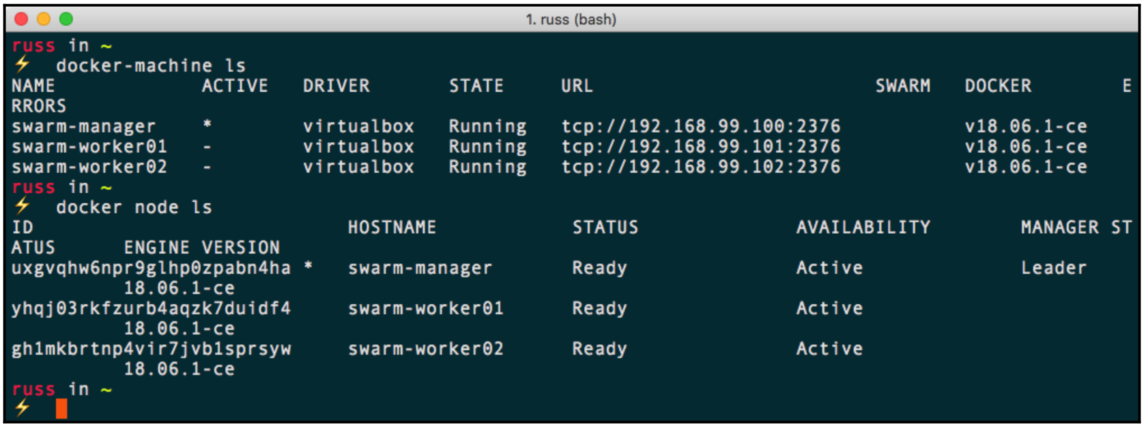
Check that your local Docker client is still configured to connect to the Swarm manager node, and if it isn't, rerun the following command:

```
$ eval $(docker-machine env swarm-manager)
```

Now that we are connecting to the Swarm manager node, you can run the following command:

```
$ docker node ls
```

This will connect to the Swarm master and query all of the nodes that form our cluster. You should see that all three of our nodes are listed:



1. russ (bash)

```
russ in ~
⚡  docker-machine ls
NAME             ACTIVE    DRIVER       STATE     URL                         SWARM   DOCKER        E
RRORS
swarm-manager    *         virtualbox   Running   tcp://192.168.99.100:2376           v18.06.1-ce
swarm-worker01   -         virtualbox   Running   tcp://192.168.99.101:2376           v18.06.1-ce
swarm-worker02   -         virtualbox   Running   tcp://192.168.99.102:2376           v18.06.1-ce
russ in ~
⚡  docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER ST
ATUS          ENGINE VERSION
uxgvqhw6npr9glhp0zpabn4ha *   swarm-manager       Ready               Active              Leader
              18.06.1-ce
yhqj03rkfzurb4aqzk7duidf4     swarm-worker01      Ready               Active
              18.06.1-ce
gh1mkbrtnp4vir7jvb1sprsyw     swarm-worker02      Ready               Active
              18.06.1-ce
russ in ~
⚡
```

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 these Swarm hosts and the containers on each host that you are creating, but first, you need to know some information about them.

Finding information on the cluster

As we have already seen, we can list the nodes within the cluster using our local Docker client, as it is already configured to connect to the Swarm manager host. We can simply type this:

```
$ docker info
```

This will give us lots of information about the host, as you can see from the following output, which I have truncated:

```
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
 Log: awslogs fluentd gcplogs gelf journald json-file logentries splunk
syslog
Swarm: active
 NodeID: uxgvqhw6npr9glhp0zpabn4ha
 Is Manager: true
 ClusterID: pavj3f2ym8u1u1ul5epr3c73f
 Managers: 1
 Nodes: 3
 Orchestration:
  Task History Retention Limit: 5
 Raft:
  Snapshot Interval: 10000
  Number of Old Snapshots to Retain: 0
  Heartbeat Tick: 1
  Election Tick: 10
 Dispatcher:
  Heartbeat Period: 5 seconds
 CA Configuration:
  Expiry Duration: 3 months
  Force Rotate: 0
 Autolock Managers: false
 Root Rotation In Progress: false
 Node Address: 192.168.99.100
 Manager Addresses:
  192.168.99.100:2377
Runtimes: runc
Default Runtime: runc
Init Binary: docker-init
```

```
containerd version: 468a545b9edcd5932818eb9de8e72413e616e86e
runc version: 69663f0bd4b60df09991c08812a60108003fa340
init version: fec3683
Kernel Version: 4.9.93-boot2docker
Operating System: Boot2Docker 18.06.1-ce (TCL 8.2.1); HEAD : c7e5c3e - Wed
Aug 22 16:27:42 UTC 2018
OSType: linux
Architecture: x86_64
CPUs: 1
Total Memory: 995.6MiB
Name: swarm-manager
ID: NRV7:WAFE:FWDS:63PT:UMZY:G3KU:OU2A:RWRN:RC7D:5ESI:NWRN:NZRU
```

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, such as the following, for example:

```
$ docker node inspect swarm-manager --pretty
```

Assessing the `--pretty` flag with the `docker node inspect` command will render the output in the easy-to-read format you see as follows. 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.

This should provide the following information on our Swarm manager:

```
ID: uxgvqhw6npr9glhp0zpabn4ha
Hostname: swarm-manager
Joined at: 2018-09-15 12:14:59.663920111 +0000 utc
Status:
 State: Ready
 Availability: Active
 Address: 192.168.99.100
Manager Status:
 Address: 192.168.99.100:2377
 Raft Status: Reachable
 Leader: Yes
Platform:
 Operating System: linux
 Architecture: x86_64
Resources:
 CPUs: 1
 Memory: 995.6MiB
Plugins:
 Log: awslogs, fluentd, gcplogs, gelf, journald, json-file, logentries,
splunk, syslog
 Network: bridge, host, macvlan, null, overlay
 Volume: local
Engine Version: 18.06.1-ce
Engine Labels:
 - provider=virtualbox
```

Run the same command, but this time targeting one of the worker nodes:

```
$ docker node inspect swarm-worker01 --pretty
```

This gives us similar information:

```
ID: yhqj03rkfzurb4aqzk7duidf4
Hostname: swarm-worker01
Joined at: 2018-09-15 12:24:09.02346782 +0000 utc
Status:
 State: Ready
 Availability: Active
 Address: 192.168.99.101
Platform:
 Operating System: linux
 Architecture: x86_64
Resources:
 CPUs: 1
 Memory: 995.6MiB
Plugins:
 Log: awslogs, fluentd, gcplogs, gelf, journald, json-file, logentries,
splunk, syslog
 Network: bridge, host, macvlan, null, overlay
 Volume: local
Engine Version: 18.06.1-ce
Engine Labels:
 - provider=virtualbox
```

But as you can see, it is missing the information about the state of the manager functionality. 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 the information about this host, such as the number of containers, the numbers of images on the host, and information about the CPU and memory, along with other interesting information.

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 `swarm-worker01` to be a new manager. To do this, run the following command:

```
$ docker node promote swarm-worker01
```

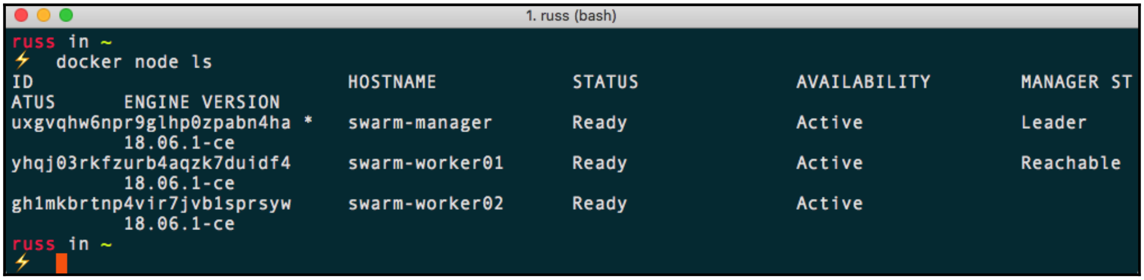
You should receive a message confirming that your node has been promoted immediately after executing the command:

```
Node swarm-worker01 promoted to a manager in the swarm.
```

List the nodes by running this:

```
$ docker node ls
```

This should show you that you now have two nodes that display something in the `MANAGER STATUS` column:



```
russ in ~
⚡  docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER ST
ATUS      ENGINE VERSION
uxgvqhw6npr9glhp0zpabn4ha *   swarm-manager       Ready               Active              Leader
          18.06.1-ce
yhqj03rkfzurb4aqzk7duidf4     swarm-worker01      Ready               Active              Reachable
          18.06.1-ce
gh1mkbrtnp4vir7jvb1sprsyw     swarm-worker02      Ready               Active
          18.06.1-ce
russ in ~
⚡  
```

Our `swarm-manager` node is still the primary manager node though. Let's look at doing something about that.

Demoting a manager node

You may have already put two and two together, but to `demote` a manager node to a worker, you simply need to run this command:

```
$ docker node demote swarm-manager
```

Again, you will receive immediate feedback stating the following:

```
Manager swarm-manager 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 your local Docker client is still pointing toward 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.
```

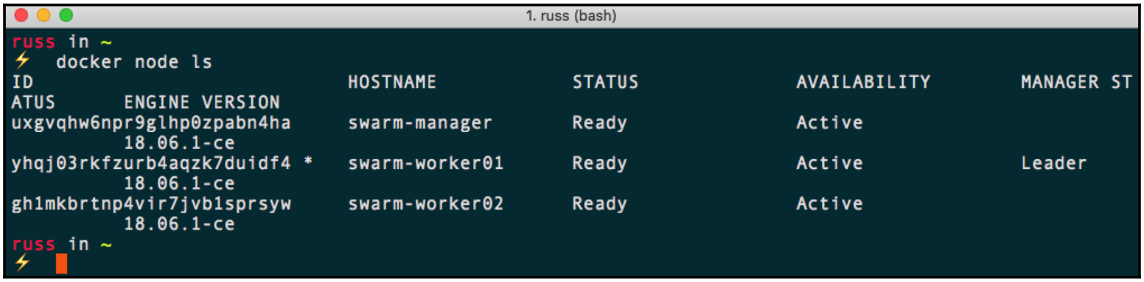
As we have already learned, it is easy to update our local client configuration to communicate with other nodes using Docker Machine. To point your local client to the new manager node, run the following command:

```
$ eval $(docker-machine env swarm-worker01)
```

Now that out client is talking to a manager node again, rerun this:

```
$ docker node ls
```

It should list the nodes, as expected:



```
russ in ~
⚡  docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER ST
ATUS      ENGINE VERSION
uxgvqhw6npr9glhp0zpabn4ha     swarm-manager       Ready               Active
          18.06.1-ce
yhqj03rkfzurb4aqzk7duidf4 *   swarm-worker01      Ready               Active              Leader
          18.06.1-ce
gh1mkbrtnp4vir7jvb1sprsyw     swarm-worker02      Ready               Active
          18.06.1-ce
russ in ~
⚡
```

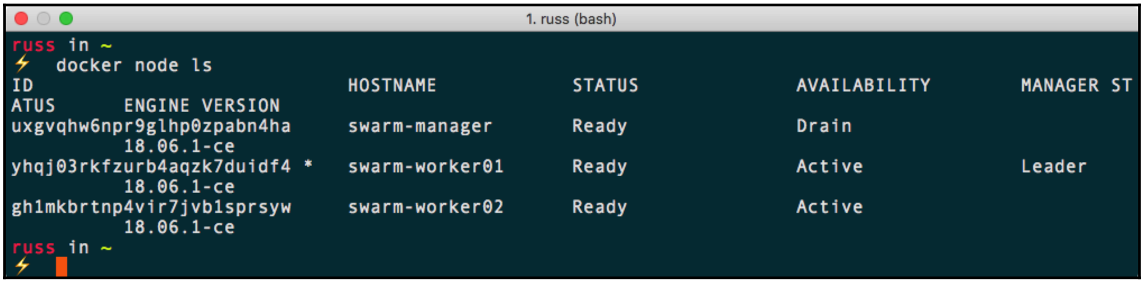
Draining a node

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 swarm-manager
```

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.

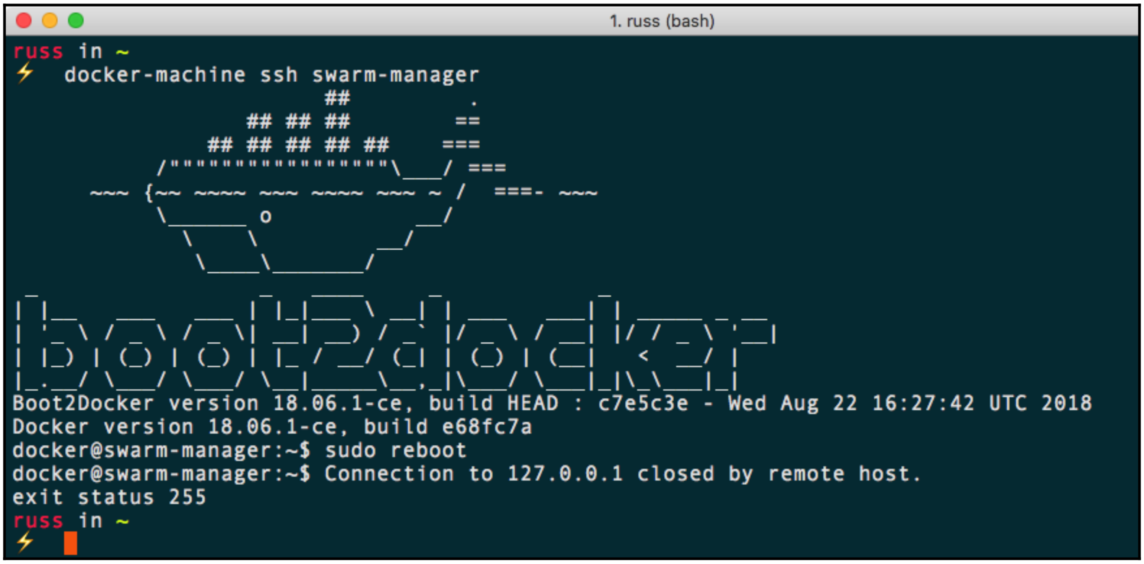
As you can see from the following Terminal output, listing the nodes now shows that `swarm-manager` node is listed as `Drain` in the `AVAILABILITY` column:



```
russ in ~
⚡  docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER ST
ATUS      ENGINE VERSION
uxgvqhw6npr9glhp0zpabn4ha     swarm-manager       Ready               Drain
          18.06.1-ce
yhqj03rkfzurb4aqzk7duidf4 *   swarm-worker01      Ready               Active              Leader
          18.06.1-ce
gh1mkbrtnp4vir7jvb1sprsyw     swarm-worker02      Ready               Active
          18.06.1-ce
russ in ~
⚡
```

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 Swarm manager, run the following two commands, ensuring that you are connected to the Docker host (you should see the `boot2docker` banner, like in the screenshot following the commands):

```
$ docker-machine ssh swarm-manager
$ sudo reboot
```



```
russ in ~
⚡  docker-machine ssh swarm-manager
                        ##         .
                  ## ## ##        ==
               ## ## ## ## ##    ===
           /"""""""""""""""""\___/ ===
      ~~~ {~~ ~~~~ ~~~ ~~~~ ~~~ ~ /  ===- ~~~
           \______ o           __/
             \    \         __/
              \____\_______/
 _                 _   ____     _            _
| |__   ___   ___ | |_|___ \ __| | ___   ___| | _____ _ __
| '_ \ / _ \ / _ \| __| __) / _` |/ _ \ / __| |/ / _ \ '__|
| |_) | (_) | (_) | |_ / __/ (_| | (_) | (__|   <  __/ |
|_.__/ \___/ \___/ \__|_____\__,_|\___/ \___|_|\_\___|_|
Boot2Docker version 18.06.1-ce, build HEAD : c7e5c3e - Wed Aug 22 16:27:42 UTC 2018
Docker version 18.06.1-ce, build e68fc7a
docker@swarm-manager:~$ sudo reboot
docker@swarm-manager:~$ Connection to 127.0.0.1 closed by remote host.
exit status 255
russ in ~
⚡
```

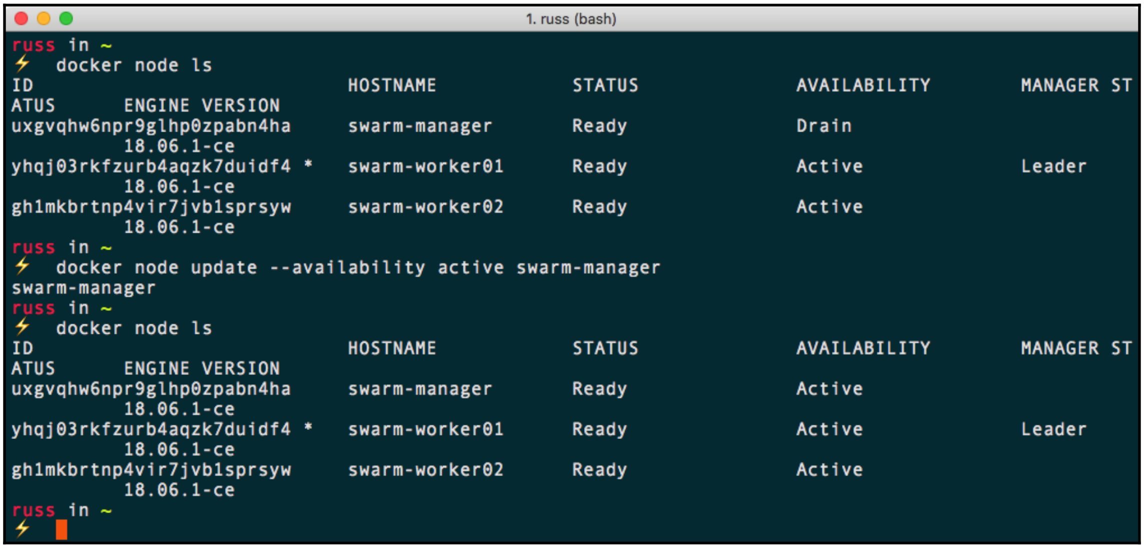
Once the host has been rebooted, run this command:

```
$ docker node ls
```

It should show that the node has an `AVAILABILITY` of `Drain`. To add the node back into the cluster, simply change the `AVAILABILITY` to active by running this:

```
$ docker node update --availability active swarm-manager
```

As you can see from the following Terminal output, our node is now active, meaning new tasks can be executed against it:



1. russ (bash)

```
russ in ~
⚡  docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER ST
ATUS      ENGINE VERSION
uxgvqhw6npr9glhp0zpabn4ha     swarm-manager       Ready               Drain
          18.06.1-ce
yhqj03rkfzurb4aqzk7duidf4 *   swarm-worker01      Ready               Active              Leader
          18.06.1-ce
gh1mkbrtnp4vir7jvb1sprsyw     swarm-worker02      Ready               Active
          18.06.1-ce
russ in ~
⚡  docker node update --availability active swarm-manager
swarm-manager
russ in ~
⚡  docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER ST
ATUS      ENGINE VERSION
uxgvqhw6npr9glhp0zpabn4ha     swarm-manager       Ready               Active
          18.06.1-ce
yhqj03rkfzurb4aqzk7duidf4 *   swarm-worker01      Ready               Active              Leader
          18.06.1-ce
gh1mkbrtnp4vir7jvb1sprsyw     swarm-worker02      Ready               Active
          18.06.1-ce
russ in ~
⚡
```

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.

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. 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.

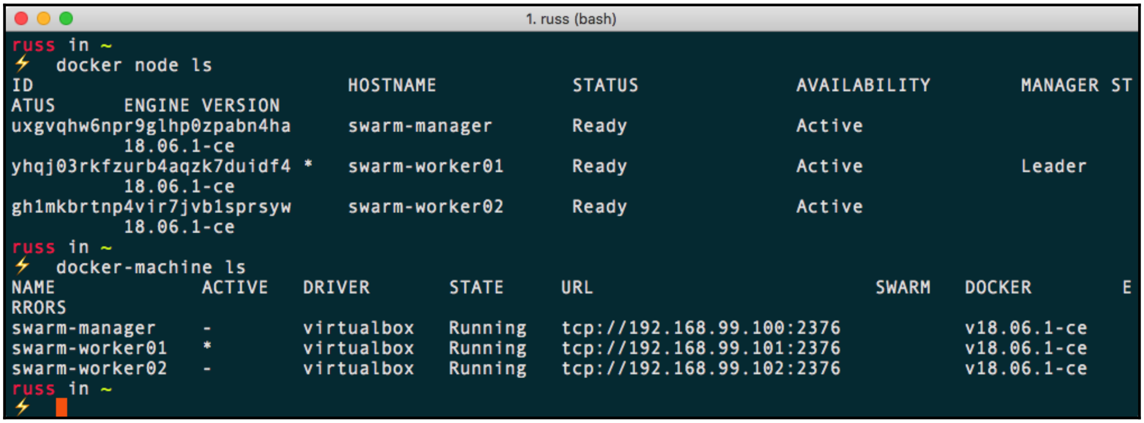
Before we look at doing more with the service, we can check whether it worked on our browser. To do this, we will need the IP address of our two worker nodes. First of all, we need to double check which are the worker nodes by running this command:

```
$ docker node ls
```

Once we know which node has which role, you can find the IP addresses of your nodes by running this command:

```
$ docker-machine ls
```

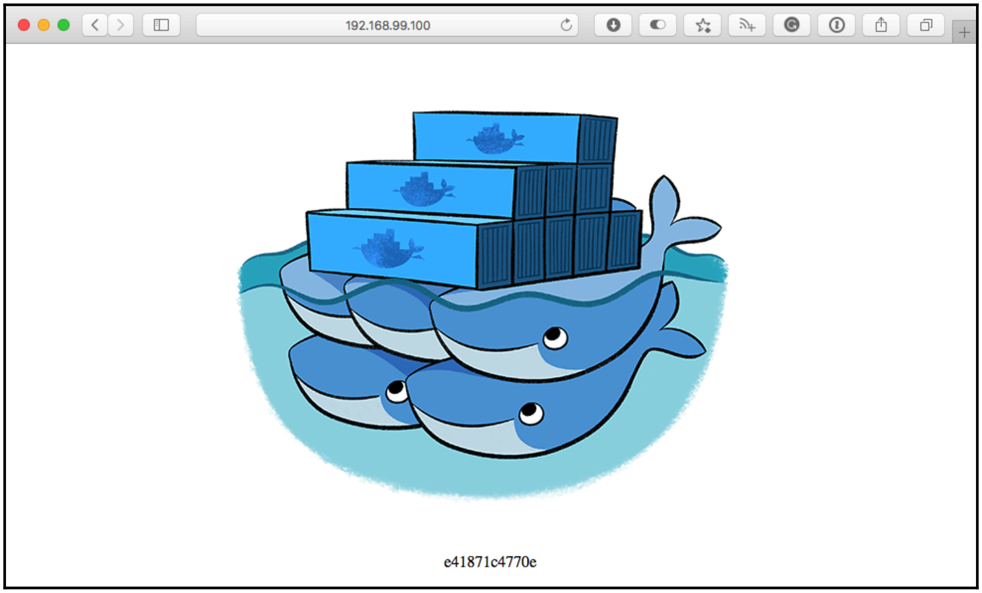
Look at the following Terminal output:



```
russ in ~
⚡  docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER ST
ATUS      ENGINE VERSION
uxgvqhw6npr9glhp0zpabn4ha     swarm-manager       Ready               Active
          18.06.1-ce
yhqj03rkfzurb4aqzk7duidf4 *   swarm-worker01      Ready               Active              Leader
          18.06.1-ce
gh1mkbrtnp4vir7jvb1sprsyw     swarm-worker02      Ready               Active
          18.06.1-ce
russ in ~
⚡  docker-machine ls
NAME            ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER        E
RRORS
swarm-manager   -        virtualbox   Running   tcp://192.168.99.100:2376           v18.06.1-ce
swarm-worker01  *        virtualbox   Running   tcp://192.168.99.101:2376           v18.06.1-ce
swarm-worker02  -        virtualbox   Running   tcp://192.168.99.102:2376           v18.06.1-ce
russ in ~
⚡
```

My worker nodes are `swarm-manager` and `swarm-worker02`, whose IP addresses are `192.168.99.100` and `192.168.99.102` respectively.

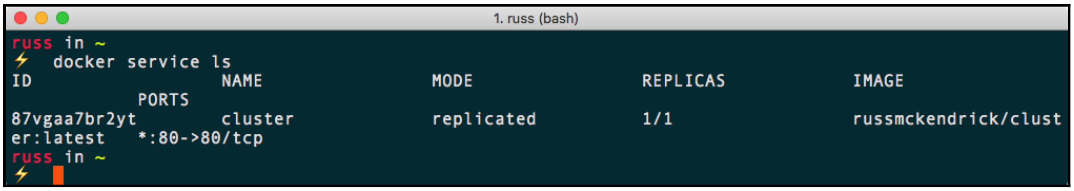
Going to either of the IP addresses of your worker nodes, such as `http://192.168.99.100/` or `http://192.168.99.102/`, in a browser will show the output of the `russmckendrick/cluster` application, which is the Docker Swarm graphic and the hostname of the container the page is being served from:



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:

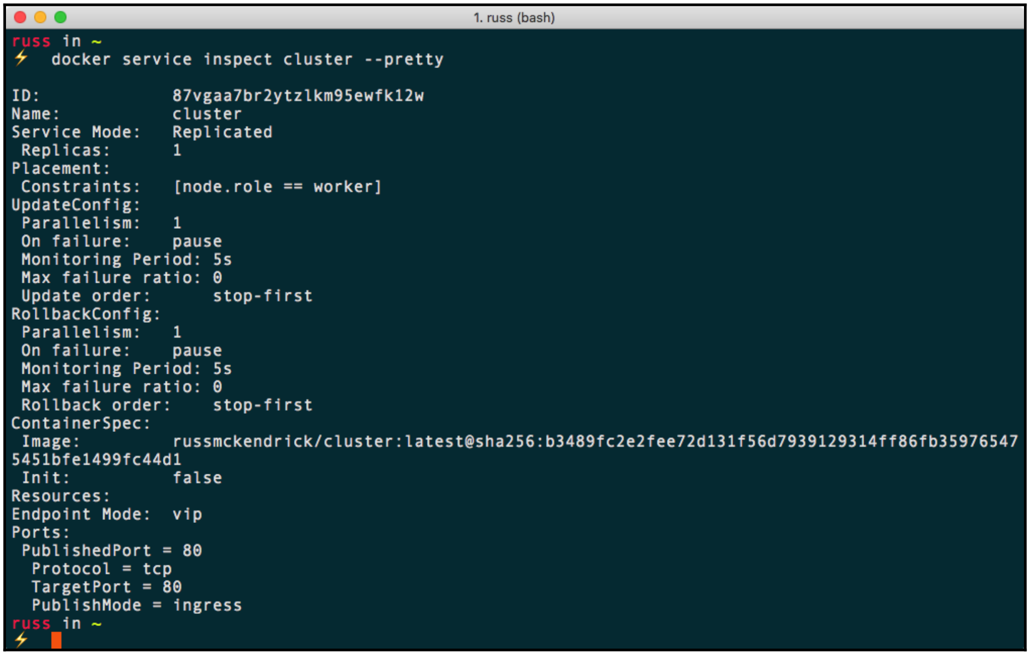


```
russ in ~
⚡  docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE
            PORTS
87vgaa7br2yt        cluster             replicated          1/1                 russmckendrick/clust
er:latest   *:80->80/tcp
russ in ~
⚡
```

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:

```
$ docker service inspect cluster --pretty
```

This will return detailed information about the service:



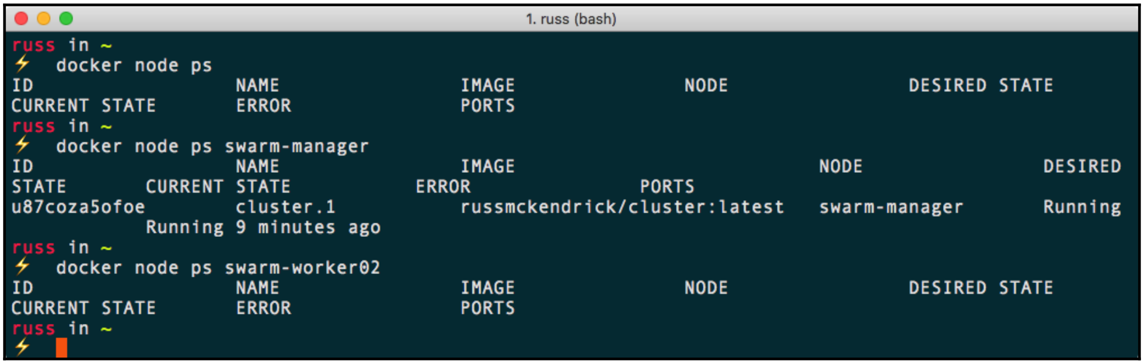
```
russ in ~
⚡  docker service inspect cluster --pretty

ID:             87vgaa7br2ytzlkm95ewfk12w
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:b3489fc2e2fee72d131f56d7939129314ff86fb35976547
5451bfe1499fc44d1
 Init:          false
Resources:
Endpoint Mode:  vip
Ports:
 PublishedPort = 80
  Protocol = tcp
  TargetPort = 80
  PublishMode = ingress
russ in ~
⚡
```

You may have noticed that so far, we haven't had to care 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 running these commands:

```
$ docker node ps
$ docker node ps swarm-manager
$ docker node ps swarm-worker02
```

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 `swarm-worker01`:

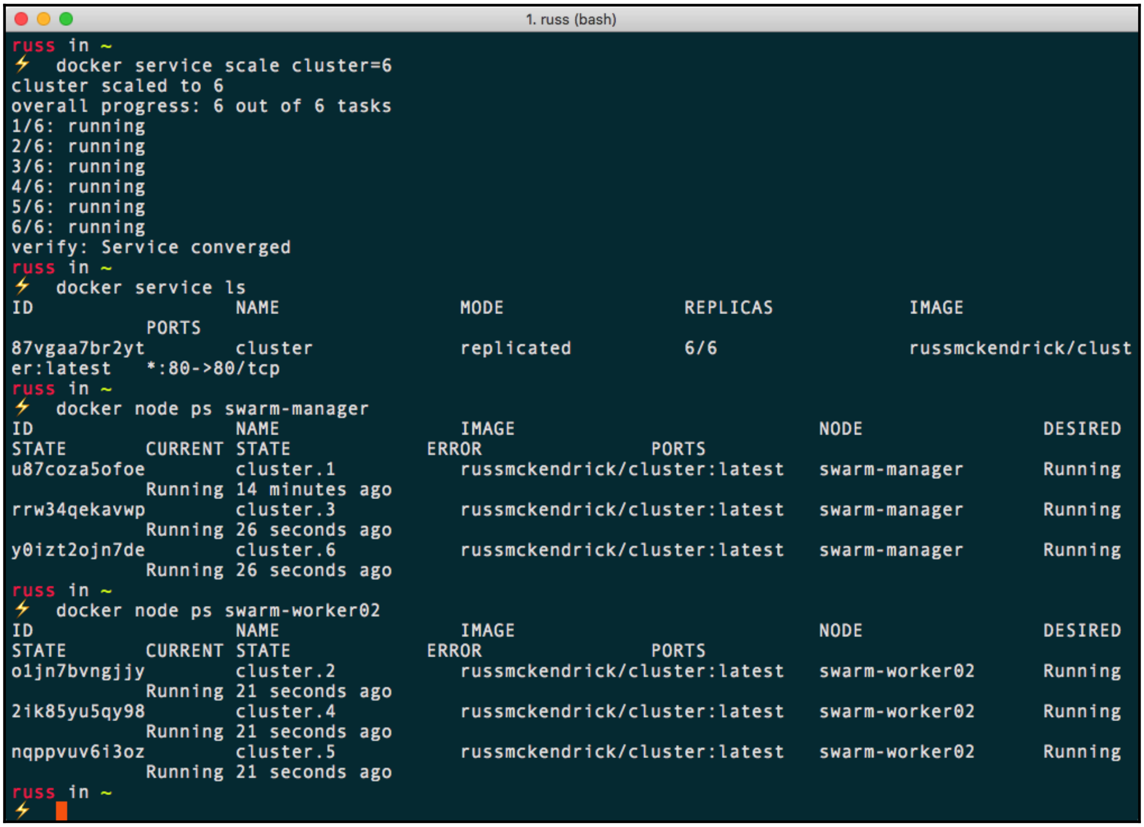


```
russ in ~
⚡  docker node ps
ID                  NAME                IMAGE               NODE                DESIRED STATE
CURRENT STATE       ERROR               PORTS
russ in ~
⚡  docker node ps swarm-manager
ID                  NAME                IMAGE                           NODE                DESIRED
STATE       CURRENT STATE           ERROR               PORTS
u87coza5ofoe        cluster.1           russmckendrick/cluster:latest   swarm-manager       Running
            Running 9 minutes ago
russ in ~
⚡  docker node ps swarm-worker02
ID                  NAME                IMAGE               NODE                DESIRED STATE
CURRENT STATE       ERROR               PORTS
russ in ~
⚡
```

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 swarm-manager
$ docker node ps swarm-worker02
```

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:



```
russ in ~
⚡  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
russ in ~
⚡  docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE
            PORTS
87vgaa7br2yt        cluster             replicated          6/6                 russmckendrick/clust
er:latest   *:80->80/tcp
russ in ~
⚡  docker node ps swarm-manager
ID                  NAME                IMAGE                           NODE                DESIRED
STATE       CURRENT STATE            ERROR               PORTS
u87coza5ofoe        cluster.1           russmckendrick/cluster:latest   swarm-manager       Running
            Running 14 minutes ago
rrw34qekavwp        cluster.3           russmckendrick/cluster:latest   swarm-manager       Running
            Running 26 seconds ago
y0izt2ojn7de        cluster.6           russmckendrick/cluster:latest   swarm-manager       Running
            Running 26 seconds ago
russ in ~
⚡  docker node ps swarm-worker02
ID                  NAME                IMAGE                           NODE                DESIRED
STATE       CURRENT STATE            ERROR               PORTS
o1jn7bvngjjy        cluster.2           russmckendrick/cluster:latest   swarm-worker02      Running
            Running 21 seconds ago
2ik85yu5qy98        cluster.4           russmckendrick/cluster:latest   swarm-worker02      Running
            Running 21 seconds ago
nqppvuv6i3oz        cluster.5           russmckendrick/cluster:latest   swarm-worker02      Running
            Running 21 seconds ago
russ in ~
⚡
```

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 and 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 difficult to share. To this end, Docker has created functionality that allows you to define your services in Docker Compose files.

The following Docker Compose file, which should be named `docker-compose.yml`, will create the same service we launched in the previous 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 a file called `docker-compose.yml`, 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
```

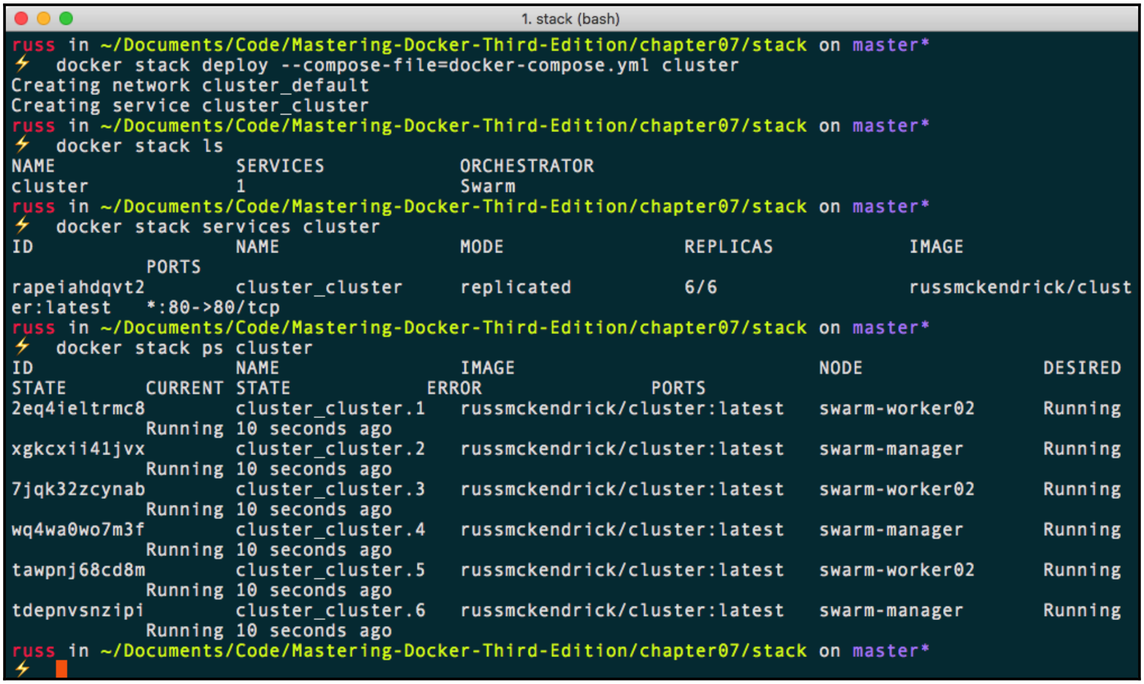
This will show that a single service has been created. You can get details of the service created by the `stack` by running this command:

```
$ 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 Terminal output:



```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter07/stack on master*
⚡  docker stack deploy --compose-file=docker-compose.yml cluster
Creating network cluster_default
Creating service cluster_cluster
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter07/stack on master*
⚡  docker stack ls
NAME                SERVICES            ORCHESTRATOR
cluster             1                   Swarm
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter07/stack on master*
⚡  docker stack services cluster
ID                  NAME                MODE                REPLICAS            IMAGE
            PORTS
rapeiahdqvt2        cluster_cluster     replicated          6/6                 russmckendrick/clust
er:latest   *:80->80/tcp
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter07/stack on master*
⚡  docker stack ps cluster
ID                  NAME                IMAGE                           NODE                DESIRED
STATE       CURRENT STATE       ERROR               PORTS
2eq4ieltrmc8        cluster_cluster.1   russmckendrick/cluster:latest   swarm-worker02      Running
            Running 10 seconds ago
xgkcxii41jvx        cluster_cluster.2   russmckendrick/cluster:latest   swarm-manager       Running
            Running 10 seconds ago
7jqk32zcynab        cluster_cluster.3   russmckendrick/cluster:latest   swarm-worker02      Running
            Running 10 seconds ago
wq4wa0wo7m3f        cluster_cluster.4   russmckendrick/cluster:latest   swarm-manager       Running
            Running 10 seconds ago
tawpnj68cd8m        cluster_cluster.5   russmckendrick/cluster:latest   swarm-worker02      Running
            Running 10 seconds ago
tdepnvsnzipi        cluster_cluster.6   russmckendrick/cluster:latest   swarm-manager       Running
            Running 10 seconds ago
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter07/stack on master*
⚡
```

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:

```
$ docker-machine rm swarm-manager swarm-worker01 swarm-worker02
```

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.

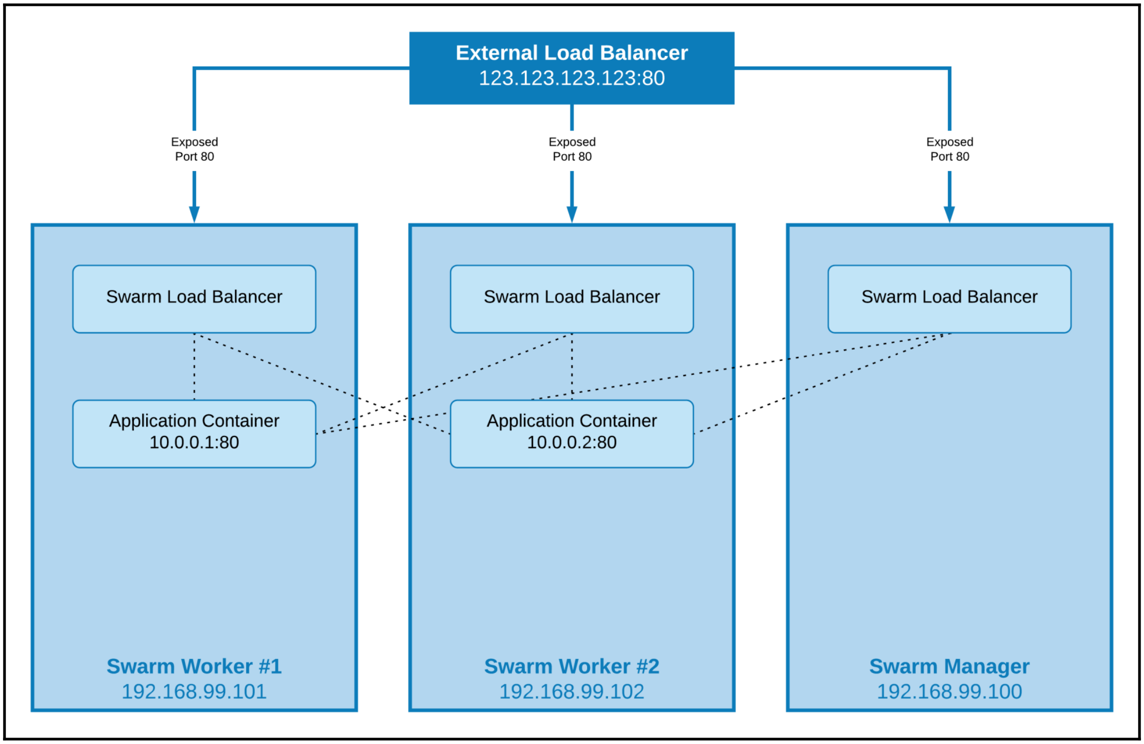
Load balancing, overlays, and scheduling

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, 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:



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.

Network overlays

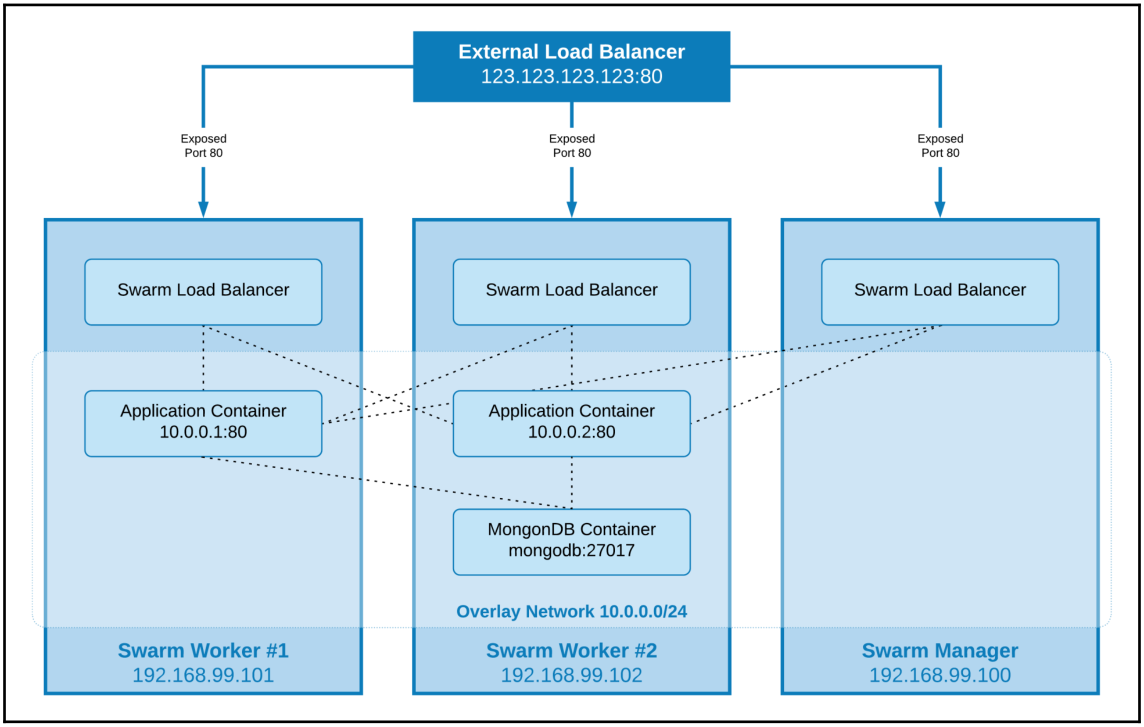
In our example, we launched a simple service running a single application. Say we wanted to add a database layer in our application, which is typically a fixed point within the network; how could we do this?

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 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:



There are some other considerations you will need to take into account when adopting this pattern, but we will cover those in `Chapter 14`, *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 to 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 this using constraints, I urge you not to over complicate 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.

In the next chapter, we are going to look at an alternative to Docker Swarm called Kubernetes. This is also supported by Docker as well as other providers.

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. What 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 Swarm manager to make it more readable?
5. How do you promote a node to be a manager?
6. What command can you use to scale your service?

Further reading

For a detailed explanation of the Raft consensus algorithms, 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
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.

The following topics will be covered in the chapter:

- An introduction to Kubernetes
- Enabling Kubernetes
- Using Kubernetes
- Kubernetes and other Docker tools

Technical requirements

Kubernetes within Docker is only supported by Docker for Mac and Docker for Windows desktop clients. 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:
`http://bit.ly/2q6xpwl`

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 where Kubernetes came from.

Kubernetes (which is pronounced **koo-ber-net-eez**) originates from the Greek name given to a helmsman or captain of a ship. **Kubernetes** (which is also known as **K8s**), an open source project, that originated at Google, 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 disc 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 toe into container waters with a container stack called **lmctfy,** which stands for **Let Me Contain That For You**. This was an alternative to the **LXC** collection of tools and libraries. It was an open sourced version of their own internal tools, which they used to manage containers in their own 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 talk, 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, this meant that 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 who's civilization is based 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 of 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. Its aim was to create a friendlier version of Borg which wasn't closely tied into Google's own internal procedures and ways of working.

Seven, which was 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.

An overview of Kubernetes

So, now we now know how Kubernetes came to be, we can dig a little deeper into what Kubernetes is. The bulk of the project, 88.5% to be precise, 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 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:

- `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 components:

- `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 RKT 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.

Kubernetes and Docker

Kubernetes was originally seen as a competitive technology to Docker Swarm, Docker's own clustering technology. However, over the last few years, Kubernetes has emerged as pretty much the de facto standard for container 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

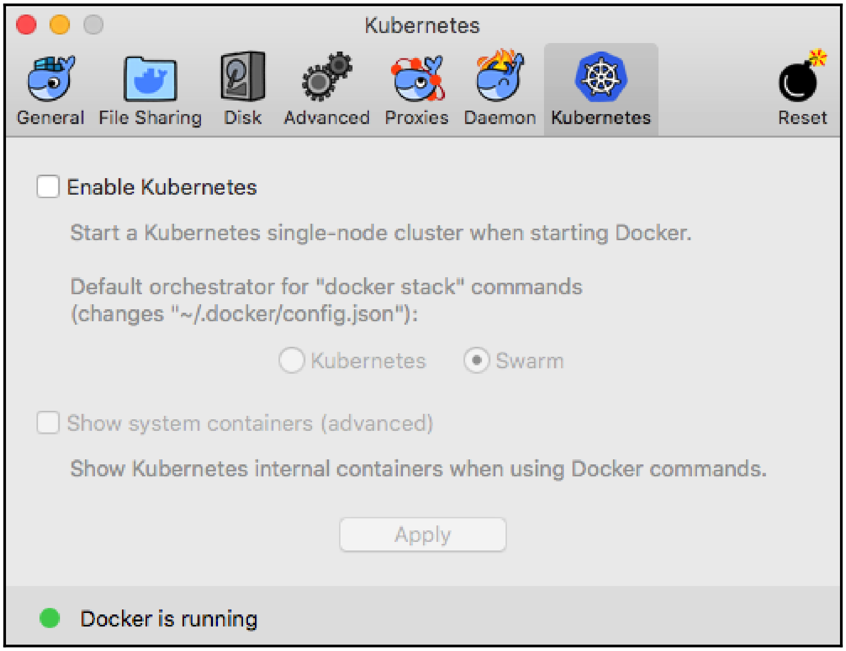
On the face of it, all of the major players supporting Kubernetes may not seem like that big a deal. However, consider that we now know a consistent way of deploying our containerized applications across multiple platforms. Traditionally, these platforms have been walled gardens and have very different ways of interacting with them.

While Docker's announcement 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 the 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 with using it.

Enabling Kubernetes

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:



□ Enable Kubernetes

Start a Kubernetes single-node cluster when starting Docker.

Default orchestrator for "docker stack" commands
(changes "~/.docker/config.json"):

○ Kubernetes ● Swarm

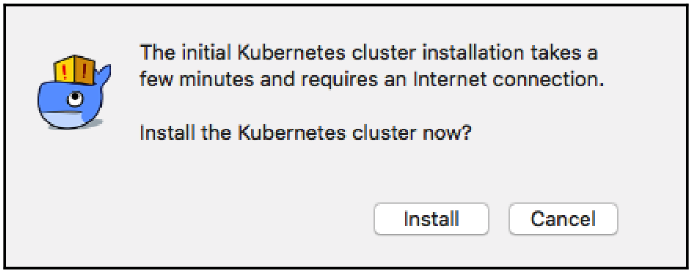
□ Show system containers (advanced)

Show Kubernetes internal containers when using Docker commands.

Apply

● Docker is running

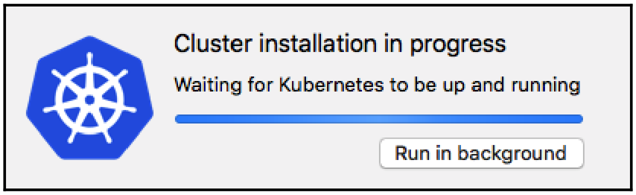
As you can see, there are two main options. Tick the **Enable Kubernetes** box and then select **Kubernetes** as the default orchestrator. Leave **Show systems containers** unticked for now; we look at this in a little more detail once we have enabled the service.
Clicking **Apply** will pop up the following message:



The initial Kubernetes cluster installation takes a few minutes and requires an Internet connection.

Install the Kubernetes cluster now?

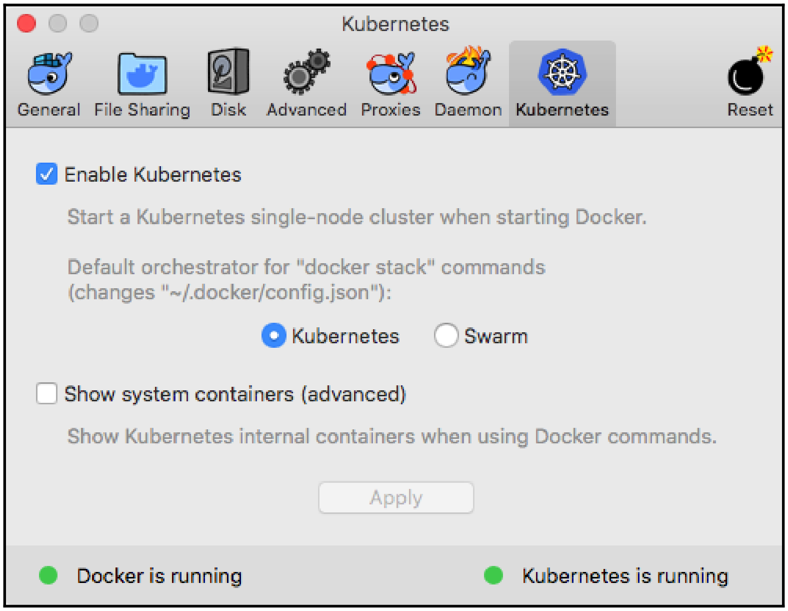
Hitting the **Install** button will download the required containers needed to enable Kubernetes support on your Docker installation:



Cluster installation in progress

Waiting for Kubernetes to be up and running

As mentioned in the first dialogue box, it will take a short while for Docker to download, configure, and launch the cluster. Once complete, you should see a green dot next to **Kubernetes is running**:



☑ Enable Kubernetes

Start a Kubernetes single-node cluster when starting Docker.

Default orchestrator for "docker stack" commands (changes "~/.docker/config.json"):

● Kubernetes ○ Swarm

□ Show system containers (advanced)

Show Kubernetes internal containers when using Docker commands.

● Docker is running ● Kubernetes is running

Open a 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
```

This should show you a list of Kubernetes-related images:

- `docker/kube-compose-controller`
- `docker/kube-compose-api-server`
- `k8s.gcr.io/kube-proxy-amd64`
- `k8s.gcr.io/kube-scheduler-amd64`
- `k8s.gcr.io/kube-apiserver-amd64`
- `k8s.gcr.io/kube-controller-manager-amd64`
- `k8s.gcr.io/etcd-amd64`
- `k8s.gcr.io/k8s-dns-dnsmasq-nanny-amd64`
- `k8s.gcr.io/k8s-dns-sidecar-amd64`
- `k8s.gcr.io/k8s-dns-kube-dns-amd64`
- `k8s.gcr.io/pause-amd64`

The images are sourced from both Docker and also the official Kubernetes images that are available from the Google Container Registry (`k8s.gcr.io`).

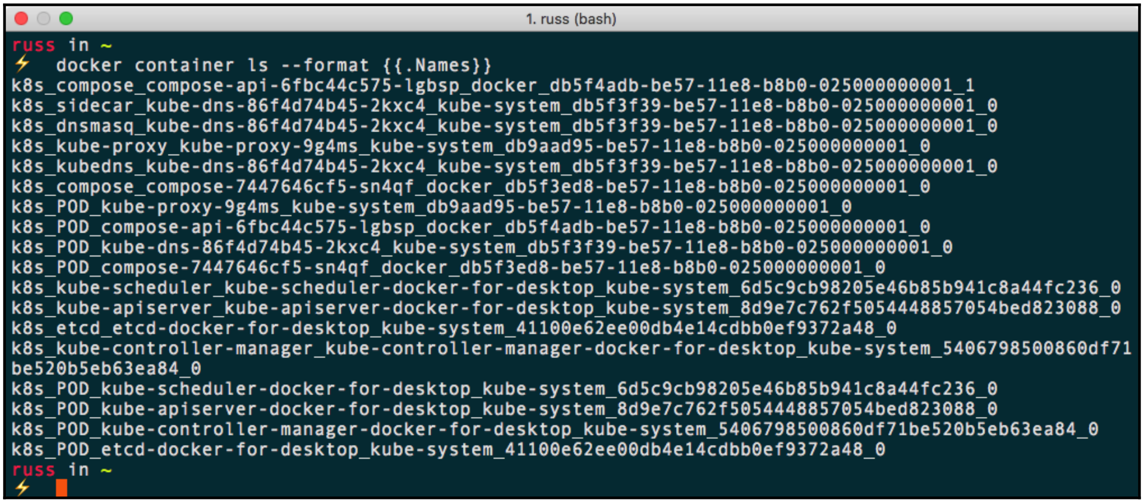
As you may have already guessed, ticking the **Show system containers (advanced)** box and then running the following command will show you a list of all of the containers running that enable the Kubernetes service on your local Docker installation:

```
$ 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}}
```

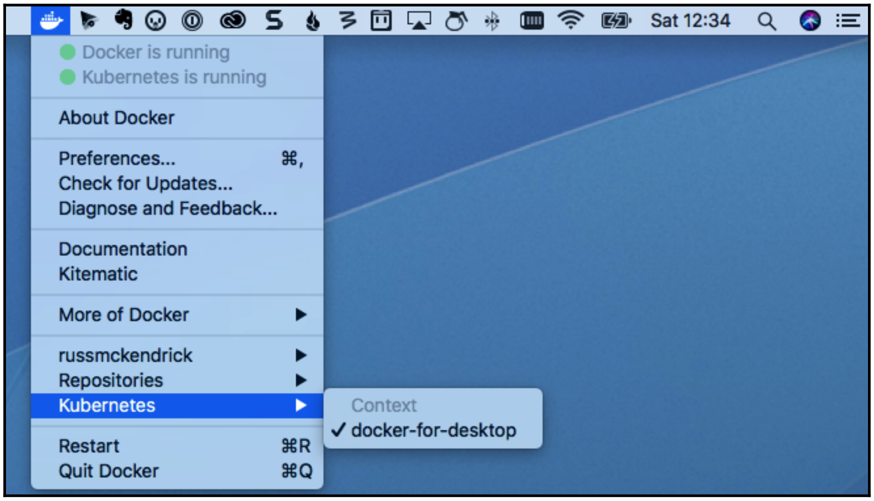
Running the command gave me the following:



```
1. russ (bash)
russ in ~
⚡  docker container ls --format {{.Names}}
k8s_compose_compose-api-6fbc44c575-lgbsp_docker_db5f4adb-be57-11e8-b8b0-025000000001_1
k8s_sidecar_kube-dns-86f4d74b45-2kxc4_kube-system_db5f3f39-be57-11e8-b8b0-025000000001_0
k8s_dnsmasq_kube-dns-86f4d74b45-2kxc4_kube-system_db5f3f39-be57-11e8-b8b0-025000000001_0
k8s_kube-proxy_kube-proxy-9g4ms_kube-system_db9aad95-be57-11e8-b8b0-025000000001_0
k8s_kubedns_kube-dns-86f4d74b45-2kxc4_kube-system_db5f3f39-be57-11e8-b8b0-025000000001_0
k8s_compose_compose-7447646cf5-sn4qf_docker_db5f3ed8-be57-11e8-b8b0-025000000001_0
k8s_POD_kube-proxy-9g4ms_kube-system_db9aad95-be57-11e8-b8b0-025000000001_0
k8s_POD_compose-api-6fbc44c575-lgbsp_docker_db5f4adb-be57-11e8-b8b0-025000000001_0
k8s_POD_kube-dns-86f4d74b45-2kxc4_kube-system_db5f3f39-be57-11e8-b8b0-025000000001_0
k8s_POD_compose-7447646cf5-sn4qf_docker_db5f3ed8-be57-11e8-b8b0-025000000001_0
k8s_kube-scheduler_kube-scheduler-docker-for-desktop_kube-system_6d5c9cb98205e46b85b941c8a44fc236_0
k8s_kube-apiserver_kube-apiserver-docker-for-desktop_kube-system_8d9e7c762f5054448857054bed823088_0
k8s_etcd_etcd-docker-for-desktop_kube-system_41100e62ee00db4e14cdbb0ef9372a48_0
k8s_kube-controller-manager_kube-controller-manager-docker-for-desktop_kube-system_5406798500860df71
be520b5eb63ea84_0
k8s_POD_kube-scheduler-docker-for-desktop_kube-system_6d5c9cb98205e46b85b941c8a44fc236_0
k8s_POD_kube-apiserver-docker-for-desktop_kube-system_8d9e7c762f5054448857054bed823088_0
k8s_POD_kube-controller-manager-docker-for-desktop_kube-system_5406798500860df71be520b5eb63ea84_0
k8s_POD_etcd-docker-for-desktop_kube-system_41100e62ee00db4e14cdbb0ef9372a48_0
russ in ~
⚡
```

There are 18 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. I would recommend unticking the **Show system containers box**, as we do not need to see a list of 18 containers running each time we look at the running containers.

The other thing to note at this point is that the Kubernetes menu item 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:



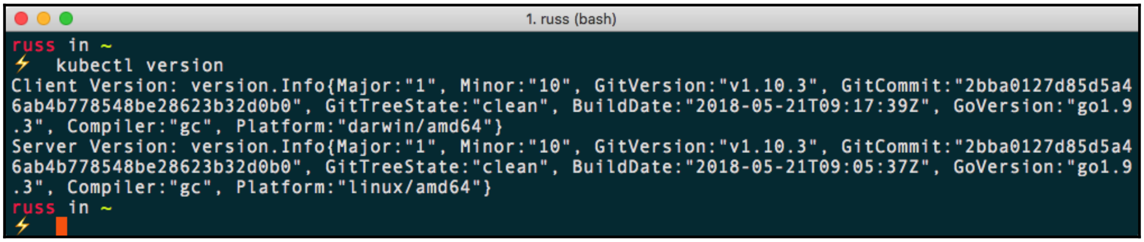
Now that we have our local Kubernetes cluster up and running, we can start to use it.

Using Kubernetes

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. The following command will show some information about the client and also the cluster it is connected to:

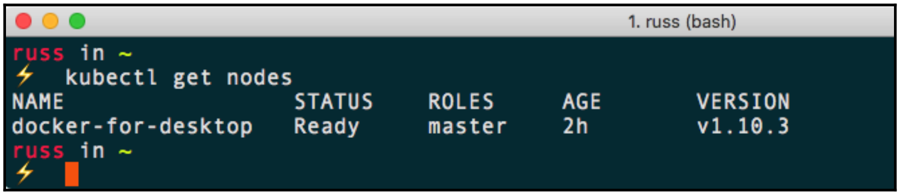
```
$ kubectl version
```



```
russ in ~
⚡  kubectl version
Client Version: version.Info{Major:"1", Minor:"10", GitVersion:"v1.10.3", GitCommit:"2bba0127d85d5a4
6ab4b778548be28623b32d0b0", GitTreeState:"clean", BuildDate:"2018-05-21T09:17:39Z", GoVersion:"go1.9
.3", Compiler:"gc", Platform:"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"10", GitVersion:"v1.10.3", GitCommit:"2bba0127d85d5a4
6ab4b778548be28623b32d0b0", GitTreeState:"clean", BuildDate:"2018-05-21T09:05:37Z", GoVersion:"go1.9
.3", Compiler:"gc", Platform:"linux/amd64"}
russ in ~
⚡
```

Next, we can run the following to see if `kubectl` can see our node:

```
$ kubectl get nodes
```



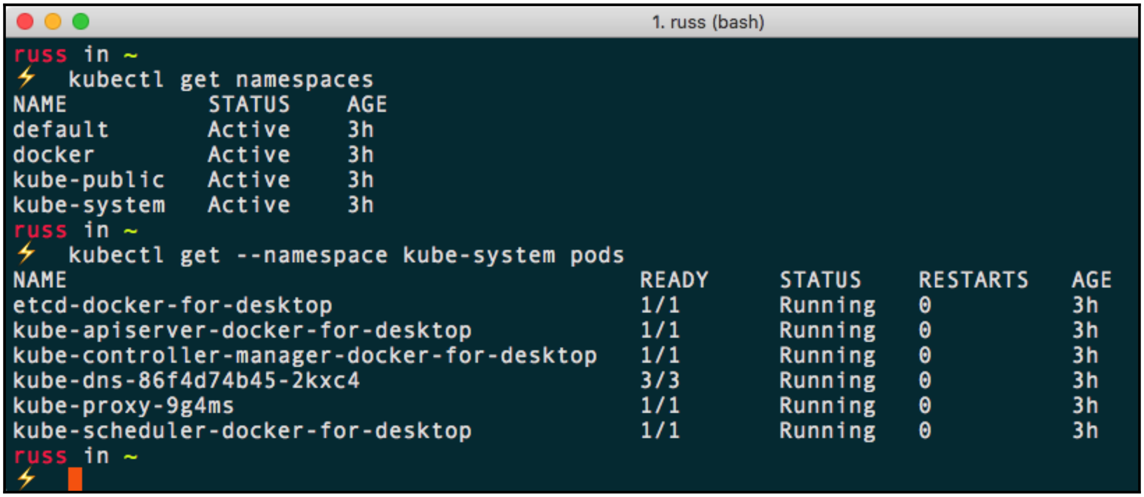
```
russ in ~
⚡  kubectl get nodes
NAME                 STATUS    ROLES     AGE       VERSION
docker-for-desktop   Ready     master    2h        v1.10.3
russ in ~
⚡
```

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 --namespace kube-system pods
```



```
russ in ~
⚡  kubectl get namespaces
NAME          STATUS    AGE
default       Active    3h
docker        Active    3h
kube-public   Active    3h
kube-system   Active    3h
russ in ~
⚡  kubectl get --namespace kube-system pods
NAME                                          READY     STATUS    RESTARTS   AGE
etcd-docker-for-desktop                       1/1       Running   0          3h
kube-apiserver-docker-for-desktop             1/1       Running   0          3h
kube-controller-manager-docker-for-desktop    1/1       Running   0          3h
kube-dns-86f4d74b45-2kxc4                     3/3       Running   0          3h
kube-proxy-9g4ms                              1/1       Running   0          3h
kube-scheduler-docker-for-desktop             1/1       Running   0          3h
russ in ~
⚡
```

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 --namespace kube-system pods kube-scheduler-docker-for-
desktop
```

The preceding command will print out details of the `kube-scheduler-docker-for-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-for-desktop` running.

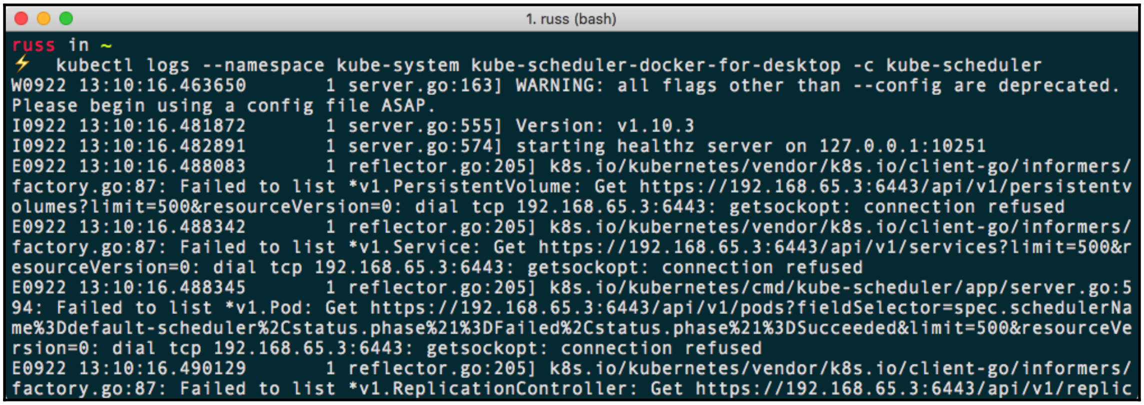
The full output of the command is shown here:

```
Name: kube-scheduler-docker-for-desktop
Namespace: kube-system
Node: docker-for-desktop/192.168.65.3
Start Time: Sat, 22 Sep 2018 14:10:14 +0100
Labels: component=kube-scheduler
 tier=control-plane
Annotations: kubernetes.io/config.hash=6d5c9cb98205e46b85b941c8a44fc236
 kubernetes.io/config.mirror=6d5c9cb98205e46b85b941c8a44fc236
 kubernetes.io/config.seen=2018-09-22T11:07:47.025395325Z
 kubernetes.io/config.source=file
 scheduler.alpha.kubernetes.io/critical-pod=
Status: Running
IP: 192.168.65.3
Containers:
 kube-scheduler:
 Container ID:
docker://7616b003b3c94ca6e7fd1bc3ec63f41fcb4b7ce845ef7a1fb8af1a2447e45859
 Image: k8s.gcr.io/kube-scheduler-amd64:v1.10.3
 Image ID: docker-pullable://k8s.gcr.io/kube-scheduler-
amd64@sha256:4770e1f1eef2229138e45a2b813c927e971da9c40256a7e2321ccf825af569
16
 Port: <none>
 Host Port: <none>
 Command:
 kube-scheduler
 --kubeconfig=/etc/kubernetes/scheduler.conf
 --address=127.0.0.1
 --leader-elect=true
 State: Running
 Started: Sat, 22 Sep 2018 14:10:16 +0100
 Ready: True
 Restart Count: 0
 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)
Conditions:
 Type Status
 Initialized True
 Ready True
 PodScheduled True
Volumes:
 kubeconfig:
 Type: HostPath (bare host directory volume)
 Path: /etc/kubernetes/scheduler.conf
 HostPathType: FileOrCreate
QoS Class: Burstable
Node-Selectors: <none>
Tolerations: :NoExecute
Events: <none>
```

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 --namespace kube-system kube-scheduler-docker-for-desktop -c
kube-scheduler
```



1. russ (bash)

```
russ in ~
⚡  kubectl logs --namespace kube-system kube-scheduler-docker-for-desktop -c kube-scheduler
W0922 13:10:16.463650       1 server.go:163] WARNING: all flags other than --config are deprecated.
Please begin using a config file ASAP.
I0922 13:10:16.481872       1 server.go:555] Version: v1.10.3
I0922 13:10:16.482891       1 server.go:574] starting healthz server on 127.0.0.1:10251
E0922 13:10:16.488083       1 reflector.go:205] k8s.io/kubernetes/vendor/k8s.io/client-go/informers/
factory.go:87: Failed to list *v1.PersistentVolume: Get https://192.168.65.3:6443/api/v1/persistentv
olumes?limit=500&resourceVersion=0: dial tcp 192.168.65.3:6443: getsockopt: connection refused
E0922 13:10:16.488342       1 reflector.go:205] k8s.io/kubernetes/vendor/k8s.io/client-go/informers/
factory.go:87: Failed to list *v1.Service: Get https://192.168.65.3:6443/api/v1/services?limit=500&r
esourceVersion=0: dial tcp 192.168.65.3:6443: getsockopt: connection refused
E0922 13:10:16.488345       1 reflector.go:205] k8s.io/kubernetes/cmd/kube-scheduler/app/server.go:5
94: Failed to list *v1.Pod: Get https://192.168.65.3:6443/api/v1/pods?fieldSelector=spec.schedulerNa
me%3Ddefault-scheduler%2Cstatus.phase%21%3DFailed%2Cstatus.phase%21%3DSucceeded&limit=500&resourceVe
rsion=0: dial tcp 192.168.65.3:6443: getsockopt: connection refused
E0922 13:10:16.490129       1 reflector.go:205] k8s.io/kubernetes/vendor/k8s.io/client-go/informers/
factory.go:87: Failed to list *v1.ReplicationController: Get https://192.168.65.3:6443/api/v1/replic
```

Running the following command would fetch the `logs` for each container in the pod:

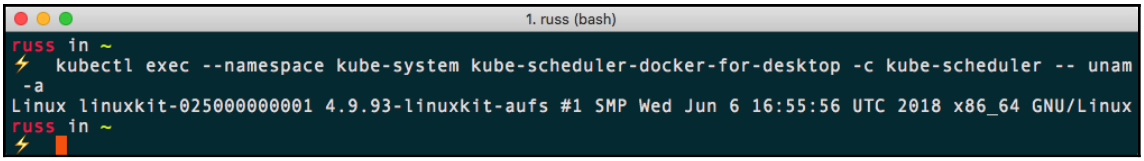
```
$ kubectl logs --namespace kube-system kube-scheduler-docker-for-desktop
```

Like Docker, you can also execute commands on your pods and containers. For example, the following commands will run the `uname -a` command:

Please ensure you add the space after the `--` in the following two commands. Failing to do so will result in errors.

```
$ kubectl exec --namespace kube-system kube-scheduler-docker-for-desktop -c
kube-scheduler -- uname -a
$ kubectl exec --namespace kube-system kube-scheduler-docker-for-desktop --
uname -a
```

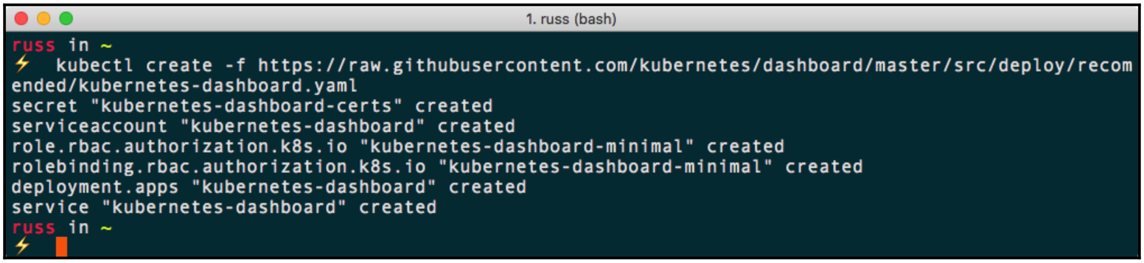
Again, we have the option of running the command on a named container or across all containers within the pod:



```
russ in ~
⚡  kubectl exec --namespace kube-system kube-scheduler-docker-for-desktop -c kube-scheduler -- unam
 -a
Linux linuxkit-025000000001 4.9.93-linuxkit-aufs #1 SMP Wed Jun 6 16:55:56 UTC 2018 x86_64 GNU/Linux
russ in ~
⚡
```

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 create -f
https://raw.githubusercontent.com/kubernetes/dashboard/master/src/deploy/re
commended/kubernetes-dashboard.yaml
```

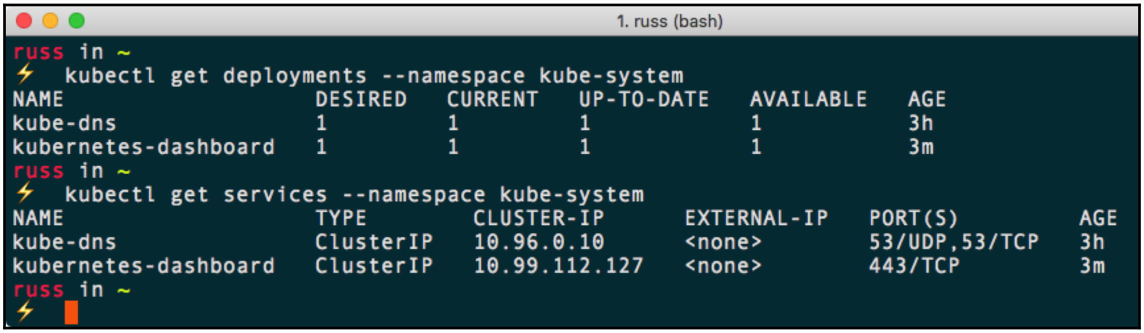


```
russ in ~
⚡  kubectl create -f https://raw.githubusercontent.com/kubernetes/dashboard/master/src/deploy/recom
ended/kubernetes-dashboard.yaml
secret "kubernetes-dashboard-certs" created
serviceaccount "kubernetes-dashboard" created
role.rbac.authorization.k8s.io "kubernetes-dashboard-minimal" created
rolebinding.rbac.authorization.k8s.io "kubernetes-dashboard-minimal" created
deployment.apps "kubernetes-dashboard" created
service "kubernetes-dashboard" created
russ in ~
⚡
```

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 deployments --namespace kube-system
$ kubectl get services --namespace kube-system
```

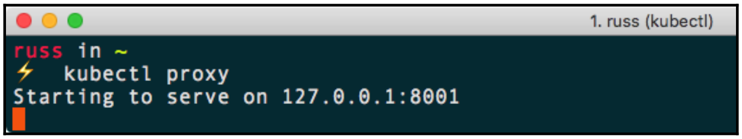
Once your output looks like the following, your dashboard should be installed and ready:



```
russ in ~
⚡  kubectl get deployments --namespace kube-system
NAME                   DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
kube-dns               1         1         1            1           3h
kubernetes-dashboard   1         1         1            1           3m
russ in ~
⚡  kubectl get services --namespace kube-system
NAME                   TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)         AGE
kube-dns               ClusterIP   10.96.0.10     <none>        53/UDP,53/TCP   3h
kubernetes-dashboard   ClusterIP   10.99.112.127  <none>        443/TCP         3m
russ in ~
⚡
```

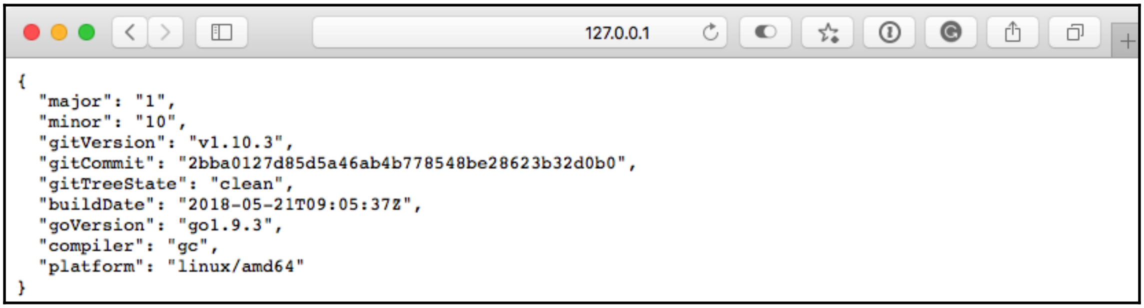
Now that we have our dashboard running, we will find a way to access it. We can do this using the inbuilt proxy service in `kubectl`. Just run the following command to start it up:

```
$ kubectl proxy
```



```
russ in ~
⚡  kubectl proxy
Starting to serve on 127.0.0.1:8001
```

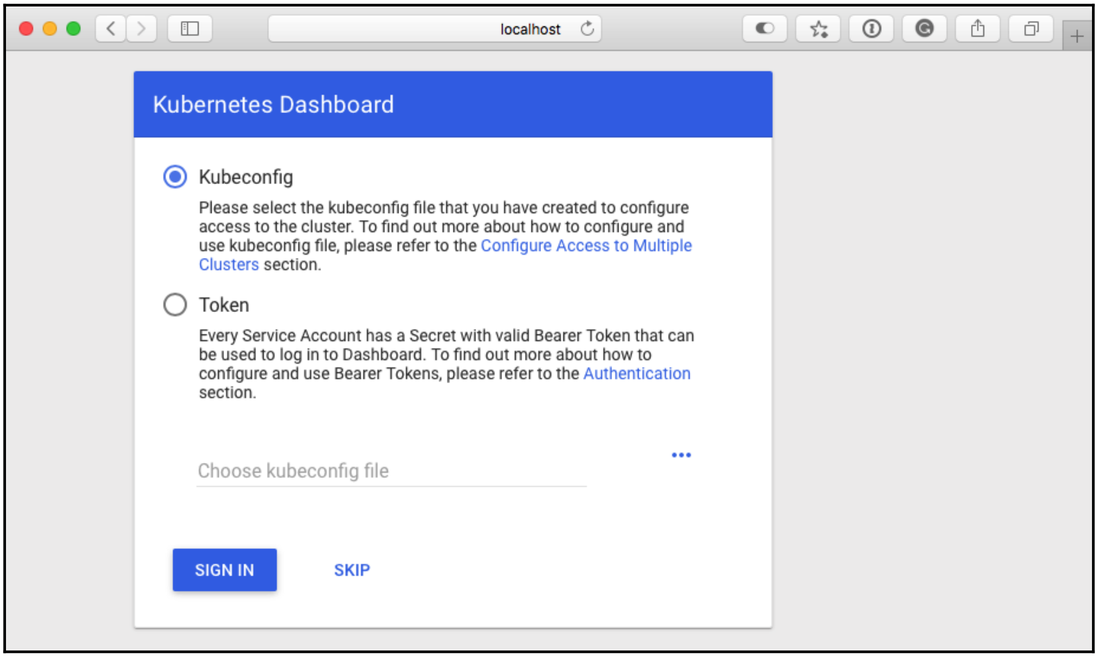
This will start the proxy and opening your browser and going to `http://127.0.0.1:8001/version/` will show you some information on your cluster:



```
{
  "major": "1",
  "minor": "10",
  "gitVersion": "v1.10.3",
  "gitCommit": "2bba0127d85d5a46ab4b778548be28623b32d0b0",
  "gitTreeState": "clean",
  "buildDate": "2018-05-21T09:05:37Z",
  "goVersion": "go1.9.3",
  "compiler": "gc",
  "platform": "linux/amd64"
}
```

However, it's the dashboard we want to see. This can be accessed at `http://localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/`.

You will be greeted with a login screen when you first open the URL in your browser. As we are accessing the dashboard through the proxy, we can just press the **SKIP** button:



Kubernetes Dashboard

● Kubeconfig

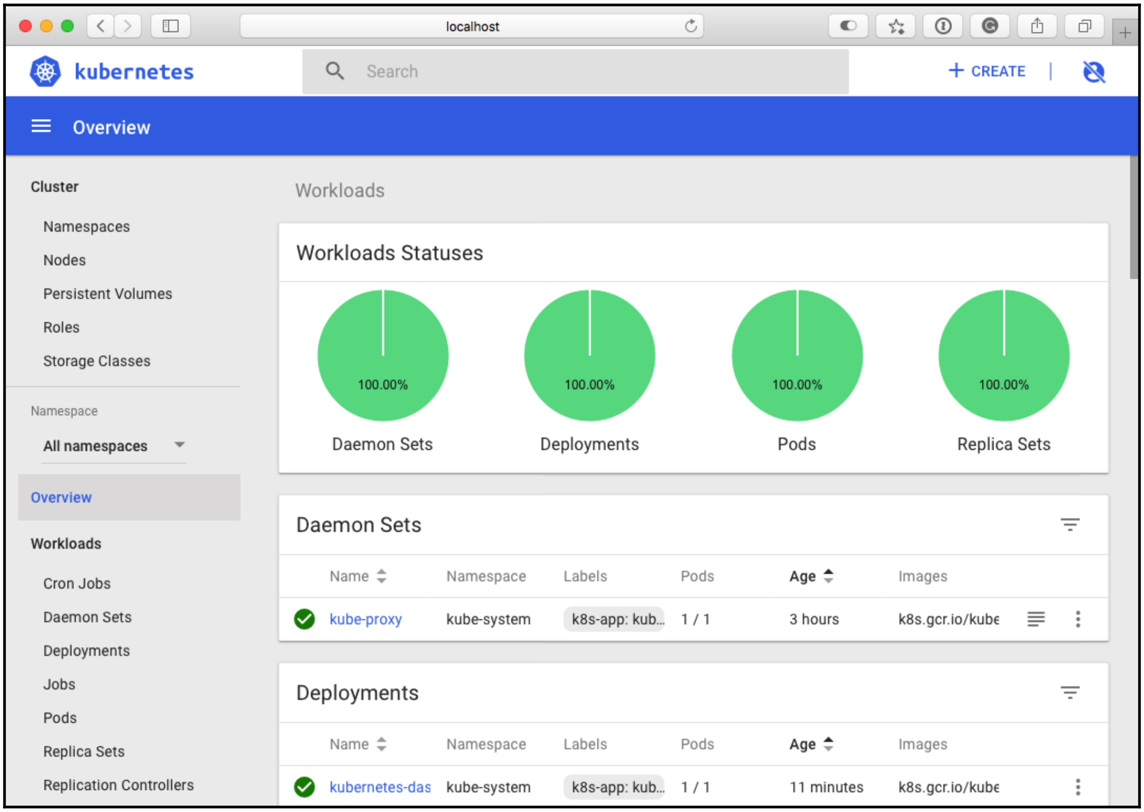
Please select the kubeconfig file that you have created to configure access to the cluster. To find out more about how to configure and use kubeconfig file, please refer to the Configure Access to Multiple Clusters section.

○ 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.

Choose kubeconfig file ...

Once logged in, you will be able to see quite a bit of information on your cluster:



Daemon Sets

	Name	Namespace	Labels	Pods	Age	Images
✓	kube-proxy	kube-system	k8s-app: kub...	1 / 1	3 hours	k8s.gcr.io/kube

Deployments

	Name	Namespace	Labels	Pods	Age	Images
✓	kubernetes-das	kube-system	k8s-app: kub...	1 / 1	11 minutes	k8s.gcr.io/kube

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 Kubernetes as the default orchestrator for Docker stack commands. In the previous chapter, the Docker `stack` command would launch our Docker Compose files in Docker Swarm. The Docker Compose we used looked like the following:

```
version: "3"
services:
  cluster:
    image: russmckendrick/cluster
```

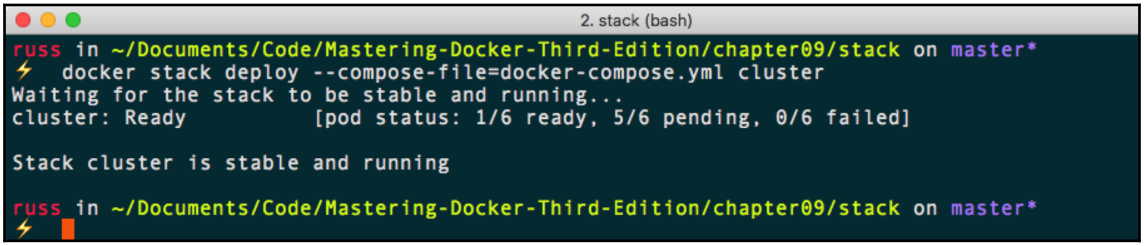
```
      ports:
        - "80:80"
      deploy:
        replicas: 6
        restart_policy:
          condition: on-failure
        placement:
          constraints:
            - node.role == worker
```

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:

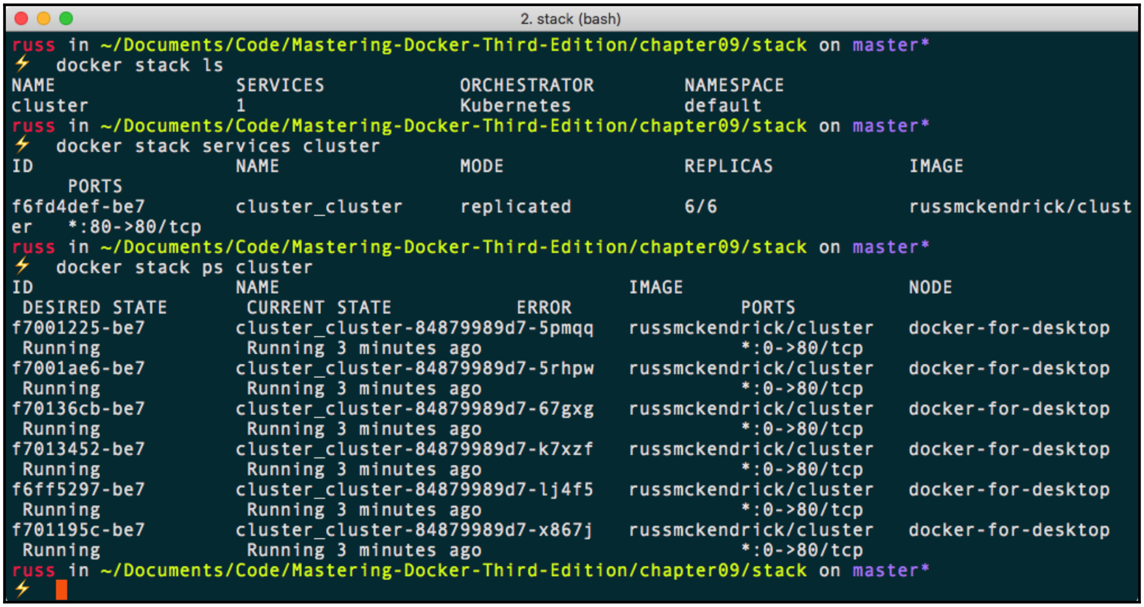
```
$ docker stack deploy --compose-file=docker-compose.yml cluster
```



```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on 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

russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on master*
⚡
```

As you can see, Docker waits until the stack is available before returning you to your prompt. 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:

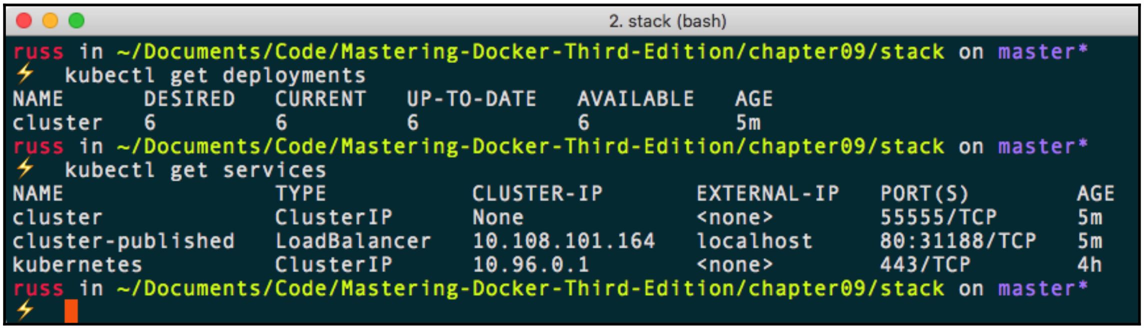
```
$ docker stack ls
$ docker stack services cluster
$ docker stack ps cluster
```



```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on master*
⚡  docker stack ls
NAME                SERVICES            ORCHESTRATOR        NAMESPACE
cluster             1                   Kubernetes          default
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on master*
⚡  docker stack services cluster
ID                  NAME                MODE                REPLICAS            IMAGE
     PORTS
f6fd4def-be7        cluster_cluster     replicated          6/6                 russmckendrick/clust
er   *:80->80/tcp
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on master*
⚡  docker stack ps cluster
ID                  NAME                                IMAGE                     NODE
 DESIRED STATE       CURRENT STATE           ERROR               PORTS
f7001225-be7        cluster_cluster-84879989d7-5pmqq   russmckendrick/cluster    docker-for-desktop
 Running             Running 3 minutes ago                        *:0->80/tcp
f7001ae6-be7        cluster_cluster-84879989d7-5rhpw   russmckendrick/cluster    docker-for-desktop
 Running             Running 3 minutes ago                        *:0->80/tcp
f70136cb-be7        cluster_cluster-84879989d7-67gxg   russmckendrick/cluster    docker-for-desktop
 Running             Running 3 minutes ago                        *:0->80/tcp
f7013452-be7        cluster_cluster-84879989d7-k7xzf   russmckendrick/cluster    docker-for-desktop
 Running             Running 3 minutes ago                        *:0->80/tcp
f6ff5297-be7        cluster_cluster-84879989d7-lj4f5   russmckendrick/cluster    docker-for-desktop
 Running             Running 3 minutes ago                        *:0->80/tcp
f701195c-be7        cluster_cluster-84879989d7-x867j   russmckendrick/cluster    docker-for-desktop
 Running             Running 3 minutes ago                        *:0->80/tcp
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on master*
⚡
```

We can also see details using `kubectl`:

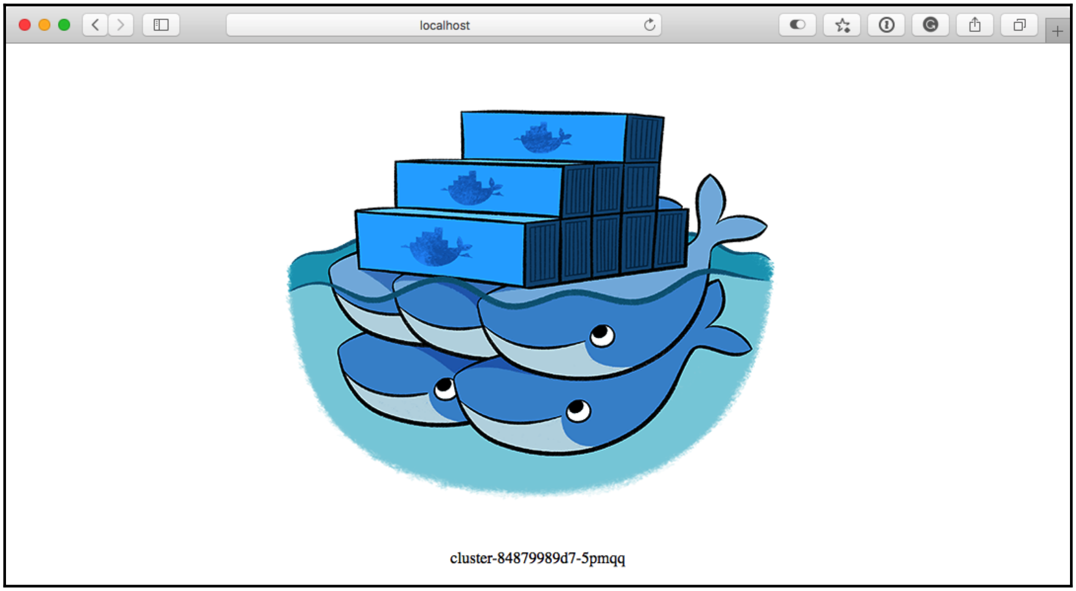
```
$ kubectl get deployments
$ kubectl get services
```



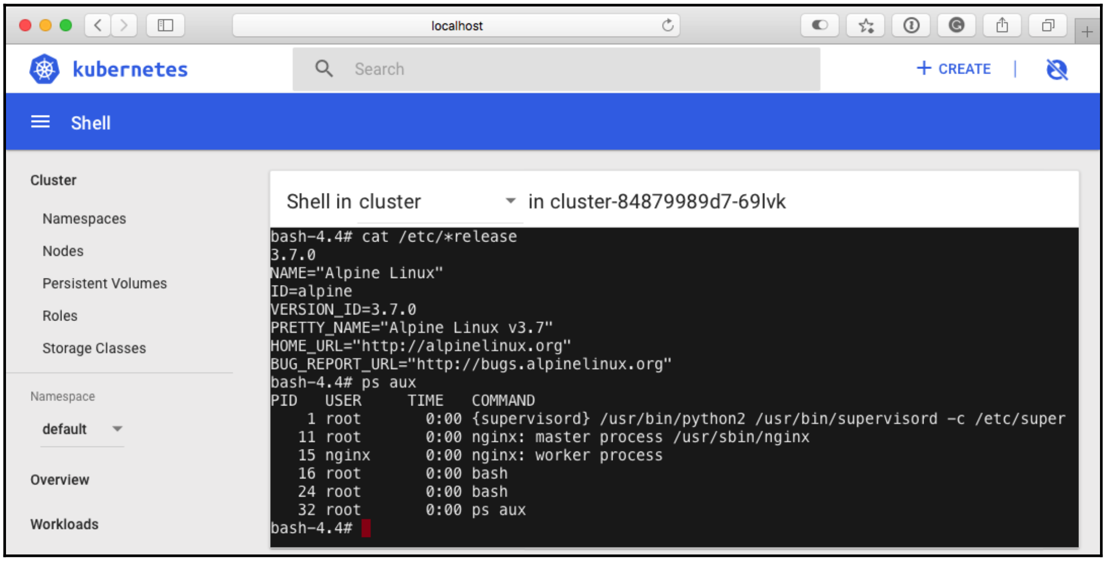
```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on master*
⚡  kubectl get deployments
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
cluster   6         6         6            6           5m
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on master*
⚡  kubectl get services
NAME                TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)         AGE
cluster             ClusterIP      None             <none>        55555/TCP       5m
cluster-published   LoadBalancer   10.108.101.164   localhost     80:31188/TCP    5m
kubernetes          ClusterIP      10.96.0.1        <none>        443/TCP         4h
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on master*
⚡
```

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. 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:



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-84879989d7-69lvk

```
bash-4.4# cat /etc/*release
3.7.0
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.7.0
PRETTY_NAME="Alpine Linux v3.7"
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/super
   11 root       0:00 nginx: master process /usr/sbin/nginx
   15 nginx      0:00 nginx: worker process
   16 root       0:00 bash
   24 root       0:00 bash
   32 root       0:00 ps aux
bash-4.4#
```

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, then you won't be back 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 a Docker Compose file and convert it on the fly to Kubernetes definition files.

To install Kompose on macOS, run the following commands:

```
$ curl -L
https://github.com/kubernetes/kompose/releases/download/v1.16.0/kompose-dar
win-amd64 -o /usr/local/bin/kompose
$ chmod +x /usr/local/bin/kompose
```

Windows 10 users can use Chocolatey to install the binary:

Chocolatey is a command-line based package manager that can be used to install various software packages on your Windows-based machine, similar to how you can use `yum` or `apt-get` on Linux machines or `brew` on macOS.

```
$ choco install kubernetes-kompose
```

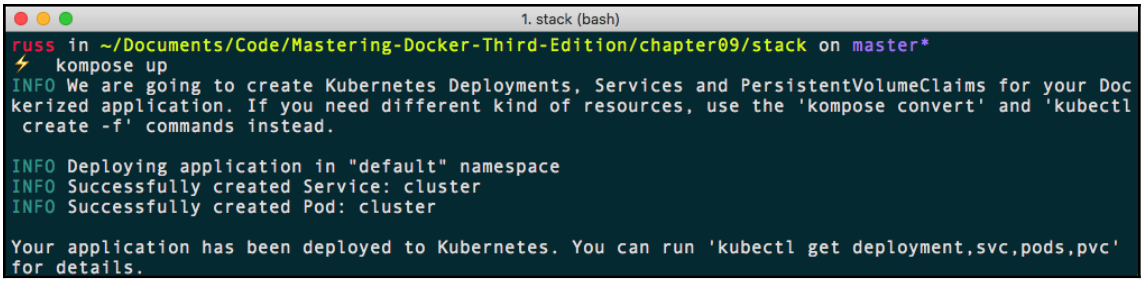
Finally, Linux users can run the following commands:

```
$ curl -L
https://github.com/kubernetes/kompose/releases/download/v1.16.0/kompose-lin
ux-amd64 -o /usr/local/bin/kompose
$ chmod +x /usr/local/bin/kompose
```

Once installed, you can launch your Docker Compose file by running the following command:

```
$ kompose up
```

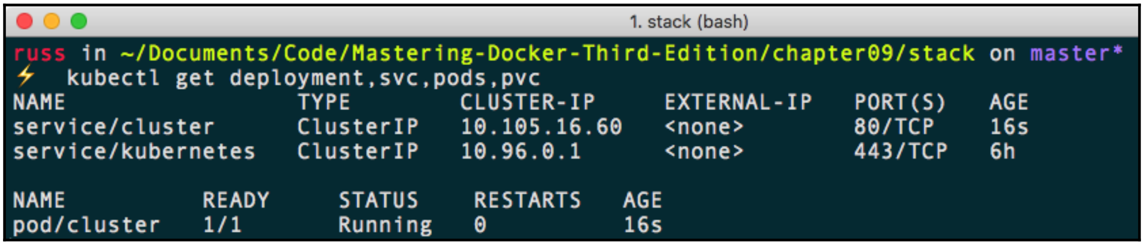
You will get something like the following output:



```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on 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.
```

As suggested by the output, running the following command will give you details on the service and pod we just launched:

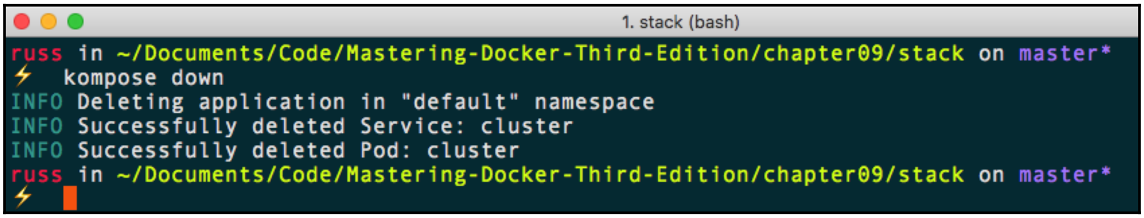
```
$ kubectl get deployment,svc,pods,pvc
```



```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on master*
⚡  kubectl get deployment,svc,pods,pvc
NAME                 TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
service/cluster      ClusterIP   10.105.16.60   <none>        80/TCP    16s
service/kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP   6h

NAME          READY     STATUS    RESTARTS   AGE
pod/cluster   1/1       Running   0          16s
```

You can remove the services and pods by running the following command:

```
$ kompose down
```

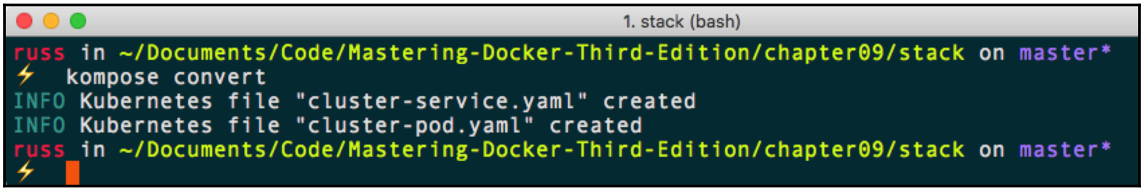


```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on master*
⚡  kompose down
INFO Deleting application in "default" namespace
INFO Successfully deleted Service: cluster
INFO Successfully deleted Pod: cluster
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on master*
⚡  
```

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
```

This will generate the pod and service files:



```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on master*
⚡  kompose convert
INFO Kubernetes file "cluster-service.yaml" created
INFO Kubernetes file "cluster-pod.yaml" created
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on master*
⚡
```

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:

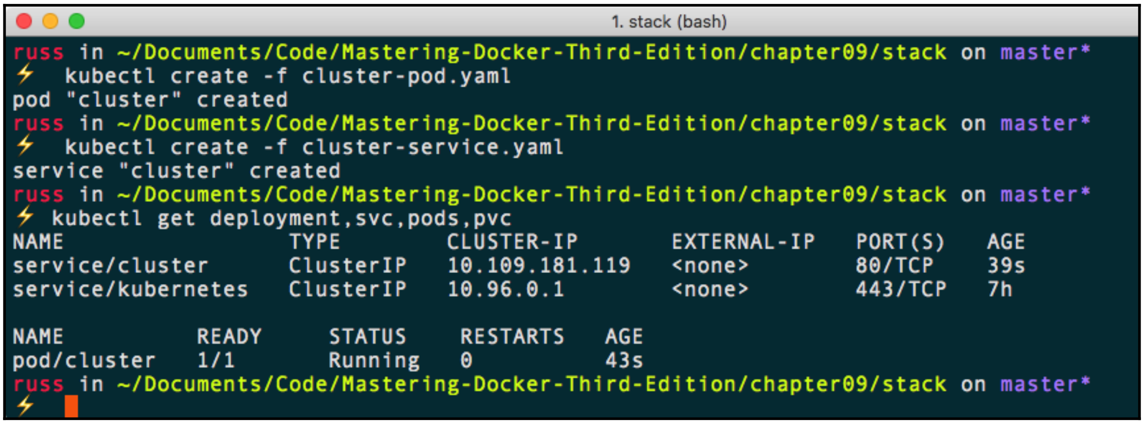
```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    io.kompose.service: cluster
  name: cluster
spec:
  containers:
  - image: russmckendrick/cluster
    name: cluster
    ports:
    - containerPort: 80
    resources: {}
  restartPolicy: OnFailure
status: {}
```

The `cluster-service.yaml` file looks like the following:

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    kompose.cmd: kompose convert
    kompose.version: 1.16.0 (0c01309)
  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
```



```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on master*
⚡  kubectl create -f cluster-pod.yaml
pod "cluster" created
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on master*
⚡  kubectl create -f cluster-service.yaml
service "cluster" created
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on master*
⚡ kubectl get deployment,svc,pods,pvc
NAME                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
service/cluster      ClusterIP   10.109.181.119  <none>        80/TCP    39s
service/kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP   7h

NAME          READY     STATUS    RESTARTS   AGE
pod/cluster   1/1       Running   0          43s
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter09/stack on master*
⚡
```

To remove the cluster pod and service, we just need to run the following command:

```
$ kubectl delete service/cluster pod/cluster
```

While Kubernetes will be popping up in upcoming chapters, you may want to disable the Kubernetes integration within your Docker desktop installation as it does add a slight overhead when it is idle. To do this, just untick **Enable Kubernetes**. When you click **Apply**, Docker will stop all the containers it needed for running 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 under 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.

In the next chapter, we are going to take a look at Docker on public clouds, such as Amazon Web Services, along with briefly revisiting Kubernetes.

Questions

- True or false: When **Show system containers (advanced)** is unticked, you cannot see the images used to launch Kubernetes.
- Which of the four namespaces hosts the containers used to run Kubernetes and enable support within Docker?
- Which command would you run to find out details about a container running in a pod?
- Which command would you use to launch a Kubernetes definition YAML file?
- Typically, which port does the command `kubectl` proxy open on your local machine?
- 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 at:

- 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: `https://pdfs.semanticscholar.org/presentation/4df0/b2bcd39b7757867b1ead3009a628e07d8b57.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 at:

- 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 at:

- Kubernetes for Docker Enterprise announcement: `https://blog.docker.com/2017/10/docker-enterprise-edition-kubernetes/`
- Kubernetes makes the stable release: `https://blog.docker.com/2018/07/kubernetes-is-now-available-in-docker-desktop-stable-channel/`

Finally, the home page for Kompose can be found at:

- Kompose - `http://kompose.io/`

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 using the tools provided by Docker to launch a Docker Swarm cluster in Amazon Web Services and also Microsoft Azure. We will then look at the container solutions offered by Amazon Web Services, Microsoft Azure, and Google Cloud.

The following topics will be covered in this chapter:

- Docker Cloud
- Amazon ECS and AWS Fargate
- Microsoft Azure App Services
- Kubernetes in Microsoft Azure, Google Cloud, and Amazon Web Services

Technical requirements

In this chapter, we will be using various cloud providers, so if you are following along, you will need active accounts with each. Again, the screenshots in this chapter will be from my preferred operating system, macOS. As before, the commands we will be running should work on all three of the operating systems we have targeted so far, unless otherwise stated.

We will also be looking at some of the command-line tools provided by the cloud providers to help manage their services – this chapter does not serve as a detailed how-to guide for these tools, though, and links to documentation will be provided in the *Further reading* section in this chapter for more detailed usage guides.

Check out the following video to see the Code in Action:
`http://bit.ly/2Se544n`

Docker Cloud

Before we start looking at other services, I thought it would be good to quickly discuss Docker Cloud as there are still a lot of references to the cloud management services that were once provided by Docker.

Docker Cloud was made up of several Docker services. These included SaaS offerings for building and hosting images, which was another one of the services offered application, node, and Docker Swarm cluster management. On May 21, 2018, all services that offered the management of remote nodes where closed down.

Docker recommended that Docker Cloud users who managed their nodes using this service should migrate those workloads to either Docker **Community Edition** (**CE**) or Docker **Enterprise Edition** (**EE**) and into the cloud of their own hardware. Docker also recommended the Azure Container Service and Google Kubernetes Engine.

So, for this reason, we will not be discussing any Docker hosted services in this chapter like we did in previous editions of *Mastering Docker*.

However, considering what we have discussed, the next section may seem a little confusing. While Docker has stopped all hosted cloud management services, it still provides tools to help you manage your Docker Swarm clusters in two of the major public cloud providers.

Docker on-cloud

In this section, we are going to look at the two templated cloud offerings from Docker. These both launch Docker Swarm clusters that have deep levels of integration with their target platforms, and have also been built with Docker best practices in mind. Let's look at the Amazon Web Services template first.

Docker Community Edition for AWS

Docker Community Edition for AWS (which we will call Docker for AWS from now on) is an Amazon CloudFormation template created by Docker that is designed to easily launch a Docker Swarm mode cluster in AWS with Docker best practices and recommendations applied.

CloudFormation is a service that's offered by Amazon that allows you to define how you would like your infrastructure to look in a template file that can then be shared or brought under version control.

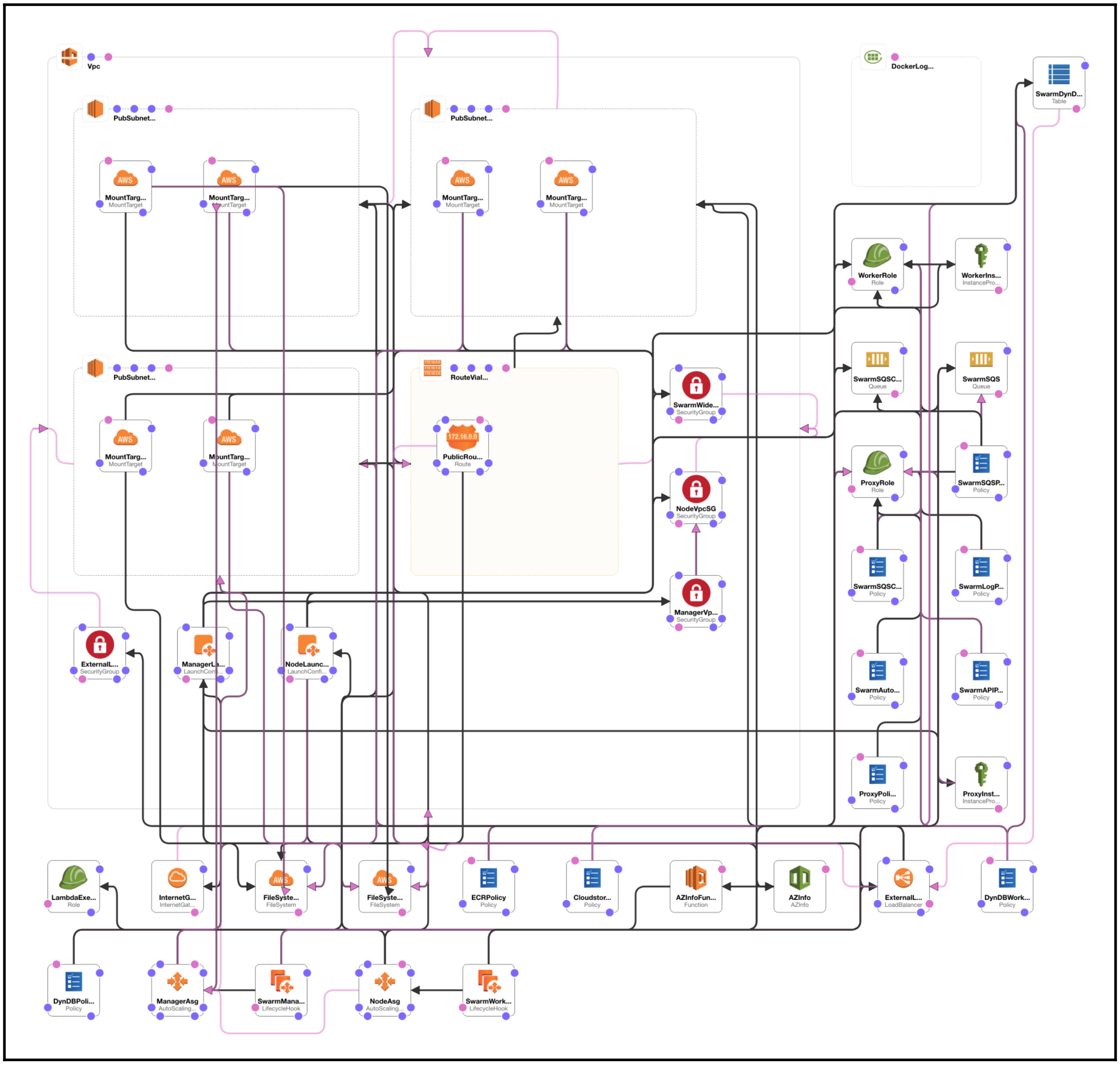
The first thing we need to do – and it's also the only thing we need to configure ahead of launching Docker for AWS—is to ensure that we have an SSH key assigned to our account in the region we will be launching our cluster. To do this, log in to the AWS Console at `https://console.aws.amazon.com/`, or your organization's custom sign-in page if you use one. Once logged in, go to the Service menu, which can be found in the top-left of the page, and find the **EC2** service.

To make sure that you are in your desired region, you can use the region switcher in the top right between your username and the support menu. Once you are in the right region, click on **Key Pairs**, which can be found under **Network & Security** in the left-hand menu. Once on the **Key Pairs** page, you should see a list of your current key pairs. If you have none listed or don't have access to them, you can either click on **Create Key Pair** or **Import Key Pair** and follow the onscreen prompts.

Docker for AWS can be found in the Docker Store at `https://store.docker.com/editions/community/docker-ce-aws`. You have two choices of Docker for AWS: Stable and Edge version.

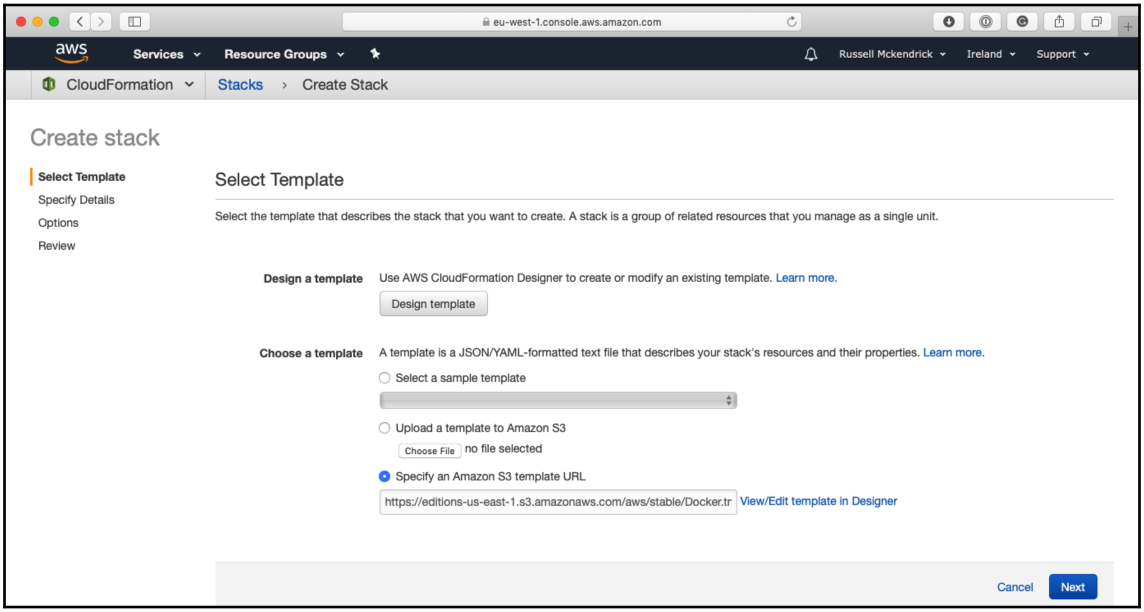
The Edge version contains experimental features from upcoming versions of Docker; because of that, we are going to look at launching Docker for AWS (stable). To do that, just click on the button and you will be taken straight to CloudFormation in your AWS Console with the Docker template already loaded.

You can view the raw template, which is currently made up of 3,100 lines of code, by going to `https://editions-us-east-1.s3.amazonaws.com/aws/stable/Docker.tmpl`, or you can visualize the template in the CloudFormation designer. As you can see from the following visualization, there is a lot going on to launch the cluster:



The beauty of this approach is that you don't have to worry about any of these complications. Docker has you covered and has taken on all of the work of worrying about how to launch the aforementioned infrastructure and services away from you.

The first step in launching the cluster has already been sorted for you. All you have to do is click on **Next** on the **Select Template** page:



Create stack

Select Template

Select the template that describes the stack that you want to create. A stack is a group of related resources that you manage as a single unit.

Design a template Use AWS CloudFormation Designer to create or modify an existing template. Learn more.

Choose a template A template is a JSON/YAML-formatted text file that describes your stack's resources and their properties. Learn more.

○ Select a sample template

○ Upload a template to Amazon S3

Choose File no file selected

● Specify an Amazon S3 template URL

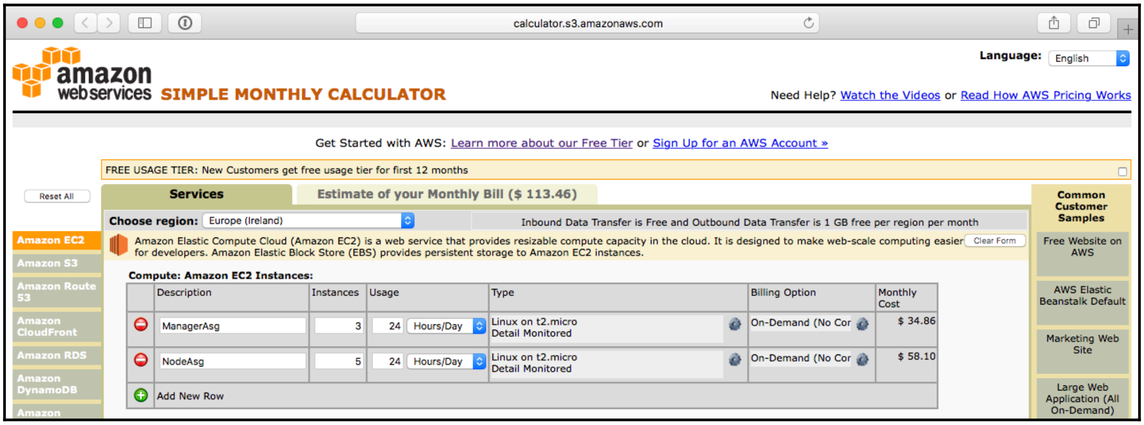
https://editions-us-east-1.s3.amazonaws.com/aws/stable/Docker.tr

Next up, we have to specify some details about our cluster. Other than the SSH key, we are going to be leaving everything at their default values:

- **Stack name**: `Docker`
- **Number of Swarm managers**: `3`
- **Number of Swarm worker nodes**: `5`
- **Which SSH key to use**: (Select your key from the list)
- **Enable daily resource cleanup**: **No**
- **Use CloudWatch for container logging**: **Yes**
- **Create EFS prerequisites for CloudStore**: **No**
- **Swarm manager instance type**: **t2.micro**

- **Manager ephemeral storage volume size**: 20
- **Manager ephemeral storage volume type**: Standard
- **Agent worker instance type**: t2.micro
- **Worker ephemeral storage volume size**: 20
- **Worker ephemeral storage volume type**: Standard
- **Enable EBS I/O optimization?** No
- **Encrypt EFS objects?** False

Once you have checked that everything is **OK**, click on the **Next** button. In the next step, we can leave everything as it is and click on the **Next** button to be taken to a review page. On the review page, you should find a link that gives you the estimated cost:



calculator.s3.amazonaws.com

Language: English

amazon webservices SIMPLE MONTHLY CALCULATOR

Need Help? Watch the Videos or Read How AWS Pricing Works

Get Started with AWS: Learn more about our Free Tier or Sign Up for an AWS Account »

FREE USAGE TIER: New Customers get free usage tier for first 12 months

Reset All

Services | Estimate of your Monthly Bill ($ 113.46)

Choose region: Europe (Ireland)

Inbound Data Transfer is Free and Outbound Data Transfer is 1 GB free per region per month

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers. Amazon Elastic Block Store (EBS) provides persistent storage to Amazon EC2 instances.

Clear Form

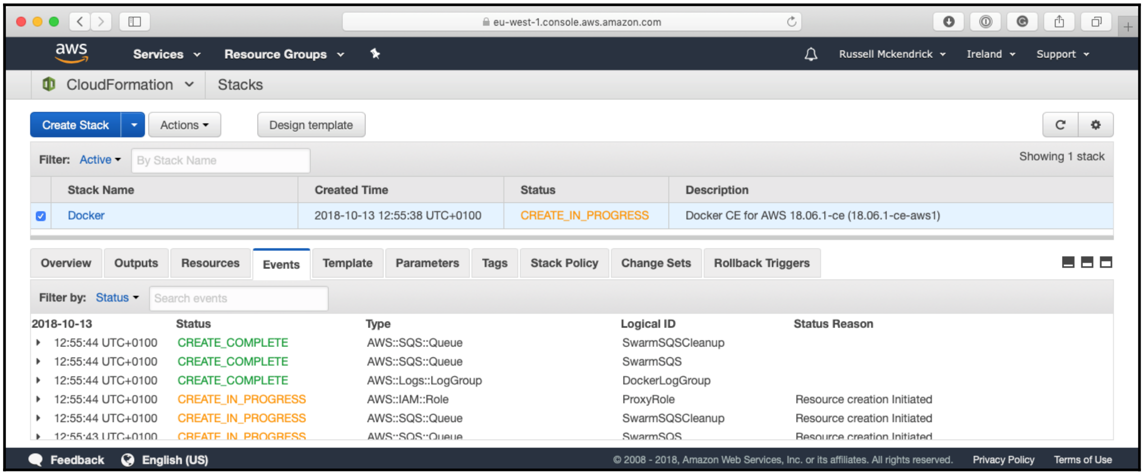
Compute: Amazon EC2 Instances:

	Description	Instances	Usage		Type	Billing Option	Monthly Cost
⊖	ManagerAsg	3	24	Hours/Day	Linux on t2.micro Detail Monitored	On-Demand (No Cor	$ 34.86
⊖	NodeAsg	5	24	Hours/Day	Linux on t2.micro Detail Monitored	On-Demand (No Cor	$ 58.10
⊕	Add New Row						

As you can see, the monthly estimate for my cluster is $113.46.

> TIP
>
> I have had varying success with the Estimate cost link—it may not appear each time you launch the template—if it doesn't and you have answered the questions as per the preceding list, then your costs will be similar to that of mine.

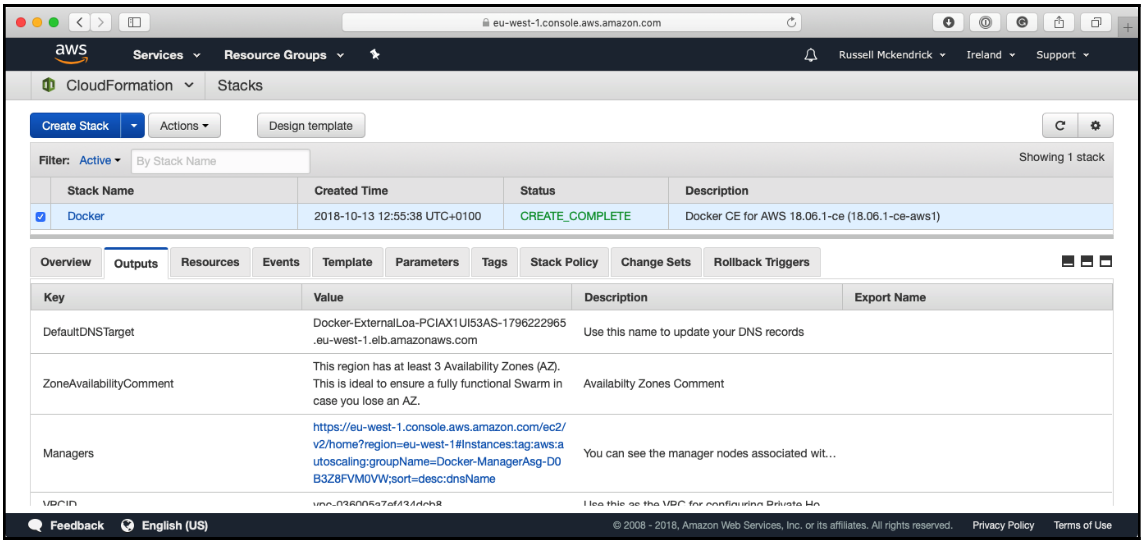
The final thing you need to do before launching the cluster is to tick the box that says **I acknowledge that AWS CloudFormation might create IAM resources** and click on the **Create** button. As you can imagine, it takes a while to launch the cluster; you can check on the status of the launch by selecting your CloudFormation stack in the AWS Console and selecting the **Events** tab:



	Stack Name	Created Time	Status	Description
☑	Docker	2018-10-13 12:55:38 UTC+0100	CREATE_IN_PROGRESS	Docker CE for AWS 18.06.1-ce (18.06.1-ce-aws1)

2018-10-13	Status	Type	Logical ID	Status Reason
12:55:44 UTC+0100	CREATE_COMPLETE	AWS::SQS::Queue	SwarmSQSCleanup	
12:55:44 UTC+0100	CREATE_COMPLETE	AWS::SQS::Queue	SwarmSQS	
12:55:44 UTC+0100	CREATE_COMPLETE	AWS::Logs::LogGroup	DockerLogGroup	
12:55:44 UTC+0100	CREATE_IN_PROGRESS	AWS::IAM::Role	ProxyRole	Resource creation Initiated
12:55:44 UTC+0100	CREATE_IN_PROGRESS	AWS::SQS::Queue	SwarmSQSCleanup	Resource creation Initiated
12:55:43 UTC+0100	CREATE_IN_PROGRESS	AWS::SQS::Queue	SwarmSQS	Resource creation Initiated

After about 15 minutes, you should see the status change from **CREATE_IN_PROGRESS** to **CREATE_COMPLETE**. When you see this, click on the **Outputs** tab and you should see a list of URLs and links:



	Stack Name	Created Time	Status	Description
☑	Docker	2018-10-13 12:55:38 UTC+0100	CREATE_COMPLETE	Docker CE for AWS 18.06.1-ce (18.06.1-ce-aws1)

Key	Value	Description	Export Name
DefaultDNSTarget	Docker-ExternalLoa-PCIAX1UI53AS-1796222965 .eu-west-1.elb.amazonaws.com	Use this name to update your DNS records	
ZoneAvailabilityComment	This region has at least 3 Availability Zones (AZ). This is ideal to ensure a fully functional Swarm in case you lose an AZ.	Availabilty Zones Comment	
Managers	https://eu-west-1.console.aws.amazon.com/ec2/v2/home?region=eu-west-1#Instances:tag:aws:autoscaling:groupName=Docker-ManagerAsg-D0B3Z8FVM0VW;sort=desc:dnsName	You can see the manager nodes associated wit…	
VPCID	vpc-036005a7ef434dcb8	Use this as the VPC for configuring Private Ho	

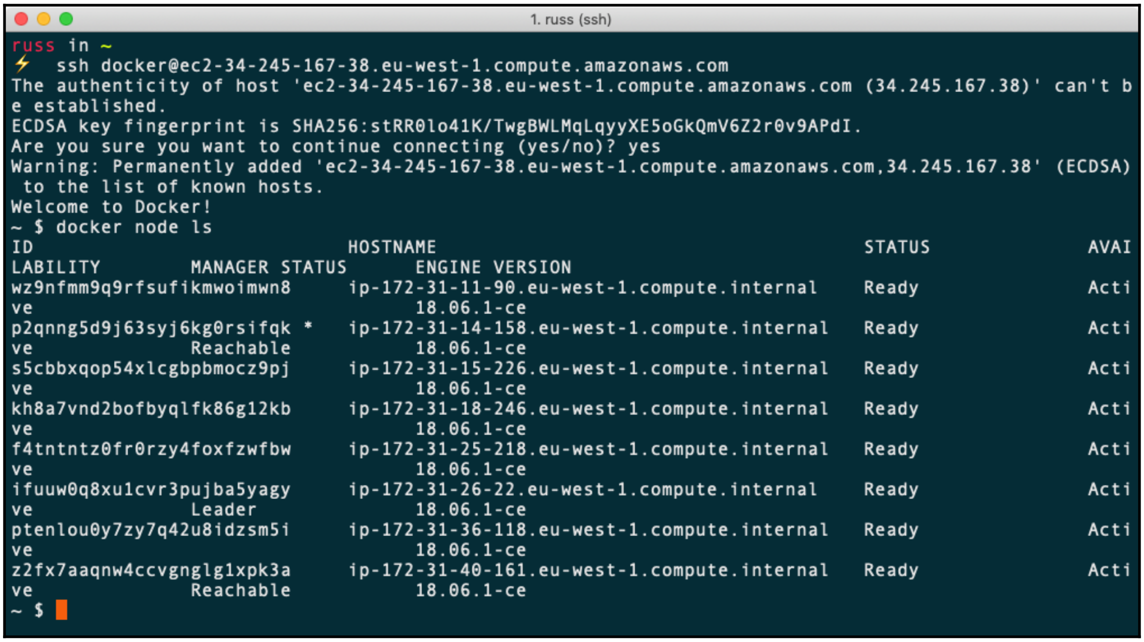
To log in to our Swarm cluster, click on the link next to **Managers** to be taken to a list of EC2 instances, which are our manager nodes. Select one of the instances and then make a note of its public DNS address. In a terminal, SSH to the node, using docker as the username. For example, I ran the following commands to log in and get a list of all nodes:

```
$ ssh docker@ec2-34-245-167-38.eu-west-1.compute.amazonaws.com
$ docker node ls
```

TIP

If you downloaded your SSH key from the AWS Console when you added a key, you should update the preceding command to include the path to your download key, for example, `ssh -i /path/to/private.key docker@ec2-34-245-167-38.eu-west-1.compute.amazonaws.com`.

The preceding commands to log in and get a list of all nodes are shown in the following screenshot:



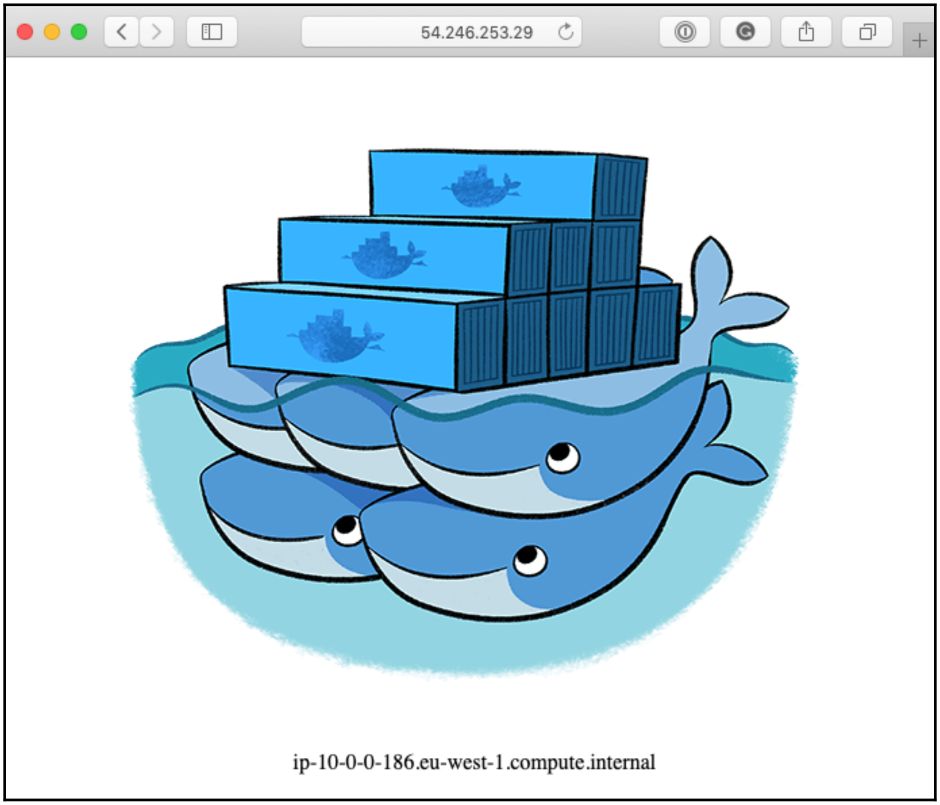
```
russ in ~
⚡  ssh docker@ec2-34-245-167-38.eu-west-1.compute.amazonaws.com
The authenticity of host 'ec2-34-245-167-38.eu-west-1.compute.amazonaws.com (34.245.167.38)' can't b
e established.
ECDSA key fingerprint is SHA256:stRR0lo41K/TwgBWLMqLqyyXE5oGkQmV6Z2r0v9APdI.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ec2-34-245-167-38.eu-west-1.compute.amazonaws.com,34.245.167.38' (ECDSA)
 to the list of known hosts.
Welcome to Docker!
~ $ docker node ls
ID                            HOSTNAME                                      STATUS              AVAI
LABILITY        MANAGER STATUS      ENGINE VERSION
wz9nfmm9q9rfsufikmwoimwn8     ip-172-31-11-90.eu-west-1.compute.internal    Ready               Acti
ve                                  18.06.1-ce
p2qnng5d9j63syj6kg0rsifqk *   ip-172-31-14-158.eu-west-1.compute.internal   Ready               Acti
ve              Reachable           18.06.1-ce
s5cbbxqop54xlcgbpbmocz9pj     ip-172-31-15-226.eu-west-1.compute.internal   Ready               Acti
ve                                  18.06.1-ce
kh8a7vnd2bofbyqlfk86g12kb     ip-172-31-18-246.eu-west-1.compute.internal   Ready               Acti
ve                                  18.06.1-ce
f4tntntz0fr0rzy4foxfzwfbw     ip-172-31-25-218.eu-west-1.compute.internal   Ready               Acti
ve                                  18.06.1-ce
ifuuw0q8xu1cvr3pujba5yagy     ip-172-31-26-22.eu-west-1.compute.internal    Ready               Acti
ve              Leader              18.06.1-ce
ptenlou0y7zy7q42u8idzsm5i     ip-172-31-36-118.eu-west-1.compute.internal   Ready               Acti
ve                                  18.06.1-ce
z2fx7aaqnw4ccvgnglg1xpk3a     ip-172-31-40-161.eu-west-1.compute.internal   Ready               Acti
ve              Reachable           18.06.1-ce
~ $ 
```

From here, you can treat it like any other Docker Swarm cluster. For example, we can launch and scale the cluster service by running these commands:

```
$ docker service create --name cluster --constraint "node.role == worker" -
p 80:80/tcp russmckendrick/cluster
$ docker service scale cluster=6
$ docker service ls
$ docker service inspect --pretty cluster
```

Now that your service has been launched, you can view your application at the URL given as the **DefaultDNSTarget** in the **Outputs** tab of the CloudFormation page. This is an Amazon Elastic load balancer that has all of our nodes sat behind it.

For example, my **DefaultDNSTarget** was `Docker-ExternalLoa-PCIAX1UI53AS-1796222965.eu-west-1.elb.amazonaws.com`. Putting this into my browser showed the clustered application:



ip-10-0-0-186.eu-west-1.compute.internal

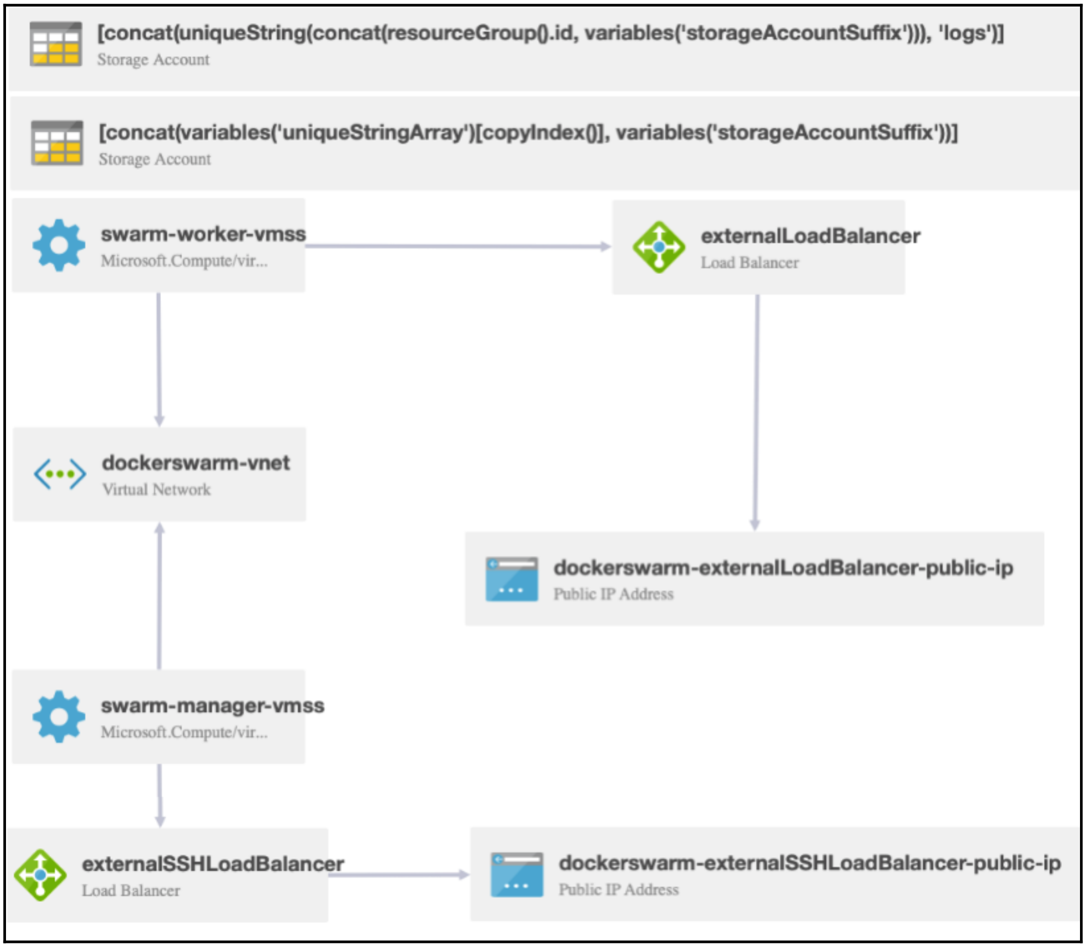
Once you have finished with your cluster, return to the CloudFormation page within the AWS Console, select your stack, and then select **Delete Stack** from the **Actions** drop-down menu. This will remove all traces of your Docker from the Amazon Web Services cluster and stop you from getting any unexpected charges.

TIP

Please make sure that you check that there have not been any problems with the deletion of the stack—if this process encounters any problems, any resources that have been left behind will be charged for.

Docker Community Edition for Azure

Next up, we have the Docker Community Edition for Azure, which I will refer to as Docker for Azure. This uses **Azure Resource Manager** (**ARM**) templates to define our Docker Swarm cluster. Using the ARMViz tool, we can visualize what the cluster will look like:


[concat(uniqueString(concat(resourceGroup().id, variables('storageAccountSuffix'))), 'logs')]
Storage Account
[concat(variables('uniqueStringArray')[copyIndex()], variables('storageAccountSuffix'))]
Storage Account
swarm-worker-vmss
Microsoft.Compute/vir...
externalLoadBalancer
Load Balancer
dockerswarm-vnet
Virtual Network
dockerswarm-externalLoadBalancer-public-ip
Public IP Address
swarm-manager-vmss
Microsoft.Compute/vir...
externalSSHLoadBalancer
Load Balancer
dockerswarm-externalSSHLoadBalancer-public-ip
Public IP Address

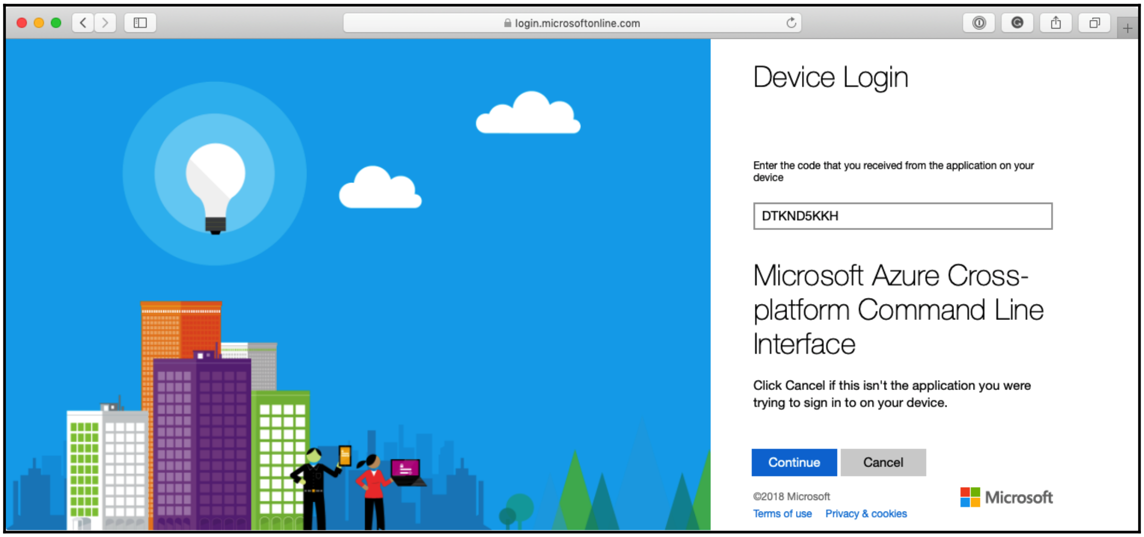
As you can see, it will launch VMs, load balancers with public IP addresses attached, and storage. Before we launch our cluster, we need to find a few bits of information about our Azure account:

- AD Service principle ID
- AD Service principle key

To generate the required information, we are going to use a helper script that runs inside of a container. To run the script, you will need admin access to a valid Azure subscription. To run the script, simply run the following command:

```
$ docker run -ti docker4x/create-sp-azure sp-name
```

This will give you a URL, `https://microsoft.com/devicelogin`, and also a code to enter. Go to the URL and enter the code:



Device Login

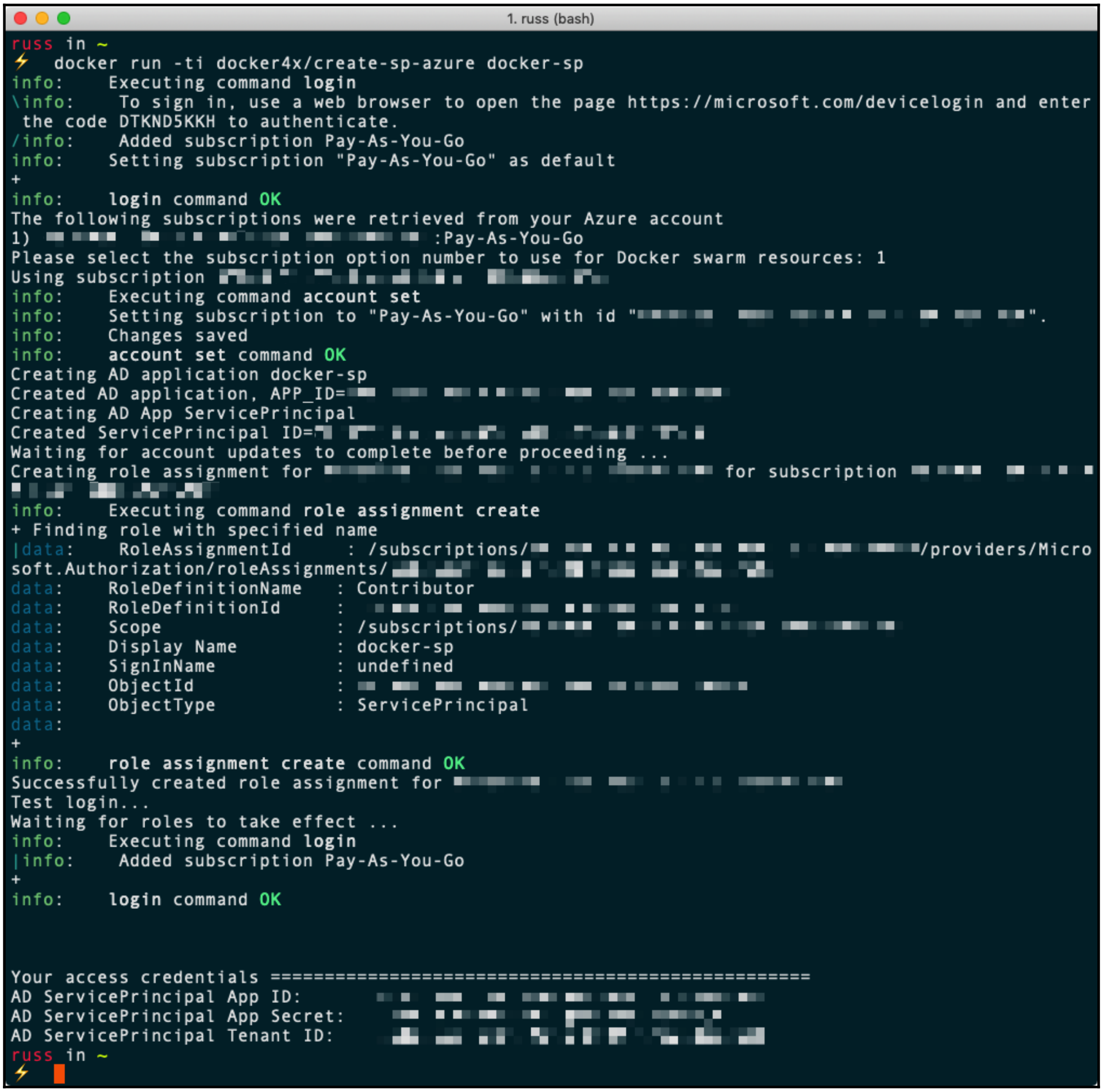
Enter the code that you received from the application on your device

DTKND5KKH

Microsoft Azure Cross-platform Command Line Interface

Click Cancel if this isn't the application you were trying to sign in to on your device.

This will log you in to your account on the command-line and ask you which of your subscriptions you would like to use. The full output of the helper script can be found in the following screenshot:



```
russ in ~
⚡  docker run -ti docker4x/create-sp-azure docker-sp
info:    Executing command login
\info:    To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter
 the code DTKND5KKH to authenticate.
/info:    Added subscription Pay-As-You-Go
info:    Setting subscription "Pay-As-You-Go" as default
+
info:    login command OK
The following subscriptions were retrieved from your Azure account
1) [redacted]:Pay-As-You-Go
Please select the subscription option number to use for Docker swarm resources: 1
Using subscription [redacted]
info:    Executing command account set
info:    Setting subscription to "Pay-As-You-Go" with id "[redacted]".
info:    Changes saved
info:    account set command OK
Creating AD application docker-sp
Created AD application, APP_ID=[redacted]
Creating AD App ServicePrincipal
Created ServicePrincipal ID=[redacted]
Waiting for account updates to complete before proceeding ...
Creating role assignment for [redacted] for subscription [redacted]
info:    Executing command role assignment create
+ Finding role with specified name
|data:    RoleAssignmentId     : /subscriptions/[redacted]/providers/Micro
soft.Authorization/roleAssignments/[redacted]
data:    RoleDefinitionName   : Contributor
data:    RoleDefinitionId     : [redacted]
data:    Scope                : /subscriptions/[redacted]
data:    Display Name         : docker-sp
data:    SignInName           : undefined
data:    ObjectId             : [redacted]
data:    ObjectType           : ServicePrincipal
data:
+
info:    role assignment create command OK
Successfully created role assignment for [redacted]
Test login...
Waiting for roles to take effect ...
info:    Executing command login
|info:    Added subscription Pay-As-You-Go
+
info:    login command OK



Your access credentials ==================================================
AD ServicePrincipal App ID:       [redacted]
AD ServicePrincipal App Secret:   [redacted]
AD ServicePrincipal Tenant ID:    [redacted]
russ in ~
⚡
```

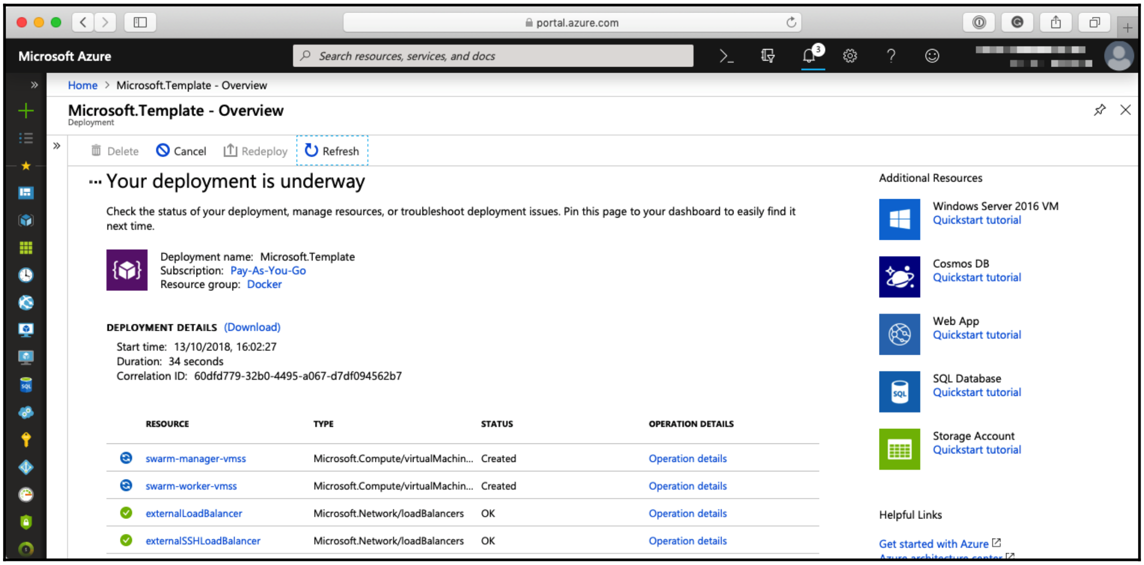
At the very end of the output is the information you need, so please make a note of it.

At the time of writing this book, there is a known issue of using the **Docker for Azure (Stable)** button on the Docker Community Edition for Azure page in the Docker Store. For now, we need to use an older version of the template. You can do this by using the following link: `https://portal.azure.com/#create/Microsoft.Template/uri/https%3A%2F%2Fdownload.docker.com%2Fazure%2Fstable%2F18.03.0%2FDocker.tmpl`.

This will open up the Azure portal and present you with a screen where you need to enter several bits of information:

- **Subscription**: Select the subscription you would like to use from the drop-down list
- **Resource group**: Select the resource group you would like to use or create a new one
- **Location**: Select where you would like to launch you Docker Swarm cluster
- **Ad Service Principle App ID**: This was generated by the helper script we just ran
- **Ad Service Principle App Secret**: This was generated by the helper script we just ran
- **Enable Ext Logs**: Yes
- **Enable System Prune**: No
- **Linux SSH Public Key**: Enter the public portion of your local SSH key here
- **Linux Worker Count**: 2
- **Linux Worker VM Size**: Standard_D2_v2
- **Manager Count**: 1
- **Manager VM Size**: Standard_D2_v2
- **Swarm Name**: dockerswarm

Agree to the terms and conditions and then click on the **Purchase** button at the bottom of the page. Once you view the progress of the launch by clicking on the **Deployment in Progress** link in the notification area on the top of the menu, you should see something like this:



Microsoft.Template - Overview
Deployment

··· Your deployment is underway

Check the status of your deployment, manage resources, or troubleshoot deployment issues. Pin this page to your dashboard to easily find it next time.

Deployment name: Microsoft.Template
Subscription: Pay-As-You-Go
Resource group: Docker

DEPLOYMENT DETAILS (Download)

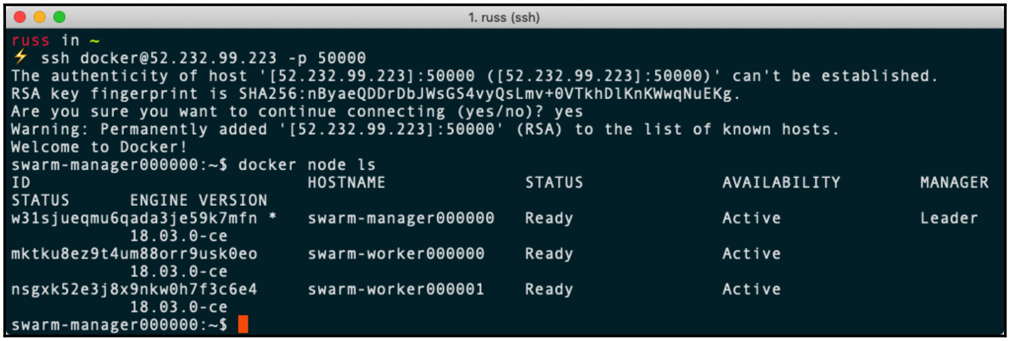
Start time: 13/10/2018, 16:02:27
Duration: 34 seconds
Correlation ID: 60dfd779-32b0-4495-a067-d7df094562b7

RESOURCE	TYPE	STATUS	OPERATION DETAILS
swarm-manager-vmss	Microsoft.Compute/virtualMachin...	Created	Operation details
swarm-worker-vmss	Microsoft.Compute/virtualMachin...	Created	Operation details
externalLoadBalancer	Microsoft.Network/loadBalancers	OK	Operation details
externalSSHLoadBalancer	Microsoft.Network/loadBalancers	OK	Operation details

Once completed, you will see several services listed under the resource group that you chose or created. One of these will be `dockerswarm-externalSSHLoadBalancer-public-ip`. Drill-down into the resource and you will be given the IP address that you can use to SSH into your Swarm Manager. To do this, run the following command:

```
$ ssh docker@52.232.99.223 -p 50000
$ docker node ls
```

Note that we are using port 5000 rather than the standard port 22. You should see something like the following:



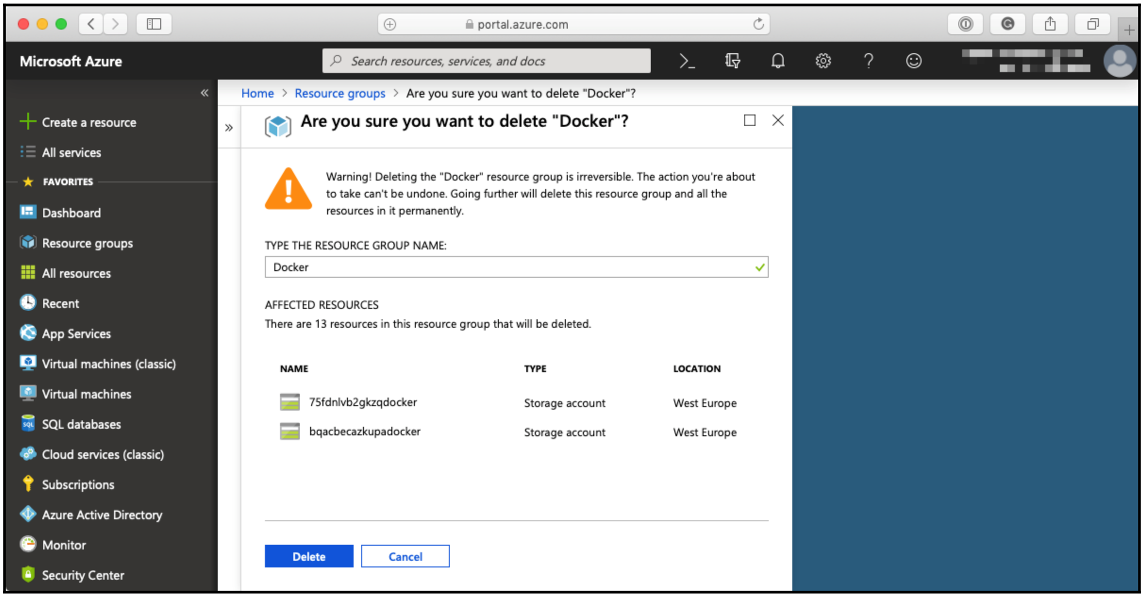
1. russ (ssh)

```
russ in ~
⚡ ssh docker@52.232.99.223 -p 50000
The authenticity of host '[52.232.99.223]:50000 ([52.232.99.223]:50000)' can't be established.
RSA key fingerprint is SHA256:nByaeQDDrDbJWsGS4vyQsLmv+0VTkhDlKnKWwqNuEKg.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[52.232.99.223]:50000' (RSA) to the list of known hosts.
Welcome to Docker!
swarm-manager000000:~$ docker node ls
ID                            HOSTNAME              STATUS              AVAILABILITY        MANAGER
STATUS      ENGINE VERSION
w31sjueqmu6qada3je59k7mfn *   swarm-manager000000   Ready               Active              Leader
            18.03.0-ce
mktku8ez9t4um88orr9usk0eo     swarm-worker000000    Ready               Active
            18.03.0-ce
nsgxk52e3j8x9nkw0h7f3c6e4     swarm-worker000001    Ready               Active
            18.03.0-ce
swarm-manager000000:~$
```

Once you are logged in to the manager node, we can then use the following commands to launch an application:

```
$ docker service create --name cluster --constraint "node.role == worker" -
p 80:80/tcp russmckendrick/cluster
$ docker service scale cluster=6
$ docker service ls
$ docker service inspect --pretty cluster
```

Once launched, going to `dockerswarm-externalLoadBalancer-public-ip`—this will show the application. Once you have finished with your cluster, I would recommend removing the resource group rather than trying to remove the individual resources:



Are you sure you want to delete "Docker"?

Warning! Deleting the "Docker" resource group is irreversible. The action you're about to take can't be undone. Going further will delete this resource group and all the resources in it permanently.

TYPE THE RESOURCE GROUP NAME:

Docker ✓

AFFECTED RESOURCES

There are 13 resources in this resource group that will be deleted.

NAME	TYPE	LOCATION
75fdnlvb2gkzqdocker	Storage account	West Europe
bqacbecazkupadocker	Storage account	West Europe

Remember, you will be charged for the resources while they are active, even if you are not using them.

Like with the Amazon Web Services cluster, please make sure that the resources are removed fully, otherwise you may end up with an unexpected bill.

Docker for Cloud summary

As you can see, it has been mostly straightforward to launch a Swarm cluster in both Azure as well as Amazon Web Services by using the templates provided by Docker. While these templates are great, if you are starting out, they get very little in the way of support from Docker. I would recommend that if you are looking at an easy way to launch containers that are running production workloads in public clouds, you can take a look at the some of the solutions that we are going to be discussing next.

Amazon ECS and AWS Fargate

Amazon Web Services offers a few different container solutions. The one we are going to look at in this section is part of the Amazon **Elastic Container Service** (**ECS**) and is called AWS Fargate.

Traditionally, Amazon ECS launches EC2 instances. Once launched, an Amazon ECS agent is deployed alongside a container runtime that allows you to then manage your containers using the AWS Console and command-line tools. AWS Fargate removes the need to launch EC2 instances, allowing you to simply launch containers without having to worry about managing a cluster or having the expense of EC2 instances.

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.

Amazon ECS uses the following components:

- Container definition
- Task definition
- Service
- Cluster

The first step in launching our AWS Fargate hosted container is to actually configure the first two components, that is, the container and task definitions.

The container definition is where the base configuration for the container is defined. Think of this as adding the flags you would use to launch a container using the Docker client on the command line—for example, you name the container, define the image to use, set the network, and so on.

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

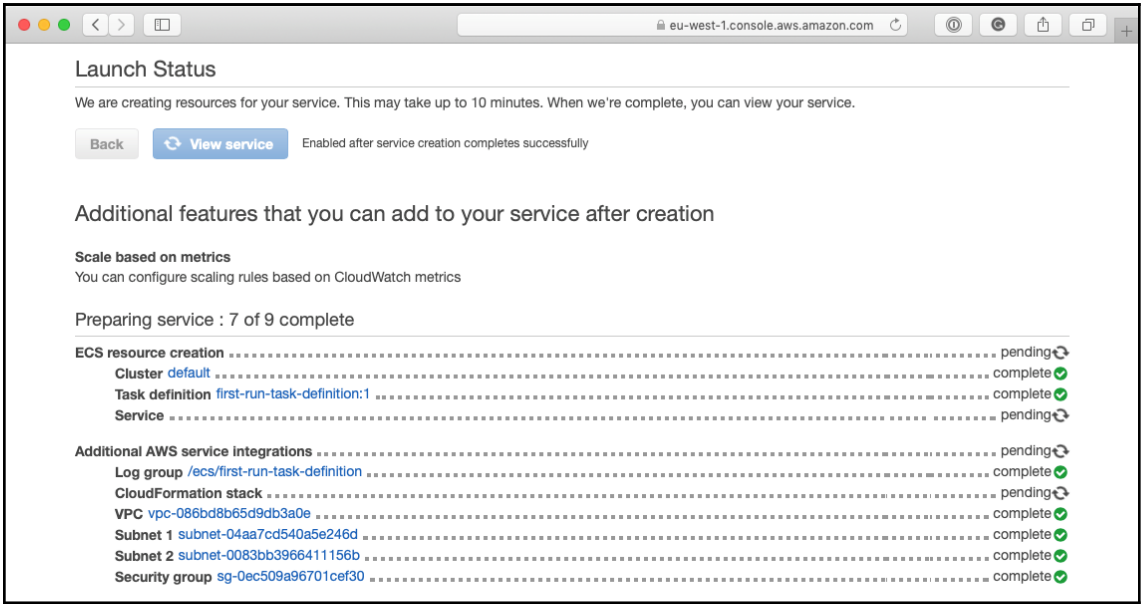
Then, click on the **Update** button. For the task definition, click on the **Edit** button and enter the following:

- **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

Once 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.

A service runs tasks which in turn has 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.

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, then 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

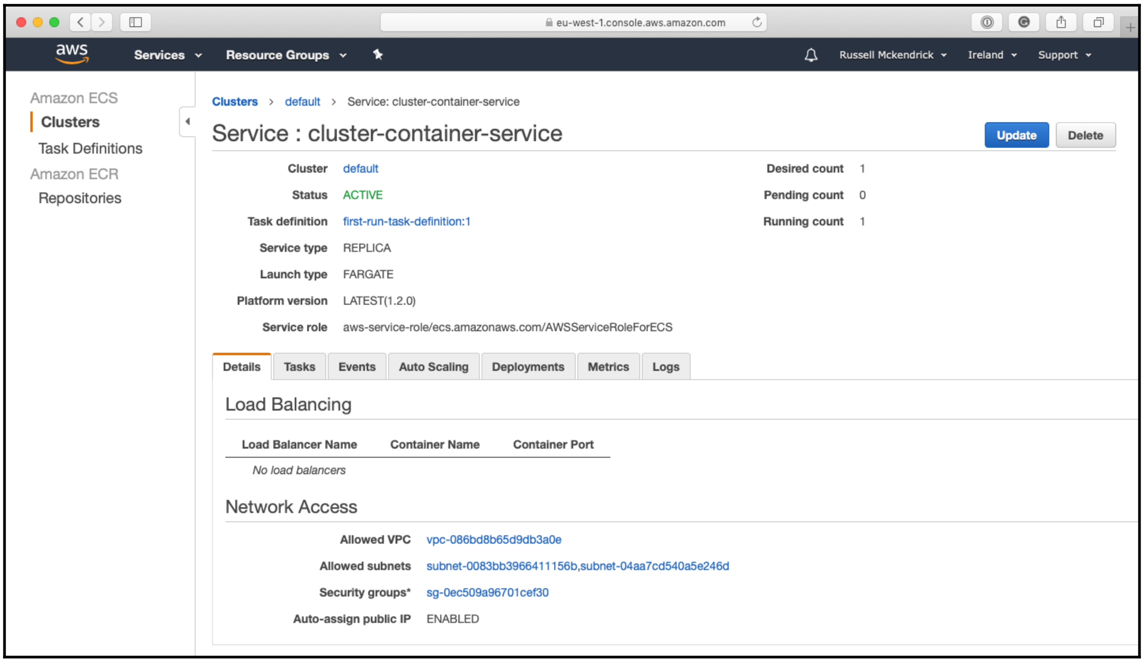
Scale based on metrics
You can configure scaling rules based on CloudWatch metrics

Preparing service : 7 of 9 complete

ECS resource creation .... pending
Cluster default .... complete ✓
Task definition first-run-task-definition:1 .... complete ✓
Service .... pending

Additional AWS service integrations .... pending
Log group /ecs/first-run-task-definition .... complete ✓
CloudFormation stack .... pending
VPC vpc-086bd8b65d9db3a0e .... complete ✓
Subnet 1 subnet-04aa7cd540a5e246d .... complete ✓
Subnet 2 subnet-0083bb3966411156b .... complete ✓
Security group sg-0ec509a96701cef30 .... complete ✓

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:



Clusters > default > Service: cluster-container-service

Service : cluster-container-service

Cluster	default	Desired count	1
Status	ACTIVE	Pending count	0
Task definition	first-run-task-definition:1	Running count	1
Service type	REPLICA		
Launch type	FARGATE		
Platform version	LATEST(1.2.0)		
Service role	aws-service-role/ecs.amazonaws.com/AWSServiceRoleForECS		

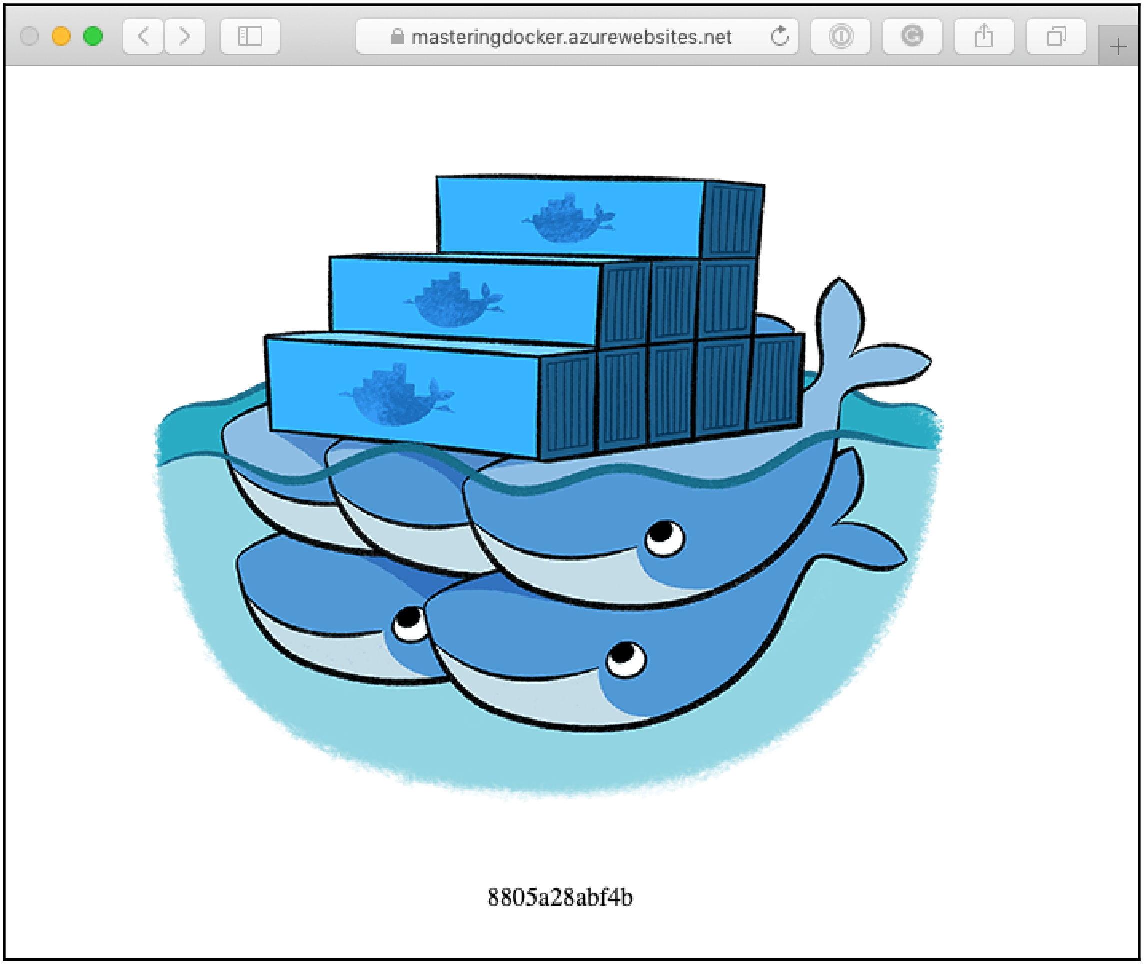
Load Balancing

Load Balancer Name	Container Name	Container Port
No load balancers		

Network Access

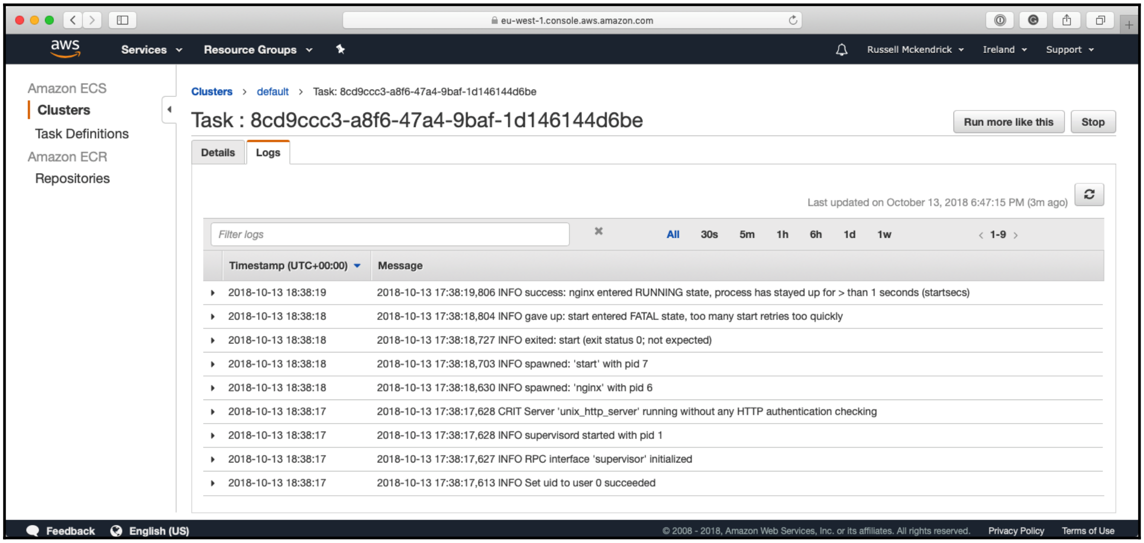
Allowed VPC	vpc-086bd8b65d9db3a0e
Allowed subnets	subnet-0083bb3966411156b,subnet-04aa7cd540a5e246d
Security groups*	sg-0ec509a96701cef30
Auto-assign public IP	ENABLED

Now, we just 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:



8805a28abf4b

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:


eu-west-1.console.aws.amazon.com
aws
Services
Resource Groups
Russell Mckendrick
Ireland
Support
Amazon ECS
Clusters
Task Definitions
Amazon ECR
Repositories
Clusters > default > Task: 8cd9ccc3-a8f6-47a4-9baf-1d146144d6be
Task : 8cd9ccc3-a8f6-47a4-9baf-1d146144d6be
Run more like this
Stop
Details
Logs
Last updated on October 13, 2018 6:47:15 PM (3m ago)
Filter logs
All
30s
5m
1h
6h
1d
1w
1-9
Feedback
English (US)
© 2008 - 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

```
Timestamp (UTC+00:00)   Message
2018-10-13 18:38:19     2018-10-13 17:38:19,806 INFO success: nginx entered RUNNING state, process has stayed up for > than 1 seconds (startsecs)
2018-10-13 18:38:18     2018-10-13 17:38:18,804 INFO gave up: start entered FATAL state, too many start retries too quickly
2018-10-13 18:38:18     2018-10-13 17:38:18,727 INFO exited: start (exit status 0; not expected)
2018-10-13 18:38:18     2018-10-13 17:38:18,703 INFO spawned: 'start' with pid 7
2018-10-13 18:38:18     2018-10-13 17:38:18,630 INFO spawned: 'nginx' with pid 6
2018-10-13 18:38:17     2018-10-13 17:38:17,628 CRIT Server 'unix_http_server' running without any HTTP authentication checking
2018-10-13 18:38:17     2018-10-13 17:38:17,628 INFO supervisord started with pid 1
2018-10-13 18:38:17     2018-10-13 17:38:17,627 INFO RPC interface 'supervisor' initialized
2018-10-13 18:38:17     2018-10-13 17:38:17,613 INFO Set uid to user 0 succeeded
```

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 great—there is a low barrier to launching the containers and the number of supporting resources in 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.

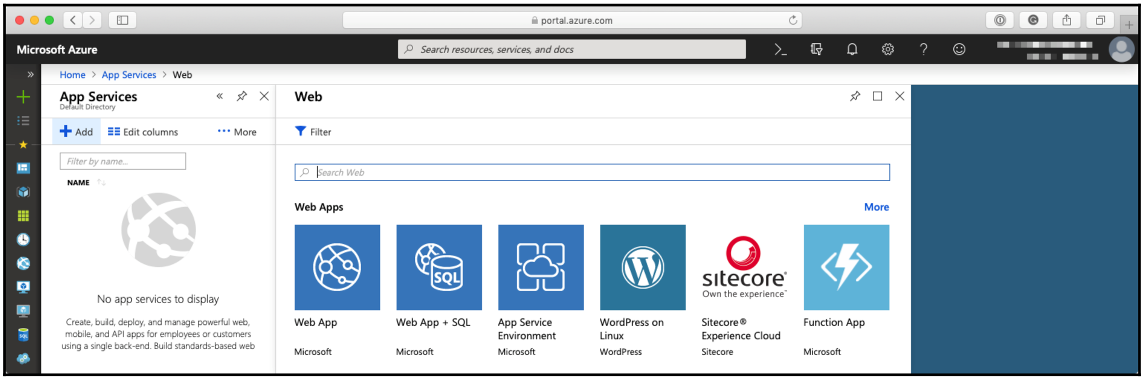
Next, we are going to take a look at Azure App Services.

Microsoft Azure App Services

Microsoft Azure App Services 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, Python, and Ruby, 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. To do this, login to the Azure portal at `https://portal.azure.com/` and select **App Services** from the left-hand side menu.

On the page that loads, click on the **+Add** button. You have several options to choose from here:



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

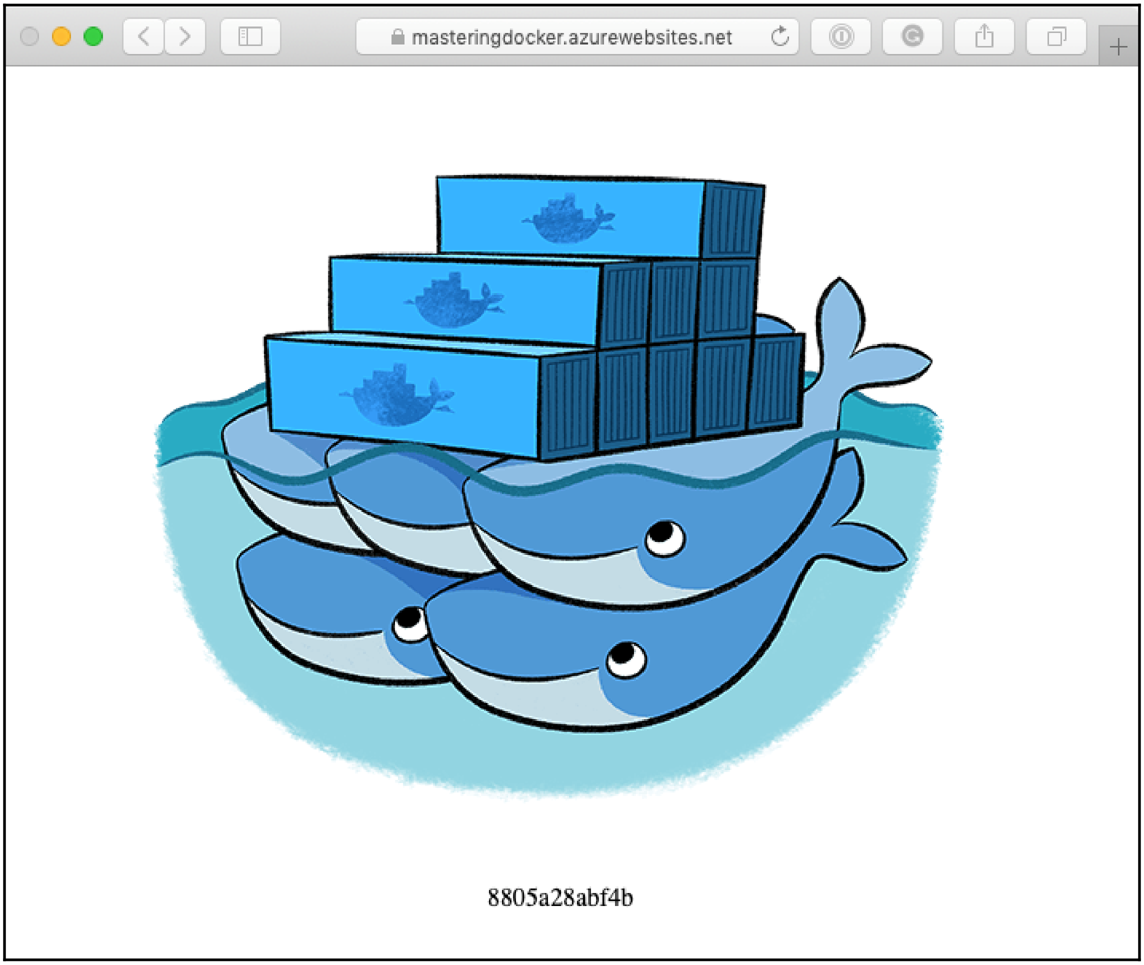
We are going to launching a Web App, so click on the tile for that. Once the tile has expanded, click on the **Create** button.

On the page that opens, there are several options. Fill them in as follows:

- **App Name**: Choose a unique name for the application.
- **Subscription**: Choose a valid subscription.
- **Resource Group**: Leave the create new option selected.
- **OS**: Leave as Linux.
- **Publish**: Select Docker Image.
- **App Service plan/location**: By default, the most expensive plan is selected, so clicking here will take you a page where you can create a new plan. To do this, click on **Create new**, name your plan and select a location, and then finally choose a Pricing tier. For our needs, the **Dev/Test** plan will be fine. Once selected, click on **Apply**.

- **Configure container:** Clicking here will take you to the container options. Here, you have a few options: Single Container, Docker Compose, or Kubernetes. For now, we are going to launching a single container. Click on the **Docker Hub** option and enter `russmckendrick/cluster:latest`. Once entered, you will be able to click on the **Apply** button.

Once all of the information has been filled in, you will be able to then click on **Create** to launch the Web App Service. Once launched, you should be able to access the service via the URL provided by Azure, for example, mine was `https://masteringdocker.azurewebsites.net/`. Opening this a browser will display the cluster application:



8805a28abf4b

As you can see, this time, we have the container ID rather than a full hostname like we got when launching the container on AWS Fargate. The container at this spec will cost us around $0.05 per hour, or $36.50 per month. To remove the container, simply remove the resource group.

Kubernetes in Microsoft Azure, Google Cloud, and Amazon Web Services

The last thing we are going to take a look at is how easy is it to launch a Kubernetes cluster in the three main public clouds. In the previous chapter, we launched a Kubernetes cluster locally using the built-in functionality of the Docker Desktop applications. To start with, we are going to look at the quickest way to get started with Kubernetes on public clouds, starting with Microsoft Azure.

Azure Kubernetes Service

The **Azure Kubernetes Service** (**AKS**), is an extremely simple service to launch and configure. I will be using the Azure command-line tools on my local machine; you will also be able to use the command-line tools by using the Azure Cloud Shell which is built into the Azure Portal.

The first thing we will need to do is create a resource group to launch our AKS cluster into. To create one called `MasteringDockerAKS`, run the following command:

```
$ az group create --name MasteringDockerAKS --location eastus
```

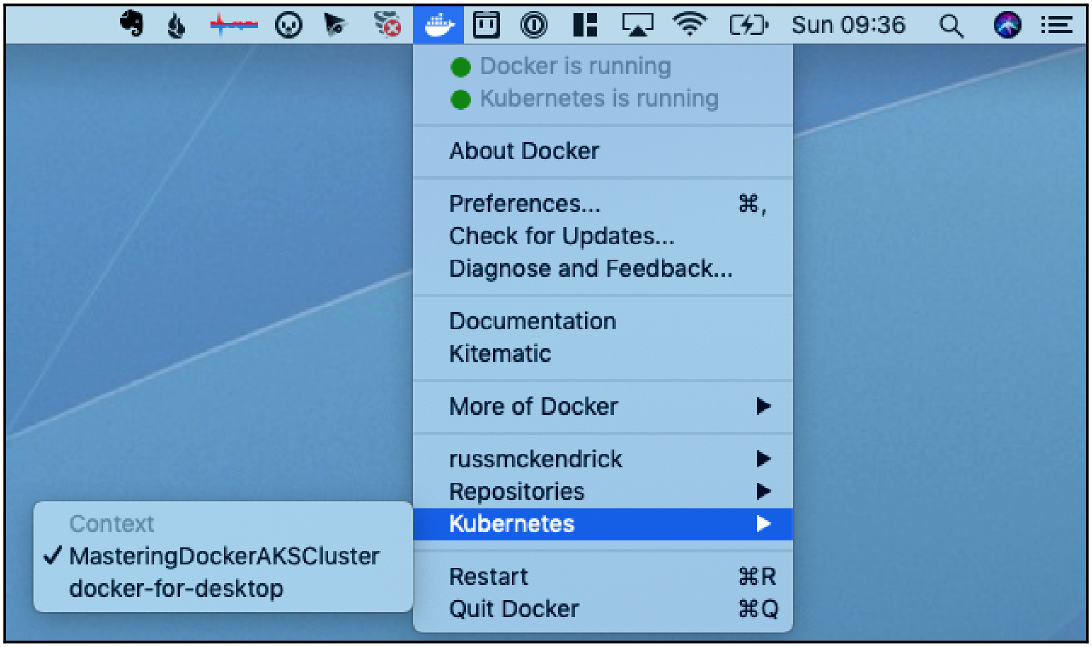
Now that we have the resource group, we can launch a two node Kubernetes cluster by running the following command:

```
$ az aks create --resource-group MasteringDockerAKS \
    --name MasteringDockerAKSCluster \
    --node-count 2 \
    --enable-addons monitoring \
    --generate-ssh-keys
```

It will take several minutes to launch the cluster. Once launched, we will need to copy the configuration so that we can interact with the cluster by using our local copy of `kubectl`. To do this, run the following command:

```
$ az aks get-credentials \
    --resource-group MasteringDockerAKS \
    --name MasteringDockerAKSCluster
```

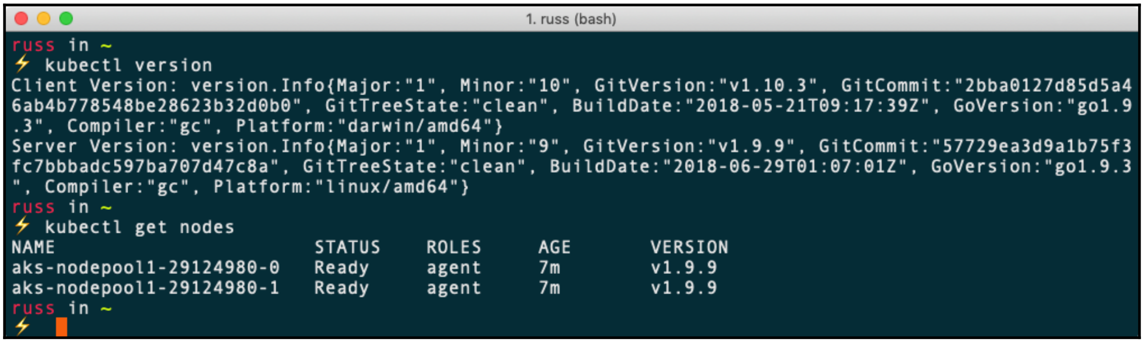
This will configure your local copy of `kubectl` to talk to the AKS cluster you have just launched. You should now see the cluster listed in the Docker menu under Kubernetes:



Running the following commands will show you the version of the server that your `kubectl` client is talking to as well as details regarding the nodes:

```
$ kubectl version
$ kubectl get nodes
```

You can see the output of the preceding commands in the following screenshot:



```
russ in ~
⚡ kubectl version
Client Version: version.Info{Major:"1", Minor:"10", GitVersion:"v1.10.3", GitCommit:"2bba0127d85d5a4
6ab4b778548be28623b32d0b0", GitTreeState:"clean", BuildDate:"2018-05-21T09:17:39Z", GoVersion:"go1.9
.3", Compiler:"gc", Platform:"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"9", GitVersion:"v1.9.9", GitCommit:"57729ea3d9a1b75f3
fc7bbbadc597ba707d47c8a", GitTreeState:"clean", BuildDate:"2018-06-29T01:07:01Z", GoVersion:"go1.9.3
", Compiler:"gc", Platform:"linux/amd64"}
russ in ~
⚡ kubectl get nodes
NAME                        STATUS    ROLES     AGE       VERSION
aks-nodepool1-29124980-0    Ready     agent     7m        v1.9.9
aks-nodepool1-29124980-1    Ready     agent     7m        v1.9.9
russ in ~
⚡
```

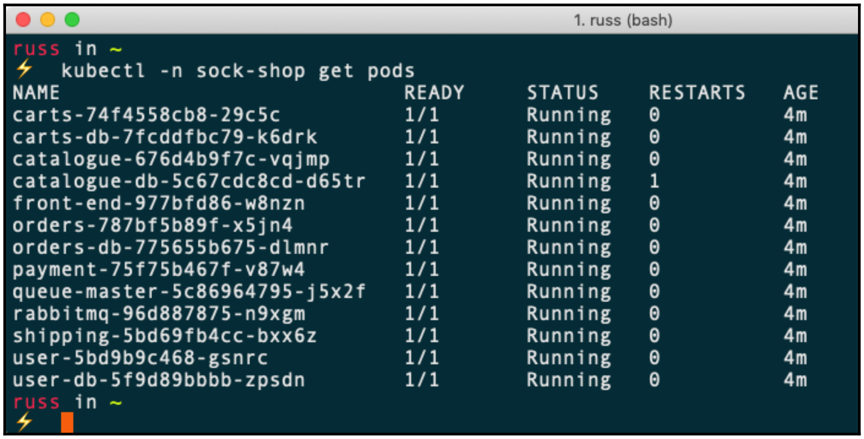
Now that we have our cluster up and running, we need to launch something. Luckily, there is an excellent open source microservices demo from Weave which launches a demo shop that sell socks. To launch the demo, we simply need to run the following commands:

```
$ kubectl create namespace sock-shop
$ kubectl apply -n sock-shop -f
"https://github.com/microservices-demo/microservices-demo/blob/master/deplo
y/kubernetes/complete-demo.yaml?raw=true"
```

It will take about five minutes for the demo to launch. You can check the status of the `pods` by running the following command:

```
$ kubectl -n sock-shop get pods
```

Once everything is up and running, you should see something like the following output:



```
russ in ~
⚡  kubectl -n sock-shop get pods
NAME                            READY     STATUS    RESTARTS   AGE
carts-74f4558cb8-29c5c          1/1       Running   0          4m
carts-db-7fcddfbc79-k6drk       1/1       Running   0          4m
catalogue-676d4b9f7c-vqjmp      1/1       Running   0          4m
catalogue-db-5c67cdc8cd-d65tr   1/1       Running   1          4m
front-end-977bfd86-w8nzn        1/1       Running   0          4m
orders-787bf5b89f-x5jn4         1/1       Running   0          4m
orders-db-775655b675-dlmnr      1/1       Running   0          4m
payment-75f75b467f-v87w4        1/1       Running   0          4m
queue-master-5c86964795-j5x2f   1/1       Running   0          4m
rabbitmq-96d887875-n9xgm        1/1       Running   0          4m
shipping-5bd69fb4cc-bxx6z       1/1       Running   0          4m
user-5bd9b9c468-gsnrc           1/1       Running   0          4m
user-db-5f9d89bbbb-zpsdn        1/1       Running   0          4m
russ in ~
⚡
```

Now that our application has launched, we need a way to access it. Check the services by running the following command:

```
$ kubectl -n sock-shop get services
```

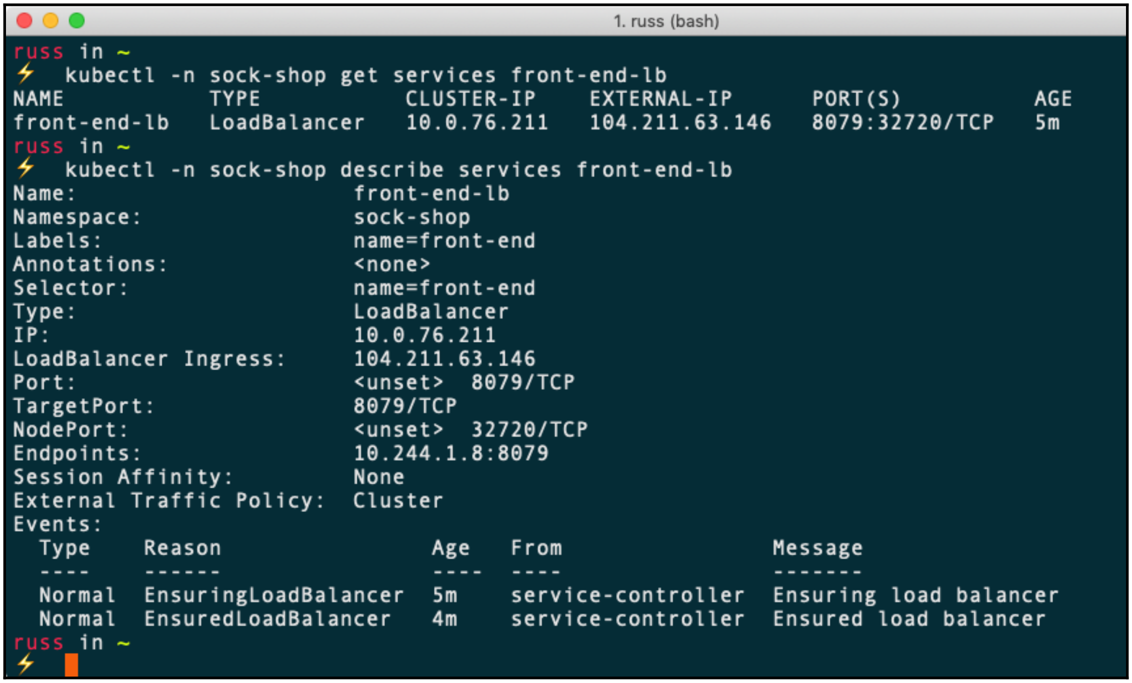
This shows us that there is a service called `front-end`. We are going to create a Load Balancer and attach it to this service. To do this, run the following command:

```
$ kubectl -n sock-shop expose deployment front-end --type=LoadBalancer --
name=front-end-lb
```

You can check the status of the Load Balancer by running the following commands:

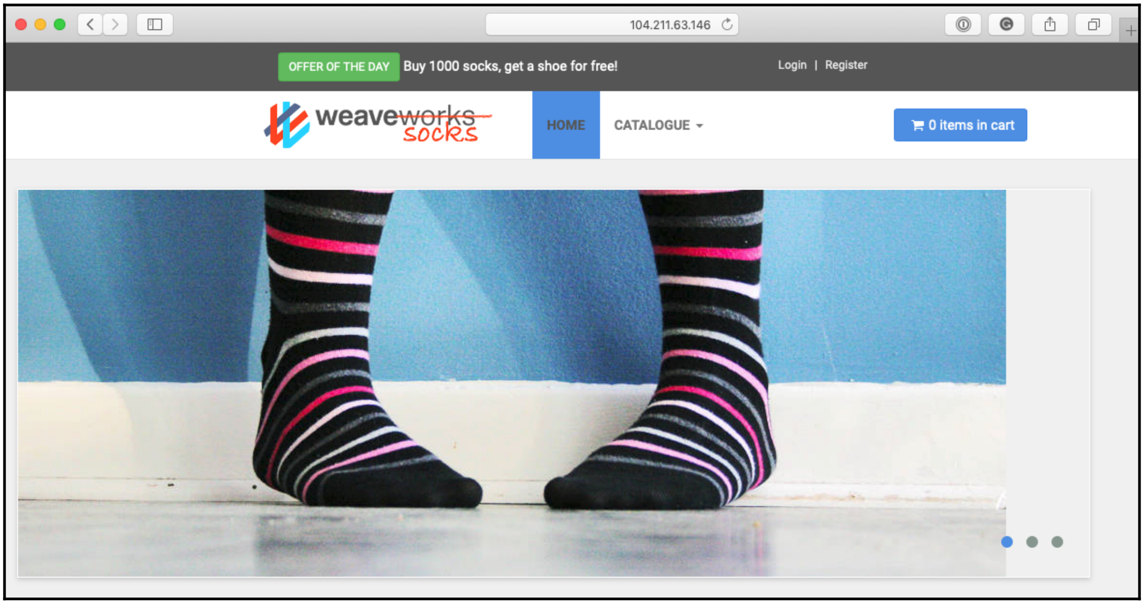
```
$ kubectl -n sock-shop get services front-end-lb
$ kubectl -n sock-shop describe services front-end-lb
```

Once launched, you should see something like the following:



```
russ in ~
⚡  kubectl -n sock-shop get services front-end-lb
NAME           TYPE           CLUSTER-IP    EXTERNAL-IP      PORT(S)          AGE
front-end-lb   LoadBalancer   10.0.76.211   104.211.63.146   8079:32720/TCP   5m
russ in ~
⚡  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:                       10.0.76.211
LoadBalancer Ingress:     104.211.63.146
Port:                     <unset>  8079/TCP
TargetPort:               8079/TCP
NodePort:                 <unset>  32720/TCP
Endpoints:                10.244.1.8:8079
Session Affinity:         None
External Traffic Policy:  Cluster
Events:
  Type    Reason                Age   From                Message
  ----    ------                ----  ----                -------
  Normal  EnsuringLoadBalancer  5m    service-controller  Ensuring load balancer
  Normal  EnsuredLoadBalancer   4m    service-controller  Ensured load balancer
russ in ~
⚡
```

As you can see from the preceding output, for my store, the IP address was `104.211.63.146` and the port was `8079`. Opening `http://104.211.63.146:8079/` in a browser presented me with the following page:



Buy 1000 socks, get a shoe for free!

Once you have finished clicking around the store, you can remove it by running the following command:

```
$ kubectl delete namespace sock-shop
```

To remove the AKS cluster and resource group, run the following commands:

```
$ az group delete --name MasteringDockerAKS --yes --no-wait
```

Remember to check that everything has been removed from the Azure portal as expected to avoid any unexpected charges. Finally, you can remove the configuration from your local `kubectl` configuration by running the following:

```
$ kubectl config delete-cluster MasteringDockerAKSCluster
$ kubectl config delete-context MasteringDockerAKSCluster
```

Next up, we are going to look at launching a similar cluster in Google Cloud.

Google Kubernetes Engine

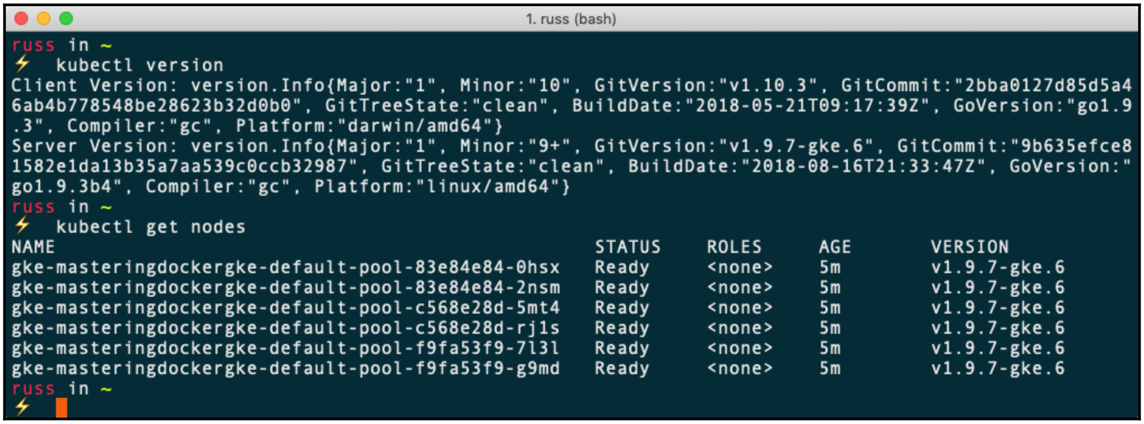
The **Google Kubernetes Engine**, as you may have already guessed, is very tightly integrated into Google's Cloud platform. Rather than going into more detail, let's dive straight in and launch a cluster. I am assuming that you already have a Google Cloud account, a project with billing enabled, and finally the Google Cloud SDK installed and configured to interact with your project.

To launch the cluster, simply run the following command:

```
$ gcloud container clusters create masteringdockergke --num-nodes=2
```

Once the cluster has been launched, your `kubectl` config will be automatically updated and the context will be set for the newly launched cluster. You can view information on the nodes by running the following:

```
$ kubectl version
$ kubectl get nodes
```



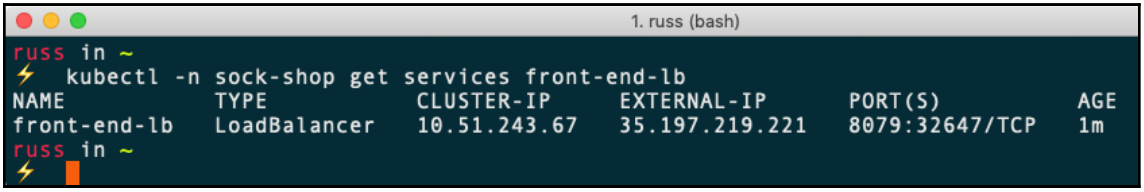
1. russ (bash)

```
russ in ~
⚡  kubectl version
Client Version: version.Info{Major:"1", Minor:"10", GitVersion:"v1.10.3", GitCommit:"2bba0127d85d5a4
6ab4b778548be28623b32d0b0", GitTreeState:"clean", BuildDate:"2018-05-21T09:17:39Z", GoVersion:"go1.9
.3", Compiler:"gc", Platform:"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"9+", GitVersion:"v1.9.7-gke.6", GitCommit:"9b635efce8
1582e1da13b35a7aa539c0ccb32987", GitTreeState:"clean", BuildDate:"2018-08-16T21:33:47Z", GoVersion:"
go1.9.3b4", Compiler:"gc", Platform:"linux/amd64"}
russ in ~
⚡  kubectl get nodes
NAME                                              STATUS    ROLES     AGE       VERSION
gke-masteringdockergke-default-pool-83e84e84-0hsx   Ready     <none>    5m        v1.9.7-gke.6
gke-masteringdockergke-default-pool-83e84e84-2nsm   Ready     <none>    5m        v1.9.7-gke.6
gke-masteringdockergke-default-pool-c568e28d-5mt4   Ready     <none>    5m        v1.9.7-gke.6
gke-masteringdockergke-default-pool-c568e28d-rj1s   Ready     <none>    5m        v1.9.7-gke.6
gke-masteringdockergke-default-pool-f9fa53f9-713l   Ready     <none>    5m        v1.9.7-gke.6
gke-masteringdockergke-default-pool-f9fa53f9-g9md   Ready     <none>    5m        v1.9.7-gke.6
russ in ~
⚡
```

Now that we have our cluster up and running, let's launch the demo shop by repeating the commands we used last time:

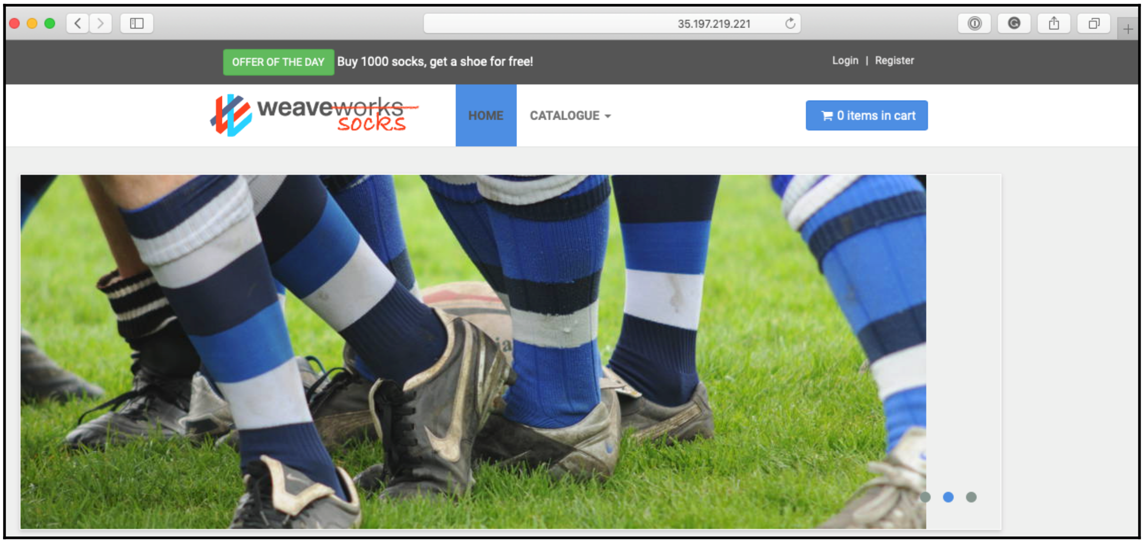
```
$ kubectl create namespace sock-shop
$ kubectl apply -n sock-shop -f
"https://github.com/microservices-demo/microservices-demo/blob/master/deplo
y/kubernetes/complete-demo.yaml?raw=true"
$ kubectl -n sock-shop get pods
$ kubectl -n sock-shop get services
$ kubectl -n sock-shop expose deployment front-end --type=LoadBalancer --
name=front-end-lb
$ kubectl -n sock-shop get services front-end-lb
```

Again, once the `front-end-lb` service has been created, you should be able to find the external IP address port to use:



```
russ in ~
⚡  kubectl -n sock-shop get services front-end-lb
NAME           TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)          AGE
front-end-lb   LoadBalancer   10.51.243.67   35.197.219.221   8079:32647/TCP   1m
russ in ~
⚡
```

Entering these into a browser will open the store:



To remove the cluster, simply run the following:

```
$ kubectl delete namespace sock-shop
$ gcloud container clusters delete masteringdockergke
```

This will also remove the context and cluster from `kubectl`.

Amazon Elastic Container Service for Kubernetes

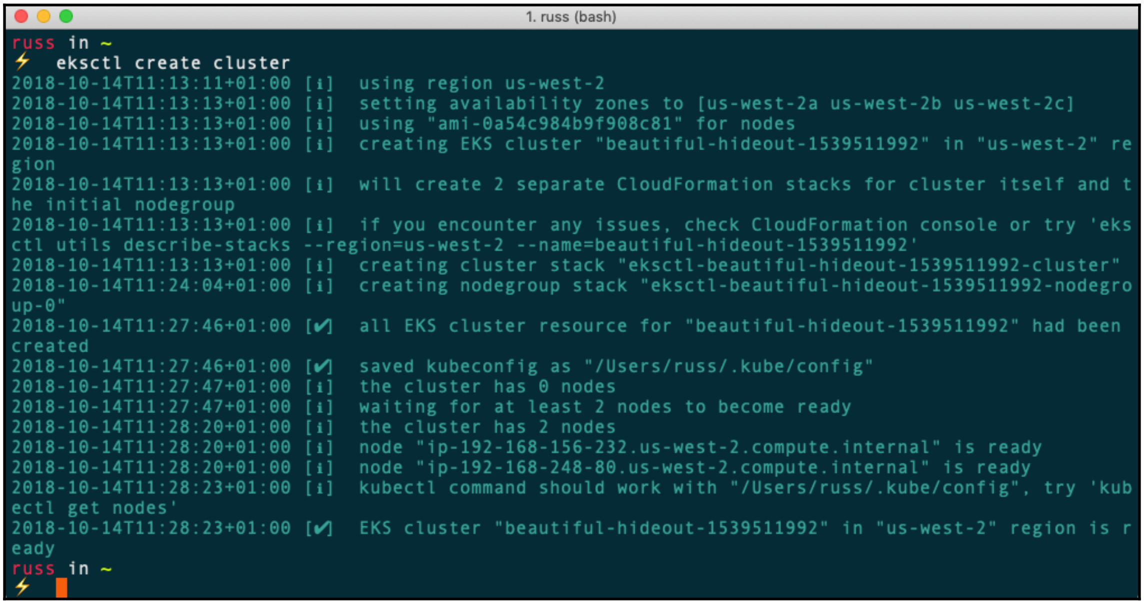
The final Kubernetes service we are going to take a look at is the **Amazon Elastic Container Service for Kubernetes**, or **Amazon EKS**, for short. This is the most recently launched service of the three services we are covering. In fact, you could say that Amazon was very late to the Kubernetes party.

Unfortunately, the command-line tools for Amazon are not as friendly as the ones we used for Microsoft Azure and Google Cloud. Because of this, I am 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. You can find details on `eksctl` and also the Amazon command-line tools in the *Further reading* section at the end of this chapter.

To launch our Amazon EKS cluster, we need to run the following command:

```
$ eksctl create cluster
```

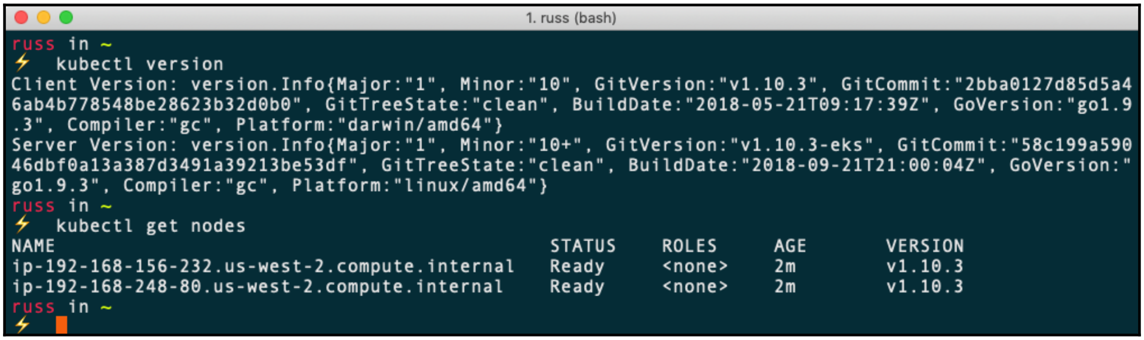
It will take several minutes to launch the cluster, but you will receive feedback on the command line throughout the process. Also, as `eksctl` is using CloudFormation, you can also check its progress in the AWS Console. Once complete, you should see something like the following output:



```
russ in ~
⚡  eksctl create cluster
2018-10-14T11:13:11+01:00 [ℹ]  using region us-west-2
2018-10-14T11:13:13+01:00 [ℹ]  setting availability zones to [us-west-2a us-west-2b us-west-2c]
2018-10-14T11:13:13+01:00 [ℹ]  using "ami-0a54c984b9f908c81" for nodes
2018-10-14T11:13:13+01:00 [ℹ]  creating EKS cluster "beautiful-hideout-1539511992" in "us-west-2" re
gion
2018-10-14T11:13:13+01:00 [ℹ]  will create 2 separate CloudFormation stacks for cluster itself and t
he initial nodegroup
2018-10-14T11:13:13+01:00 [ℹ]  if you encounter any issues, check CloudFormation console or try 'eks
ctl utils describe-stacks --region=us-west-2 --name=beautiful-hideout-1539511992'
2018-10-14T11:13:13+01:00 [ℹ]  creating cluster stack "eksctl-beautiful-hideout-1539511992-cluster"
2018-10-14T11:24:04+01:00 [ℹ]  creating nodegroup stack "eksctl-beautiful-hideout-1539511992-nodegro
up-0"
2018-10-14T11:27:46+01:00 [✔]  all EKS cluster resource for "beautiful-hideout-1539511992" had been
created
2018-10-14T11:27:46+01:00 [✔]  saved kubeconfig as "/Users/russ/.kube/config"
2018-10-14T11:27:47+01:00 [ℹ]  the cluster has 0 nodes
2018-10-14T11:27:47+01:00 [ℹ]  waiting for at least 2 nodes to become ready
2018-10-14T11:28:20+01:00 [ℹ]  the cluster has 2 nodes
2018-10-14T11:28:20+01:00 [ℹ]  node "ip-192-168-156-232.us-west-2.compute.internal" is ready
2018-10-14T11:28:20+01:00 [ℹ]  node "ip-192-168-248-80.us-west-2.compute.internal" is ready
2018-10-14T11:28:23+01:00 [ℹ]  kubectl command should work with "/Users/russ/.kube/config", try 'kub
ectl get nodes'
2018-10-14T11:28:23+01:00 [✔]  EKS cluster "beautiful-hideout-1539511992" in "us-west-2" region is r
eady
russ in ~
⚡
```

As part of the launch, `eksctl` will have configured your local `kubectl` context, meaning that you can run the following:

```
$ kubectl version
$ kubectl get nodes
```

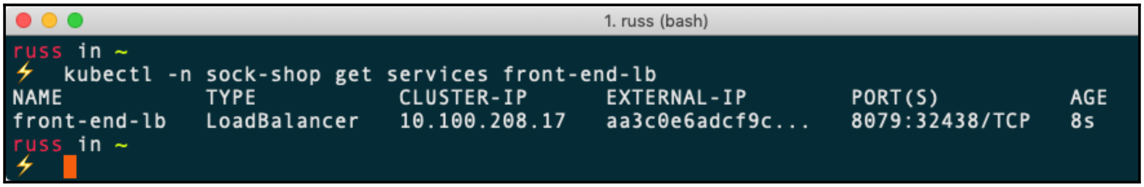


```
russ in ~
⚡  kubectl version
Client Version: version.Info{Major:"1", Minor:"10", GitVersion:"v1.10.3", GitCommit:"2bba0127d85d5a4
6ab4b778548be28623b32d0b0", GitTreeState:"clean", BuildDate:"2018-05-21T09:17:39Z", GoVersion:"go1.9
.3", Compiler:"gc", Platform:"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"10+", GitVersion:"v1.10.3-eks", GitCommit:"58c199a590
46dbf0a13a387d3491a39213be53df", GitTreeState:"clean", BuildDate:"2018-09-21T21:00:04Z", GoVersion:"
go1.9.3", Compiler:"gc", Platform:"linux/amd64"}
russ in ~
⚡  kubectl get nodes
NAME                                            STATUS    ROLES     AGE       VERSION
ip-192-168-156-232.us-west-2.compute.internal   Ready     <none>    2m        v1.10.3
ip-192-168-248-80.us-west-2.compute.internal    Ready     <none>    2m        v1.10.3
russ in ~
⚡
```

Now that we have the cluster up and running, we can launch the demo store, just like we did previously:

```
$ kubectl create namespace sock-shop
$ kubectl apply -n sock-shop -f
"https://github.com/microservices-demo/microservices-demo/blob/master/deplo
y/kubernetes/complete-demo.yaml?raw=true"
$ kubectl -n sock-shop get pods
$ kubectl -n sock-shop get services
$ kubectl -n sock-shop expose deployment front-end --type=LoadBalancer --
name=front-end-lb
$ kubectl -n sock-shop get services front-end-lb
```

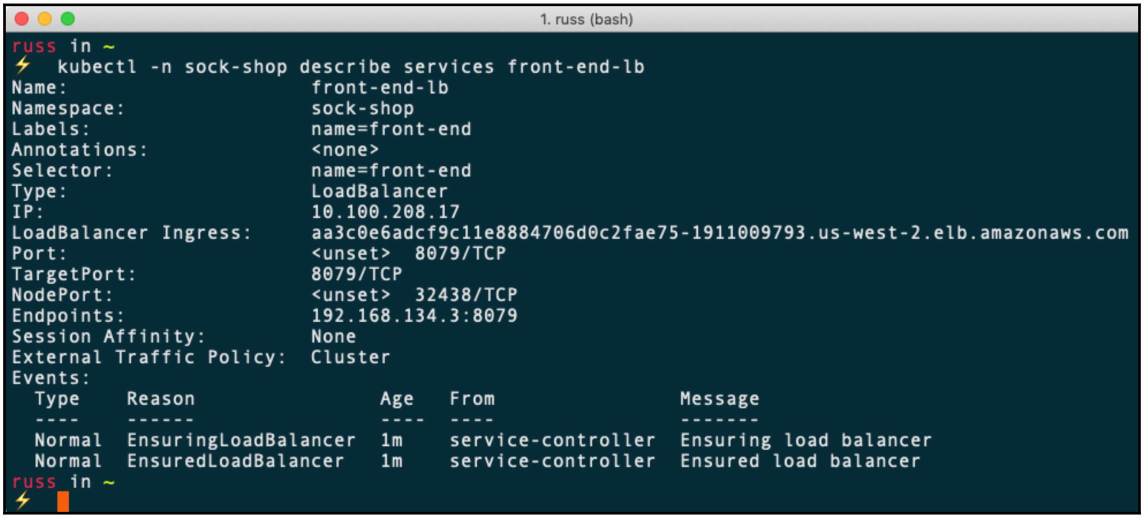
You may notice that the external IP that's listed when running that last command looks a little strange:



```
russ in ~
⚡  kubectl -n sock-shop get services front-end-lb
NAME           TYPE           CLUSTER-IP      EXTERNAL-IP        PORT(S)          AGE
front-end-lb   LoadBalancer   10.100.208.17   aa3c0e6adcf9c...   8079:32438/TCP   8s
russ in ~
⚡
```

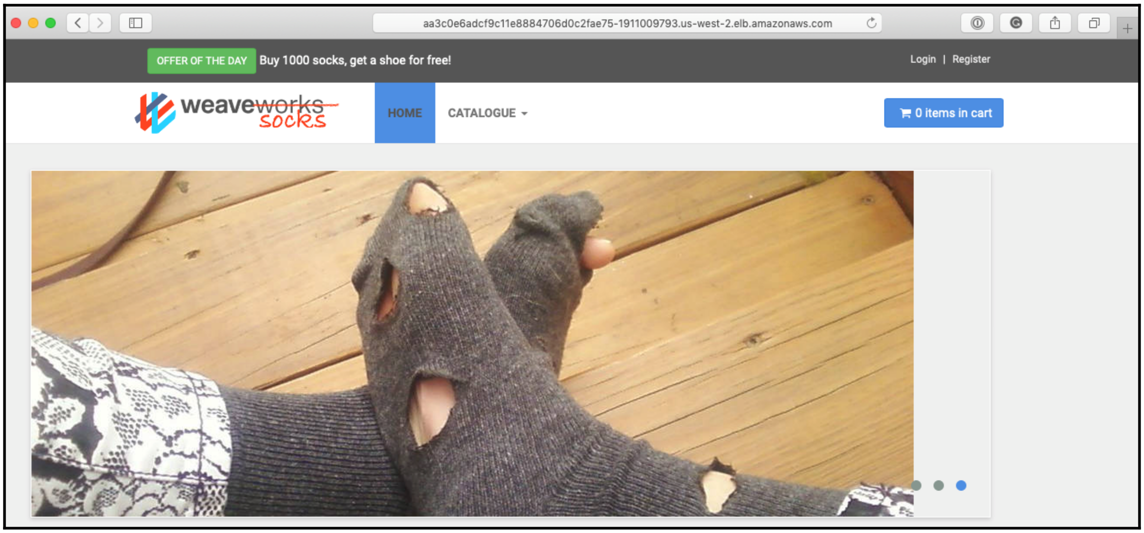
That is because it is a DNS name rather than an IP address. To find the full URL, you can run the following command:

```
$ kubectl -n sock-shop describe services front-end-lb
```



```
russ in ~
⚡  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:                       10.100.208.17
LoadBalancer Ingress:     aa3c0e6adcf9c11e8884706d0c2fae75-1911009793.us-west-2.elb.amazonaws.com
Port:                     <unset>  8079/TCP
TargetPort:               8079/TCP
NodePort:                 <unset>  32438/TCP
Endpoints:                192.168.134.3:8079
Session Affinity:         None
External Traffic Policy:  Cluster
Events:
  Type    Reason                Age   From                Message
  ----    ------                ----  ----                -------
  Normal  EnsuringLoadBalancer  1m    service-controller  Ensuring load balancer
  Normal  EnsuredLoadBalancer   1m    service-controller  Ensured load balancer
russ in ~
⚡  
```

Entering the URL and porting into a browser will, as you might have guessed, show the demo store:



aa3c0e6adcf9c11e8884706d0c2fae75-1911009793.us-west-2.elb.amazonaws.com

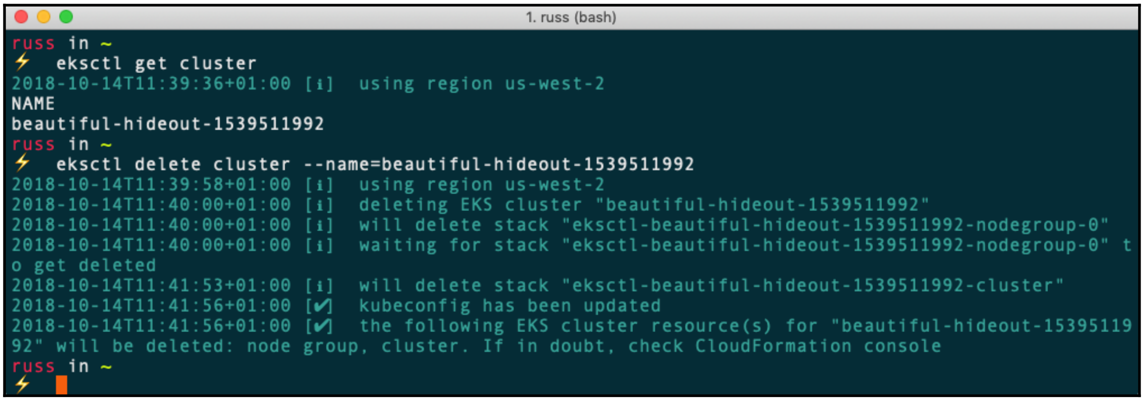
To remove the cluster, run the following commands:

```
$ kubectl delete namespace sock-shop
$ eksctl get cluster
```

This will return the names of the clusters that are running. Once you have the name, run the following command, making sure to reference your own cluster:

```
$ eksctl delete cluster --name=beautiful-hideout-1539511992
```

Your terminal output should look as follows:



```
russ in ~
⚡  eksctl get cluster
2018-10-14T11:39:36+01:00 [ℹ]  using region us-west-2
NAME
beautiful-hideout-1539511992
russ in ~
⚡  eksctl delete cluster --name=beautiful-hideout-1539511992
2018-10-14T11:39:58+01:00 [ℹ]  using region us-west-2
2018-10-14T11:40:00+01:00 [ℹ]  deleting EKS cluster "beautiful-hideout-1539511992"
2018-10-14T11:40:00+01:00 [ℹ]  will delete stack "eksctl-beautiful-hideout-1539511992-nodegroup-0"
2018-10-14T11:40:00+01:00 [ℹ]  waiting for stack "eksctl-beautiful-hideout-1539511992-nodegroup-0" t
o get deleted
2018-10-14T11:41:53+01:00 [ℹ]  will delete stack "eksctl-beautiful-hideout-1539511992-cluster"
2018-10-14T11:41:56+01:00 [✔]  kubeconfig has been updated
2018-10-14T11:41:56+01:00 [✔]  the following EKS cluster resource(s) for "beautiful-hideout-15395119
92" will be deleted: node group, cluster. If in doubt, check CloudFormation console
russ in ~
⚡
```

Kubernetes summary

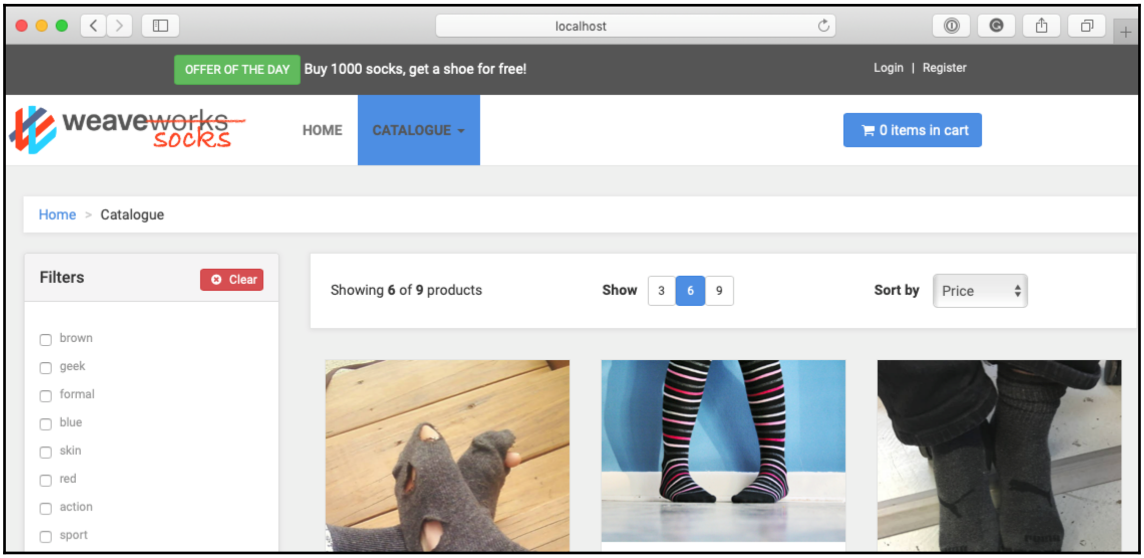
This concludes our brief look at Kubernetes in Microsoft Azure, Google Cloud, and Amazon Web Services. We covered a few interesting points here. The first is that we managed to launch and manage our clusters using the command line with a few simple steps, although we did have to use a third-party tool for Amazon EKS.

The second and most important point is that once we had access to the cluster using `kubectl`, the experience was exactly the same across all three platforms. At no point did we have to access the cloud provider's web-based control panel to tweak or review a setting. Everything was done using the same commands; deploying the same code and services was done with no thought or consideration on our part regarding any of the individual services offered by the cloud providers.

We can even run the demo store locally using Docker, with exactly the same commands. Just start your Kubernetes cluster up, make sure that you have the local Docker context selected, and then run the following commands:

```
$ kubectl create namespace sock-shop
$ kubectl apply -n sock-shop -f
"https://github.com/microservices-demo/microservices-demo/blob/master/deplo
y/kubernetes/complete-demo.yaml?raw=true"
$ kubectl -n sock-shop get pods
$ kubectl -n sock-shop get services
$ kubectl -n sock-shop expose deployment front-end --type=LoadBalancer --
name=front-end-lb
$ kubectl -n sock-shop get services front-end-lb
```

As you can see from the following output, the *load balanced* IP, in this case, is `localhost`. Opening your browser and entering `http://localhost:8079` will take you to the store:



Buy 1000 socks, get a shoe for free!

Showing **6** of **9** products

You can remove the store by running the following command:

```
$ kubectl delete namespace sock-shop
```

This level of consistency across multiple providers and even local machines hasn't really been achievable before without a lot of work and configuration or via a closed source subscription-based service.

Summary

In this chapter, we have taken a look at how we can deploy Docker Swarm clusters into a cloud provider using the tools provided by Docker themselves. We have also taken a look at two of the services offered by public clouds to run containers away from the core Docker toolset.

Finally, we looked at launching Kubernetes clusters in various clouds and running the same demo application in all of them. While it was clear from any of the commands we ran, all three of the public clouds were using various versions of Docker as the container engine. Though this could be subject to change by the time you read this, as in theory, they could switch to another engine with little impact.

In the next chapter, we are going to move back to working Docker and take a look at Portainer, a web-based interface for managing your Docker installation.

Questions

1. True or false: Docker for AWS and Docker for Azure launches Kubernetes clusters for you to launch your containers on.
2. What Amazon service don't you have to directly manage if you're using Amazon Fargate?
3. What type of application do we need to launch in Azure?
4. Once launched, what is the command we need to run to create the namespace for the Sock Shop store?
5. How do you find out full details about the Load Balancer?

Further reading

You can find details of the Docker Cloud service closing down at the following links:

- Docker Cloud Migration Notification and FAQs: `https://success.docker.com/article/cloud-migration`
- Stuck! Docker Cloud Shutdown!: `https://blog.cloud66.com/stuck-docker-cloud-shutdown/`

More details on the templating services using by Docker for AWS and Docker for Azure can be found at the following links:

- AWS CloudFormation: `https://aws.amazon.com/cloudformation/`
- Azure ARM templates: `https://azure.microsoft.com/en-gb/resources/templates/`
- ARM template Visualizer: `http://armviz.io/`

The cloud services we used to launch containers can be found at the following links:

- Amazon ECS: `https://aws.amazon.com/ecs/`
- AWS Fargate: `https://aws.amazon.com/fargate/`
- Azure Web Apps: `https://azure.microsoft.com/en-gb/services/app-service/web/`

The three 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/`

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 – a CLI for Amazon EKS: `https://eksctl.io/`

Finally, for more details on the demo store, go to the following link:

- Sock Shop: `https://microservices-demo.github.io`

11
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. The topics that will be covered are as follows:

- The road to Portainer
- Getting Portainer up and running
- Using 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. Towards the end of the chapter, we will use Docker Machine and VirtualBox 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 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 system.

Check out the following video to see the Code in Action:
`http://bit.ly/2yWAdQV`

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.

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 is now deprecated and that Portainer should be used.

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:

1. First of all, to grab the container image from the Docker Hub we just need to run the following commands:

```
$ docker image pull portainer/portainer
$ docker image ls
```

2. As you can see when we ran the `docker image ls` command, the Portainer image is only 58.7MB. To launch Portainer, we simply have to run the following command if you are running macOS or Linux:

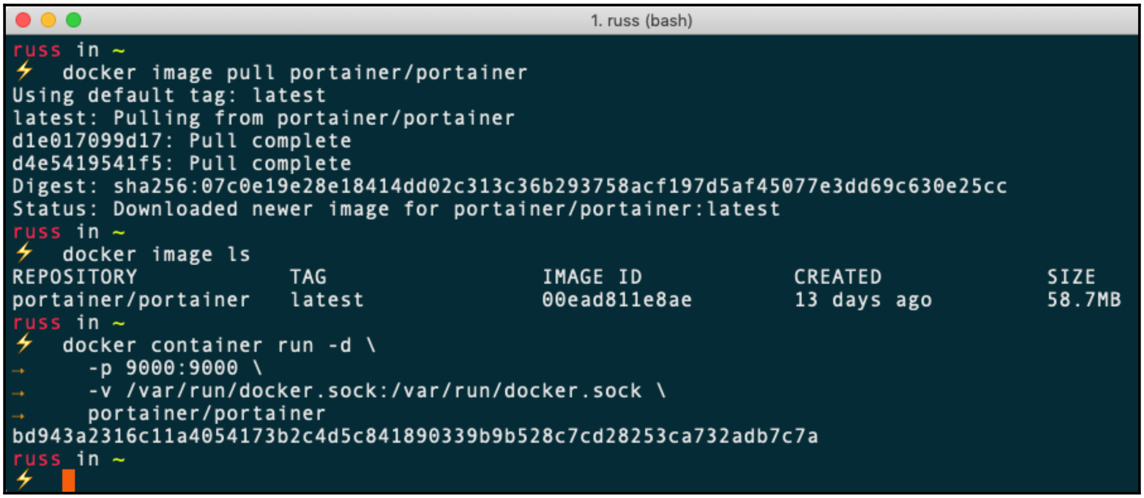
```
$ docker container run -d \
 -p 9000:9000 \
 -v /var/run/docker.sock:/var/run/docker.sock \
 portainer/portainer
```

3. 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 the Docker Engine on our Docker Host machine. Doing this will allow Portainer full unrestricted access to the 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 screenshot below shows this being executed on macOS:



1. russ (bash)

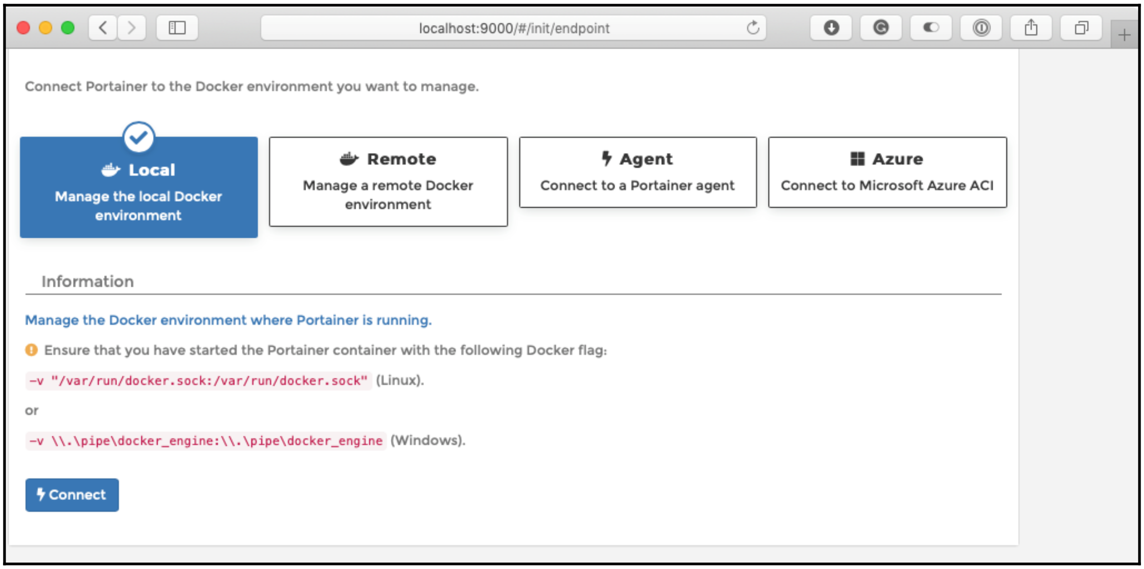
```
russ in ~
⚡  docker image pull portainer/portainer
Using default tag: latest
latest: Pulling from portainer/portainer
d1e017099d17: Pull complete
d4e5419541f5: Pull complete
Digest: sha256:07c0e19e28e18414dd02c313c36b293758acf197d5af45077e3dd69c630e25cc
Status: Downloaded newer image for portainer/portainer:latest
russ in ~
⚡  docker image ls
REPOSITORY            TAG                 IMAGE ID            CREATED             SIZE
portainer/portainer   latest              00ead811e8ae        13 days ago         58.7MB
russ in ~
⚡  docker container run -d \
→     -p 9000:9000 \
→     -v /var/run/docker.sock:/var/run/docker.sock \
→     portainer/portainer
bd943a2316c11a4054173b2c4d5c841890339b9b528c7cd28253ca732adb7c7a
russ in ~
⚡
```

4. 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.

5. 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 two options:
    - Manage the Docker instance where Portainer is running
    - Manage a remote Docker instance

For the moment, we want to manage the instance where Portainer is running, which is the Local option, rather than the default **Remote** one:



localhost:9000/#/init/endpoint

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).

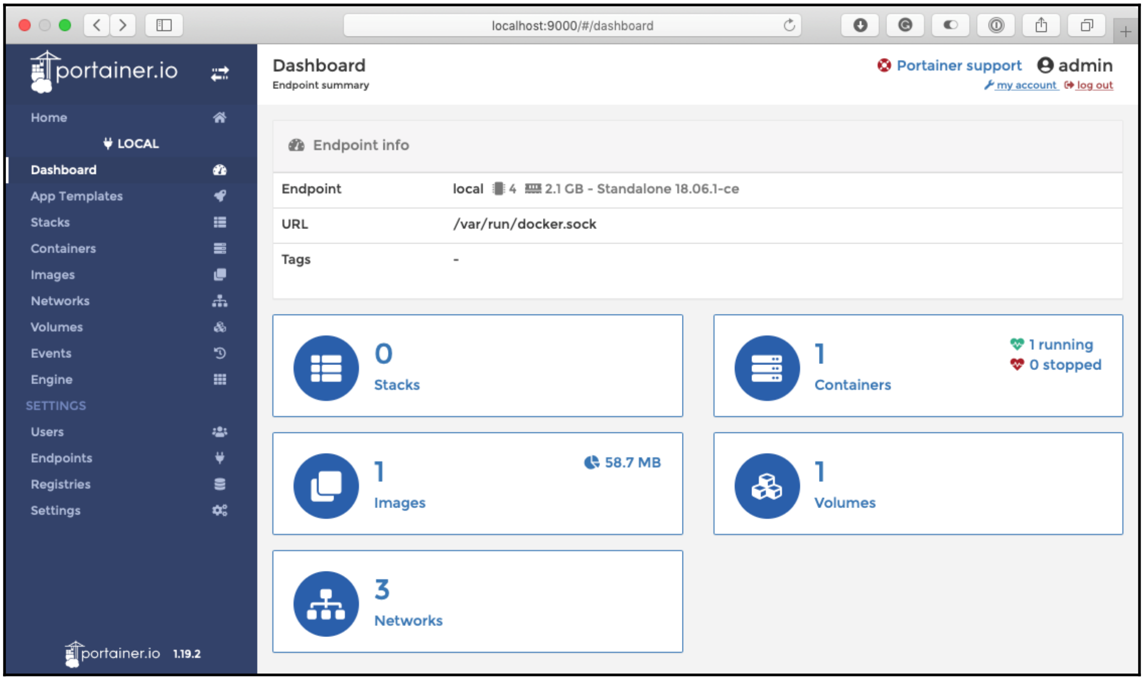
As we have already taken mounting the Docker socket file into account when launching our Portainer container, we can click on **Connect** to complete our installation. This will take us straight into Portainer itself, showing us the dashboard.

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.

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 4 2.1 GB - Standalone 18.06.1-ce
URL	/var/run/docker.sock
Tags	-

0 Stacks

1 Containers — 1 running, 0 stopped

1 Images — 58.7 MB

1 Volumes

3 Networks

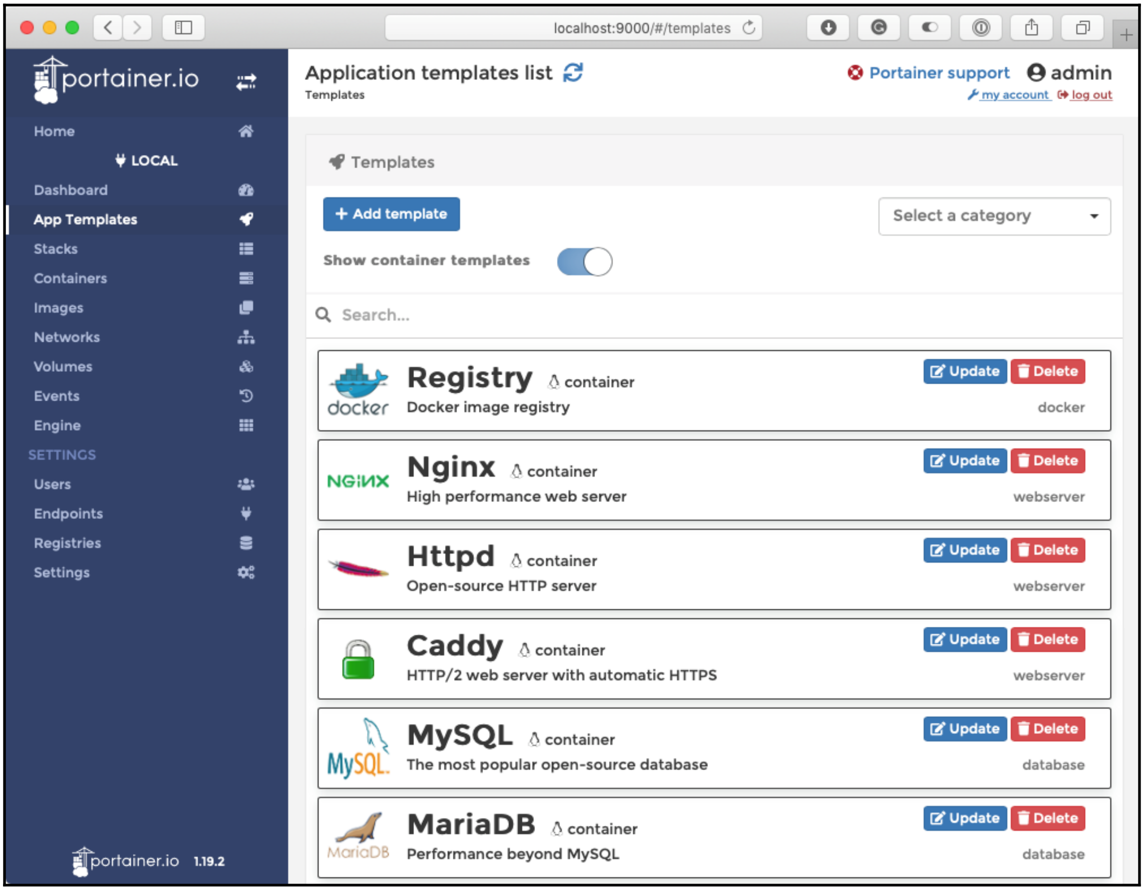
In my case, this shows how many **Containers** I have running, which at the moment is just the already running Portainer container, as well as the number of images I have downloaded. We can also see the number of **Volumes** and **Networks** available on the Docker instance, also it will show the number of running **Stacks**.

It also shows basic information on the Docker instance itself; as you can see, the Docker instance is running Moby Linux, has two CPUs and 2 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, 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 the Docker Hub:



Registry container
Docker image registry
docker

Nginx container
High performance web server
webserver

Httpd container
Open-source HTTP server
webserver

Caddy container
HTTP/2 web server with automatic HTTPS
webserver

MySQL container
The most popular open-source database
database

MariaDB container
Performance beyond MySQL
database

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:

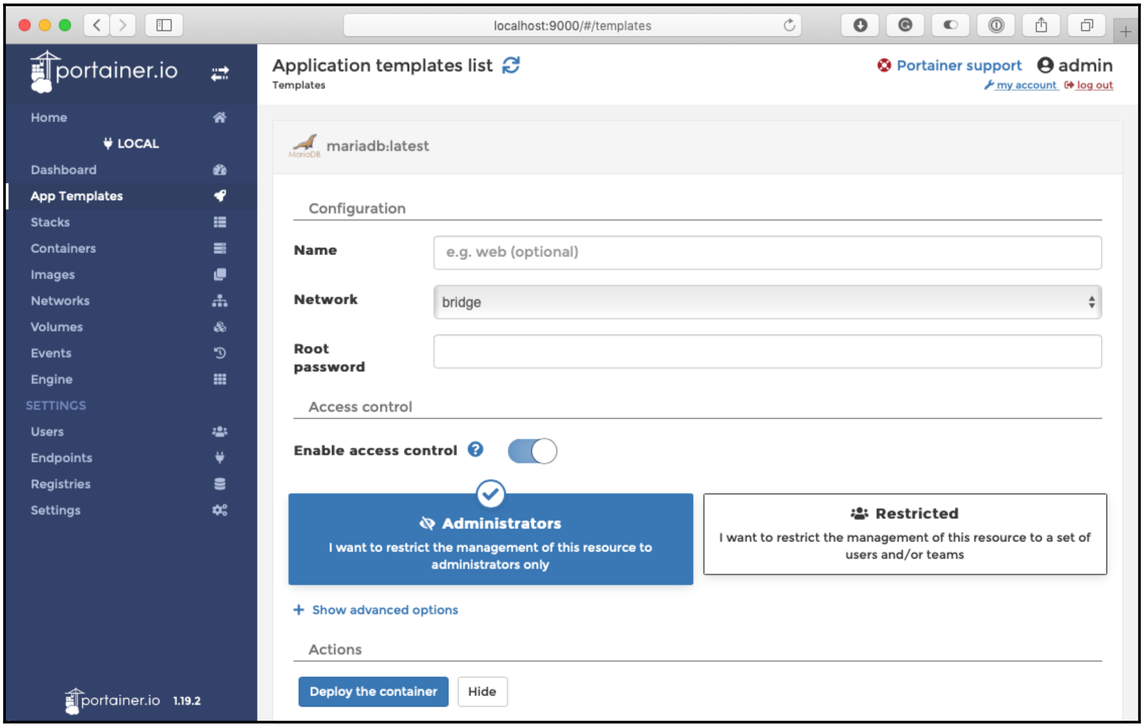
```
{
  "type": "container",
  "title": "Nginx",
  "description": "High performance web server",
  "categories": ["webserver"],
  "platform": "linux",
  "logo": "https://portainer.io/images/logos/nginx.png",
  "image": "nginx:latest",
  "ports": [
    "80/tcp",
    "443/tcp"
  ],
  "volumes": ["/etc/nginx", "/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 present the user with custom form fields that need to be completed before the container is launched, as demonstrated by the following screenshot:



Name: e.g. web (optional)

Network: bridge

Root password:

Enable access control: on

Administrators — I want to restrict the management of this resource to administrators only (selected)

Restricted — I want to restrict the management of this resource to a set of users and/or teams

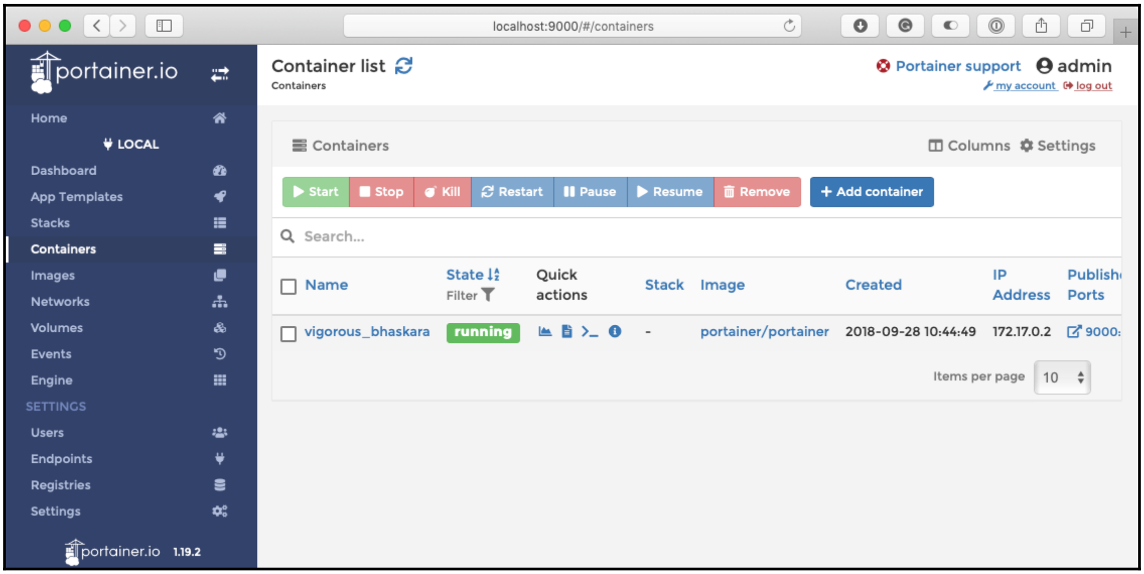
As you can see, we have a field where we can enter the root password we would like to use for our MariaDB container. Filling this in will take that value and pass it as an environment variable, building the following command to launch the container:

```
$ 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.



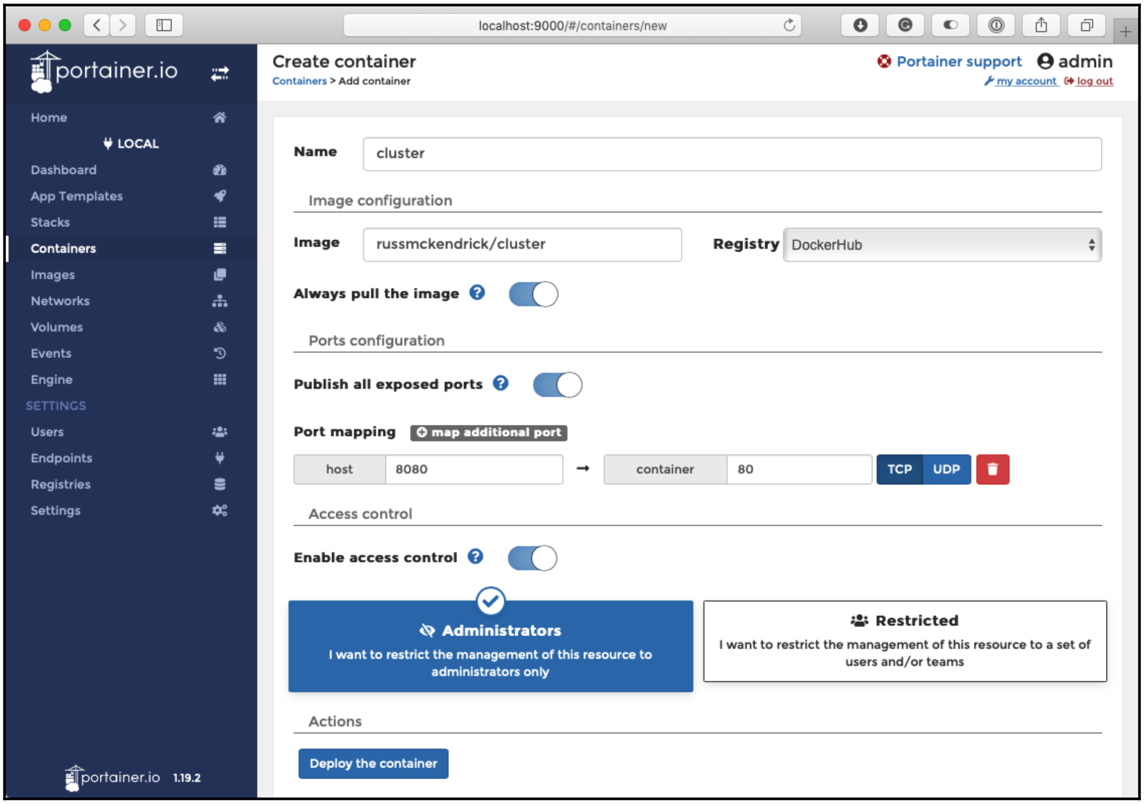
□ Name	State Filter	Quick actions	Stack	Image	Created	IP Address	Publish Ports
□ vigorous_bhaskara	running		-	portainer/portainer	2018-09-28 10:44:49	172.17.0.2	9000:

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 press 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 ports**: On

Finally, add a port mapping from port `8080` on the host to port `80` on the container by clicking on **+ map additional port**. Your completed form should look something like the following screenshot:



Create container

Containers > Add container

Name: cluster

Image configuration

Image: russmckendrick/cluster Registry: DockerHub

Always pull the image: on

Ports configuration

Publish all exposed ports: on

Port mapping

host 8080 → container 80 TCP UDP

Access control

Enable access control: on

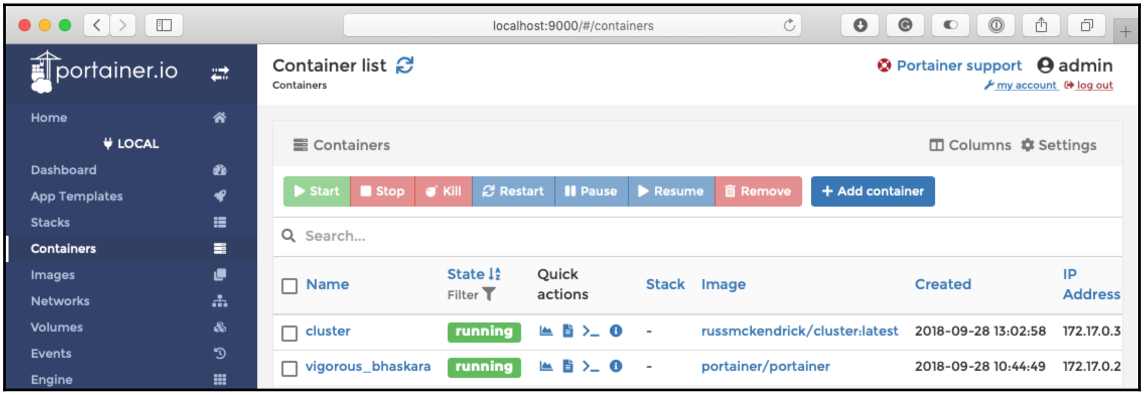
Administrators
I want to restrict the management of this resource to administrators only

Restricted
I want to restrict the management of this resource to a set of users and/or teams

Actions

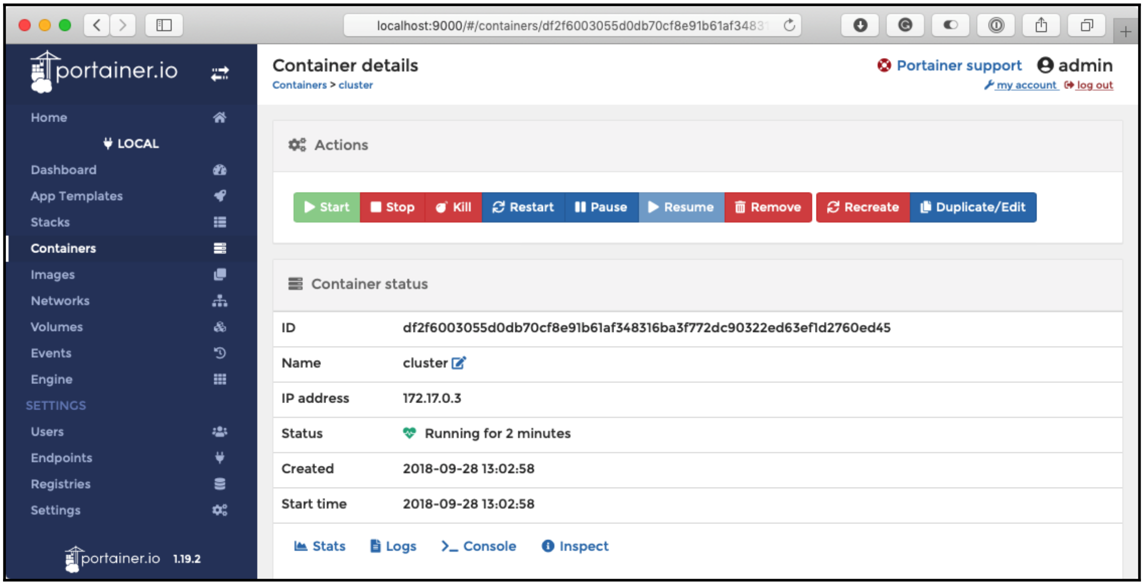
Deploy the container

Once that's done, click on **Deploy the container**, and after a few seconds, you will be returned the list of running containers, where you should see your newly launched container:



□ Name	State Filter	Quick actions	Stack	Image	Created	IP Address
□ cluster	running		-	russmckendrick/cluster:latest	2018-09-28 13:02:58	172.17.0.3
□ vigorous_bhaskara	running		-	portainer/portainer	2018-09-28 10:44:49	172.17.0.2

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	df2f6003055d0db70cf8e91b61af348316ba3f772dc90322ed63ef1d2760ed45
Name	cluster
IP address	172.17.0.3
Status	Running for 2 minutes
Created	2018-09-28 13:02:58
Start time	2018-09-28 13:02:58

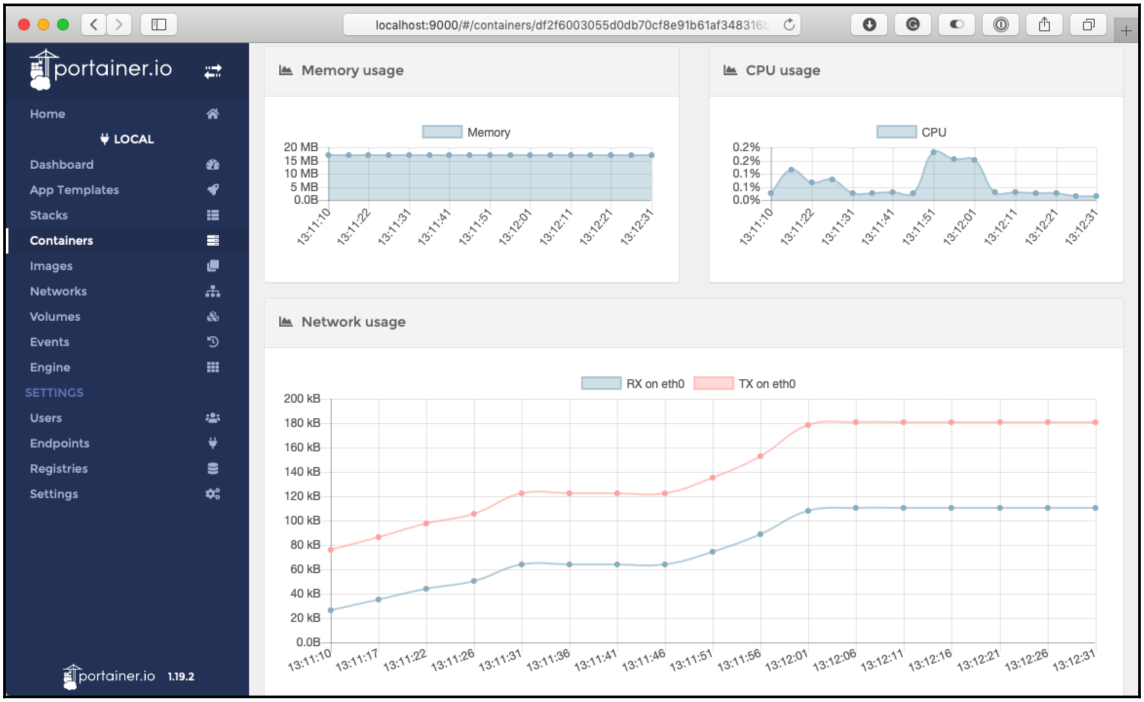
As you can see, the information about the container is the same information you would get if you were to run this command:

```
$ docker container inspect cluster
```

You can see the full output of this command by click on **Inspect**. You will also notice that there are buttons for **Stats**, **Logs**, and **Console**.

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:



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
```

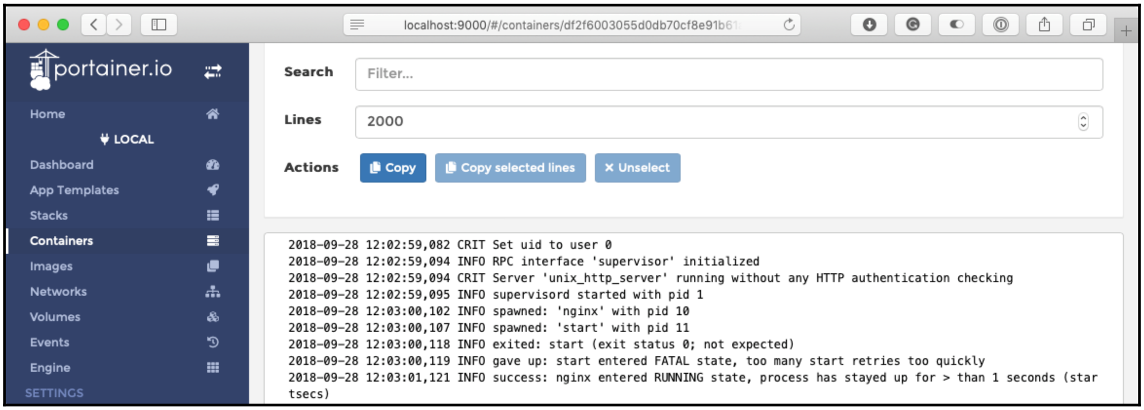
Each time the page is refreshed, the command is started from scratch as Portainer currently does not poll Docker in the background to keep a record of statistics for each of the running containers.

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: 2000

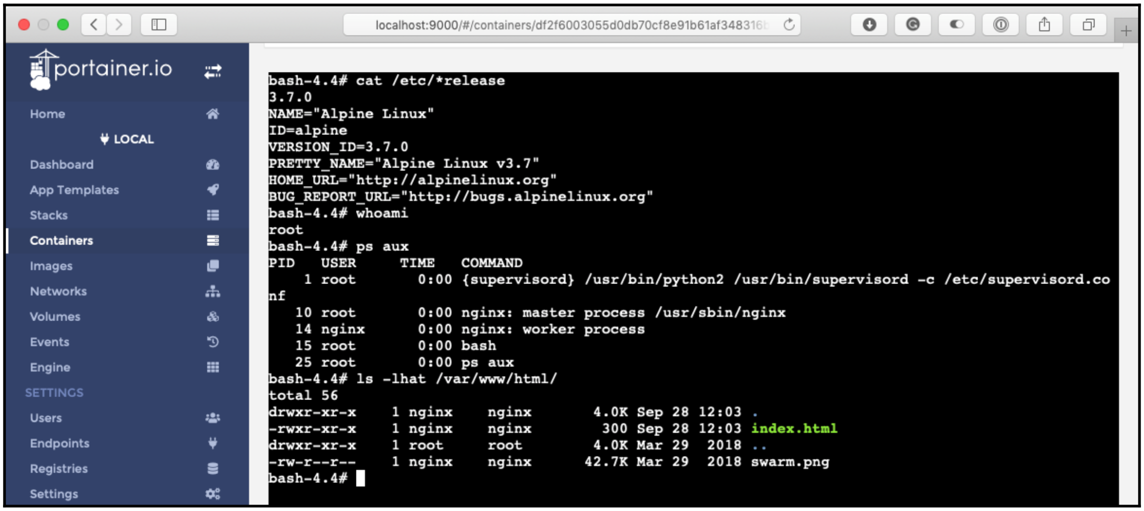
```
2018-09-28 12:02:59,082 CRIT Set uid to user 0
2018-09-28 12:02:59,094 INFO RPC interface 'supervisor' initialized
2018-09-28 12:02:59,094 CRIT Server 'unix_http_server' running without any HTTP authentication checking
2018-09-28 12:02:59,095 INFO supervisord started with pid 1
2018-09-28 12:03:00,102 INFO spawned: 'nginx' with pid 10
2018-09-28 12:03:00,107 INFO spawned: 'start' with pid 11
2018-09-28 12:03:00,118 INFO exited: start (exit status 0; not expected)
2018-09-28 12:03:00,119 INFO gave up: start entered FATAL state, too many start retries too quickly
2018-09-28 12:03:01,121 INFO success: nginx entered RUNNING state, process has stayed up for > than 1 seconds (star
tsecs)
```

You also have the option of adding timestamps to the output; this is the equivalent of running the following:

```
$ docker container logs --timestamps cluster
```

Console

Finally, 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`:



```
bash-4.4# cat /etc/*release
3.7.0
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.7.0
PRETTY_NAME="Alpine Linux v3.7"
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.co
nf
   10 root       0:00 nginx: master process /usr/sbin/nginx
   14 nginx      0:00 nginx: worker process
   15 root       0:00 bash
   25 root       0:00 ps aux
bash-4.4# ls -lhat /var/www/html/
total 56
drwxr-xr-x    1 nginx    nginx       4.0K Sep 28 12:03 .
-rwxr-xr-x    1 nginx    nginx        300 Sep 28 12:03 index.html
drwxr-xr-x    1 root     root        4.0K Mar 29  2018 ..
-rw-r--r--    1 nginx    nginx      42.7K Mar 29  2018 swarm.png
bash-4.4#
```

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 which is terminated once you disconnect from your shell session.

Images

Next up in the left-hand menu is **Images**. From here, you can manage, download, and upload images:

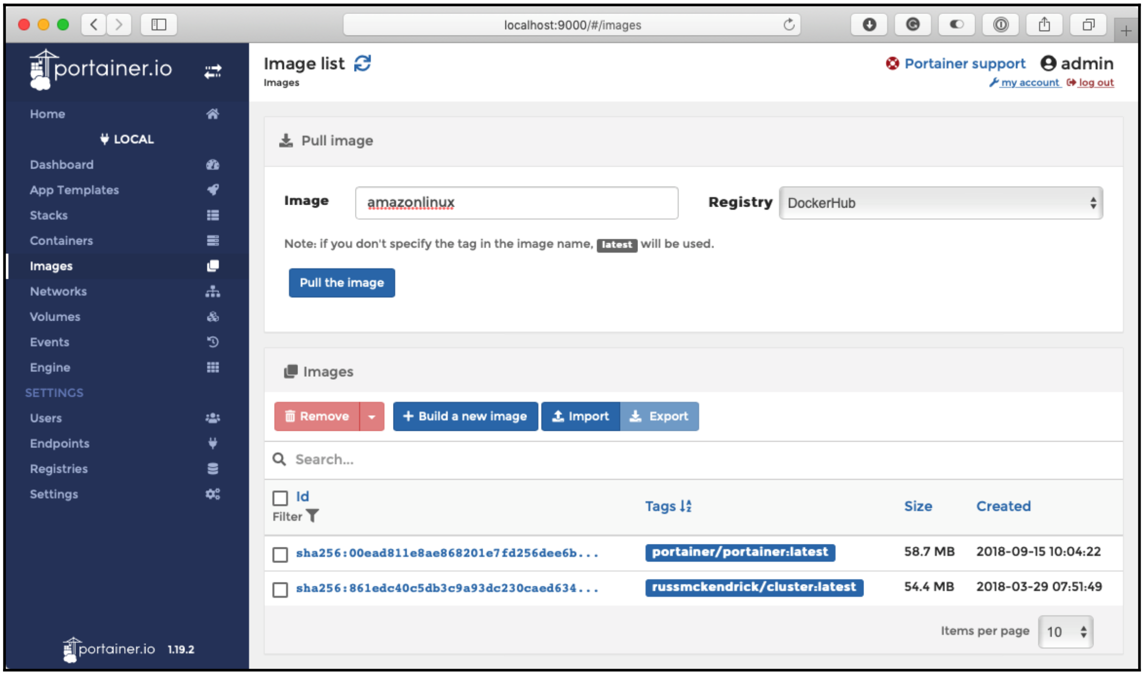


Image list
Images

Pull image

Image: amazonlinux Registry: DockerHub

Note: if you don't specify the tag in the image name, latest will be used.

Images

□ Id Filter	Tags	Size	Created
□ sha256:00ead811e8ae868201e7fd256dee6b...	portainer/portainer:latest	58.7 MB	2018-09-15 10:04:22
□ sha256:861edc40c5db3c9a93dc230caed634...	russmckendrick/cluster:latest	54.4 MB	2018-03-29 07:51:49

Items per page 10

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** 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
```

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:

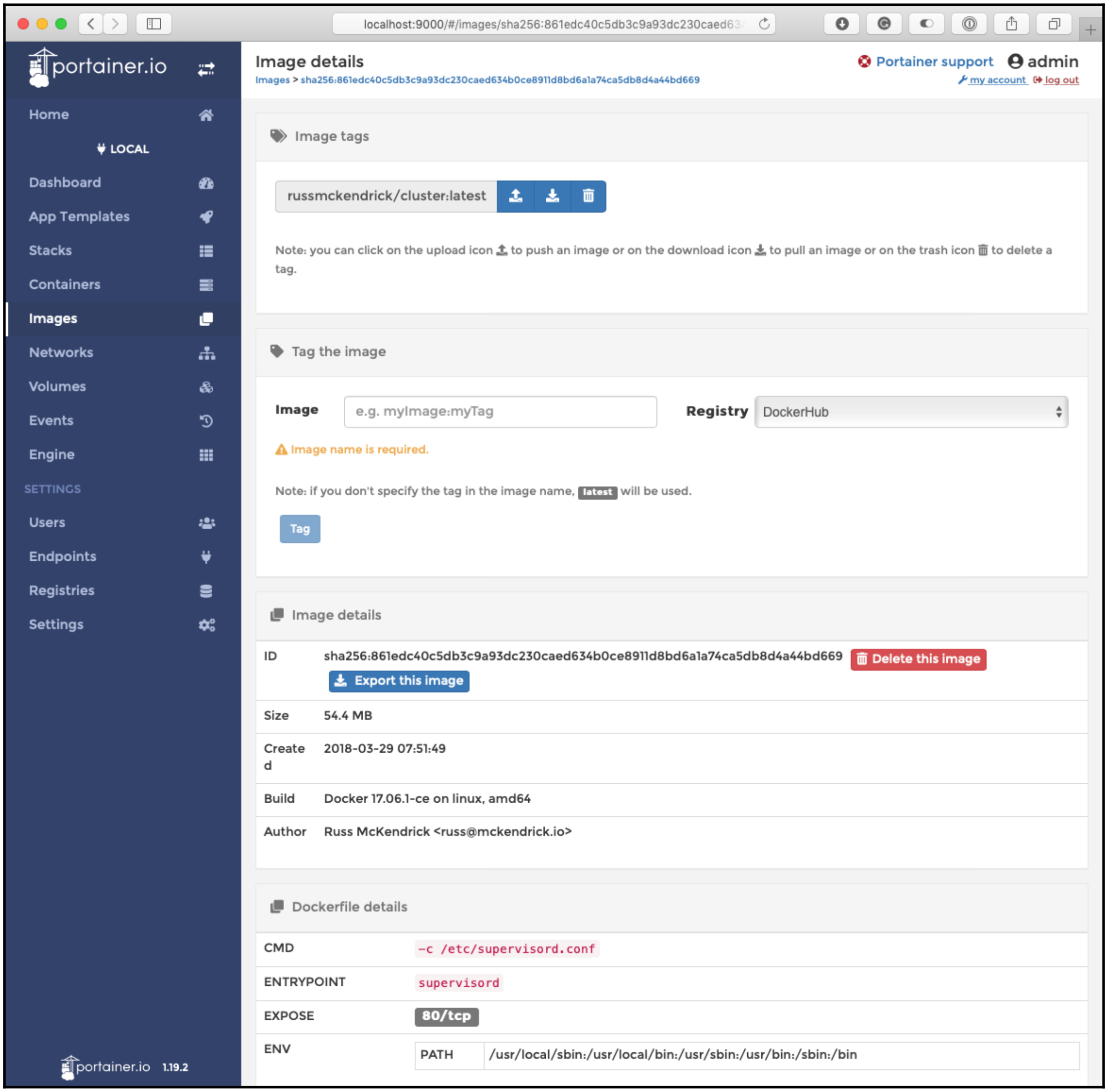


Image details

Images > sha256:861edc40c5db3c9a93dc230caed634b0ce8911d8bd6a1a74ca5db8d4a44bd669

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

Image: e.g. myImage:myTag Registry: DockerHub

⚠ Image name is required.

Note: if you don't specify the tag in the image name, latest will be used.

Tag

Image details

ID	sha256:861edc40c5db3c9a93dc230caed634b0ce8911d8bd6a1a74ca5db8d4a44bd669 Delete this image Export this image
Size	54.4 MB
Created	2018-03-29 07:51:49
Build	Docker 17.06.1-ce 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

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, the Docker Hub.

You also get a complete break down of each of the layers contained within the image, showing the command which was executed during the build and size of each layer.

Networks and volumes

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

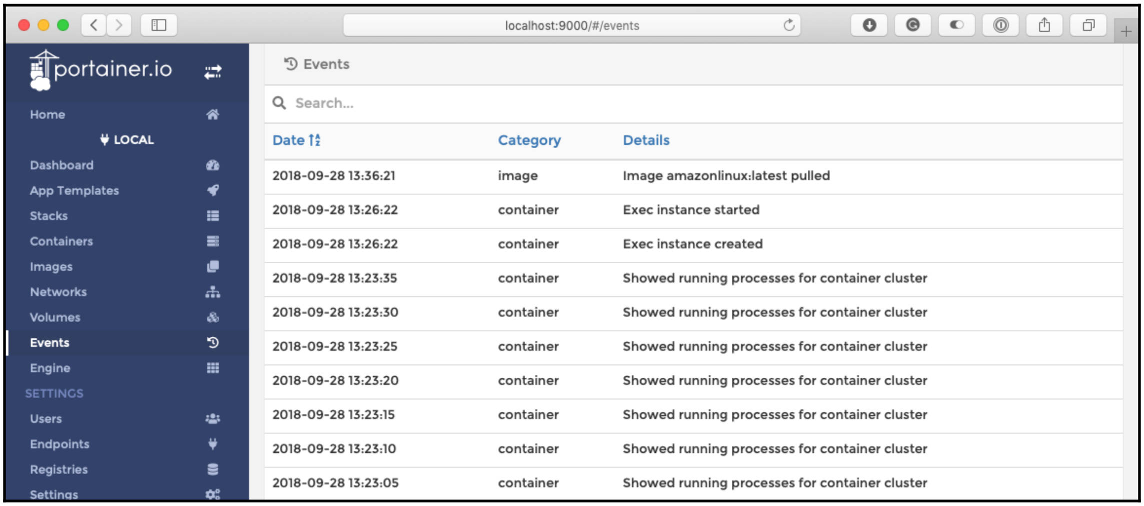
Here, you can quickly add a network using the default bridge driver. Clicking on **Advanced settings** will take you to a page with more options. These include using other drivers, defining the subnets, adding labels, and restricting external access to the network. As with other sections, you can also 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
2018-09-28 13:36:21	image	Image amazonlinux:latest pulled
2018-09-28 13:26:22	container	Exec instance started
2018-09-28 13:26:22	container	Exec instance created
2018-09-28 13:23:35	container	Showed running processes for container cluster
2018-09-28 13:23:30	container	Showed running processes for container cluster
2018-09-28 13:23:25	container	Showed running processes for container cluster
2018-09-28 13:23:20	container	Showed running processes for container cluster
2018-09-28 13:23:15	container	Showed running processes for container cluster
2018-09-28 13:23:10	container	Showed running processes for container cluster
2018-09-28 13:23:05	container	Showed running processes for container cluster

This is the equivalent of running the following command:

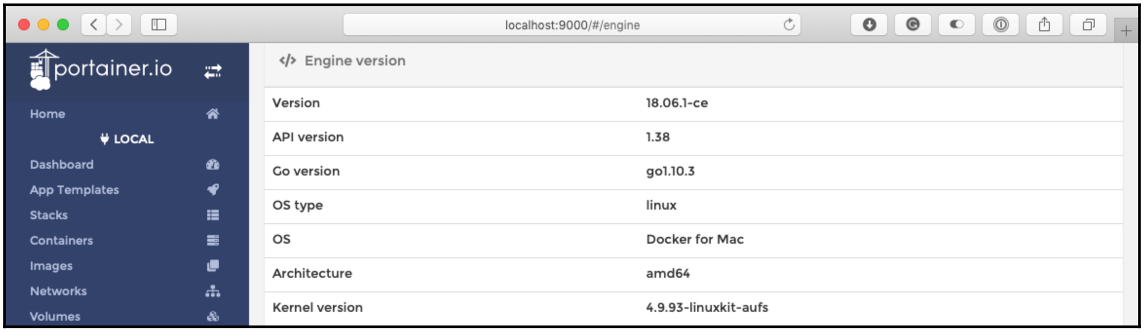
```
$ docker events --since '2018-09-27T16:30:00' --until '2018-09-28T16:30:00'
```

Engine

The final entry simply shows you the output of the following:

```
$ docker info
```

The following shows the output of the command:



Engine version

Version	18.06.1-ce
API version	1.38
Go version	go1.10.3
OS type	linux
OS	Docker for Mac
Architecture	amd64
Kernel version	4.9.93-linuxkit-aufs

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 move onto looking at Portainer running on Docker Swarm so now would be a good time to remove the running containers and also the volume which was created when we first launched Portainer, you can remove the volume using:

```
$ docker volume prune
```

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.

Creating the Swarm

As in the Docker Swarm chapter, we are going to be creating the Swarm locally using Docker Machine; to do this, run the following commands:

```
$ docker-machine create -d virtualbox swarm-manager
$ docker-machine create -d virtualbox swarm-worker01
$ docker-machine create -d virtualbox swarm-worker02
```

Once the three instances have launched, run the following command to initialize the Swarm:

```
$ docker $(docker-machine config swarm-manager) swarm init \
   --advertise-addr $(docker-machine ip swarm-manager):2377 \
   --listen-addr $(docker-machine ip swarm-manager):2377
```

Then run the following commands, inserting your own token, to add the worker nodes:

```
$
SWARM_TOKEN=SWMTKN-1-45acey6bqteiro42ipt3gy6san3kec0f8dh6fb35pnv1xz291v-418
9ei7v6az2b85kb5jnf7nku
$ docker $(docker-machine config swarm-worker01) swarm join \
    --token $SWARM_TOKEN \
    $(docker-machine ip swarm-manager):2377
$ docker $(docker-machine config swarm-worker02) swarm join \
    --token $SWARM_TOKEN \
    $(docker-machine ip swarm-manager):2377
```

Now that we have our cluster formed, run the following to point your local Docker client to the manager node:

```
$ eval $(docker-machine env swarm-manager)
```

Finally, check the status of the Swarm using the following command:

```
$ docker node ls
```

The Portainer service

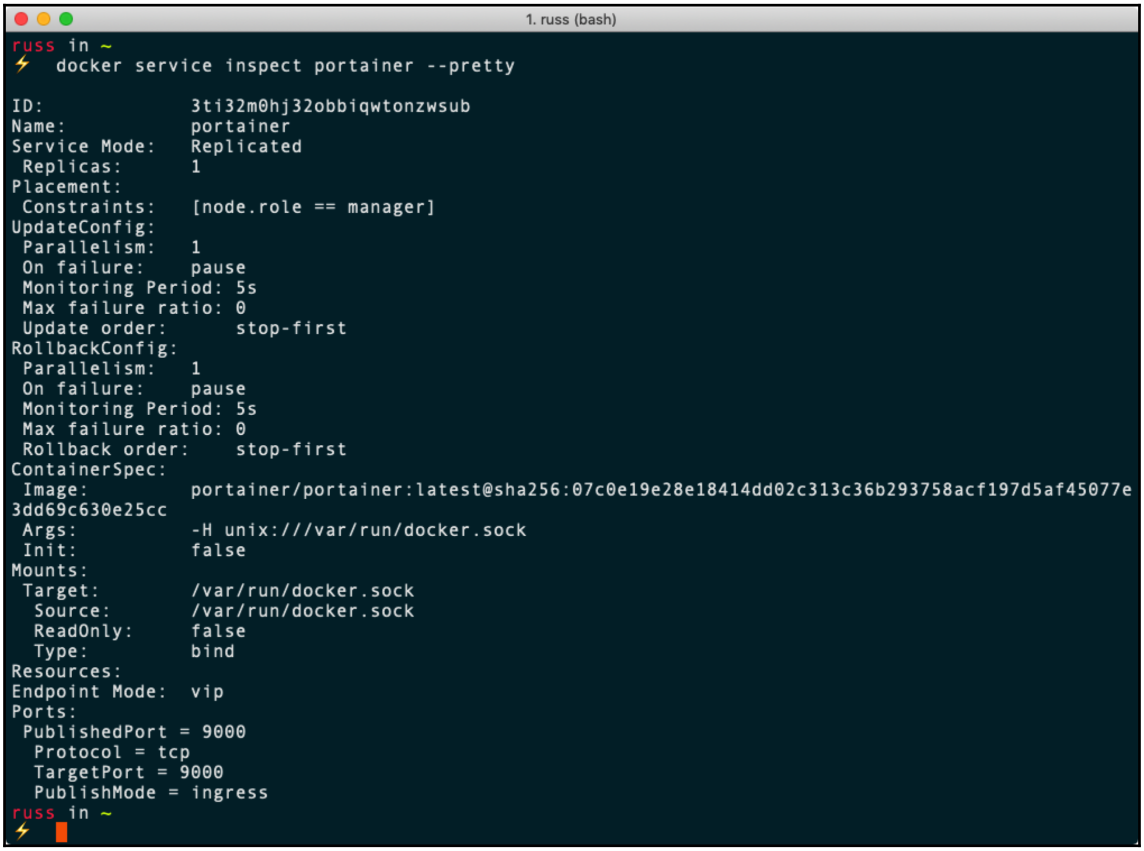
Now what we have a Docker Swarm cluster and our local client is configured to communicate with the manager node, we can launch the Portainer service by simply running:

```
$ docker service create \
  --name portainer \
  --publish 9000:9000 \
  --constraint 'node.role == manager' \
  --mount type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock \
  portainer/portainer \
  -H unix:///var/run/docker.sock
```

As you can see, this will launch Portainer as a service on the manager node and make the service mount the manager nodes socket file so that it has visibility of the rest of the Swarm. You can check that the service has launched without any errors using the following commands:

```
$ docker service ls
$ docker service inspect portainer --pretty
```

The following shows the output:



```
russ in ~
⚡  docker service inspect portainer --pretty

ID:             3ti32m0hj32obbiqwtonzwsub
Name:           portainer
Service Mode:   Replicated
 Replicas:      1
Placement:
 Constraints:   [node.role == manager]
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:         portainer/portainer:latest@sha256:07c0e19e28e18414dd02c313c36b293758acf197d5af45077e
3dd69c630e25cc
 Args:          -H unix:///var/run/docker.sock
 Init:          false
Mounts:
 Target:        /var/run/docker.sock
  Source:       /var/run/docker.sock
  ReadOnly:     false
  Type:         bind
Resources:
Endpoint Mode:  vip
Ports:
 PublishedPort = 9000
  Protocol = tcp
  TargetPort = 9000
  PublishMode = ingress
russ in ~
⚡
```

Now that the service has launched, you can access Portainer on port `9000` on any of the IP addresses of the nodes in your cluster, or run the following command:

```
$ open http://$(docker-machine ip swarm-manager):9000
```

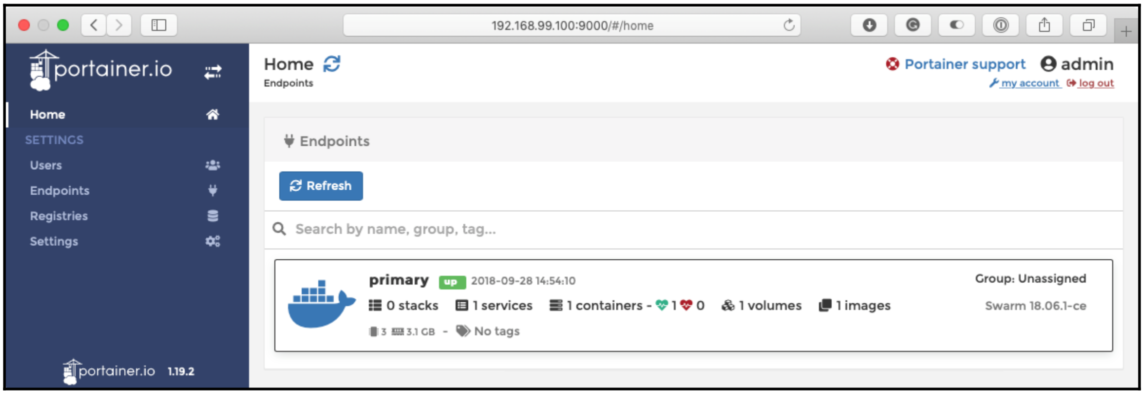
When the page opens, you will be once again be asked to set a password for the admin user; once set, you will be greeted with a login prompt. Once you have been logged in, you will be taken straight to the Dashboard. The reason for this is that when we launched Portainer this time, we passed it the argument `-H unix:///var/run/docker.sock`, which told Portainer to select the option we manually chose when we launched Portainer on our single host.

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.

Endpoints

The first thing you will have to do when you log in is select an endpoints, as you can see from the following screen, there is a single one called **primary**:

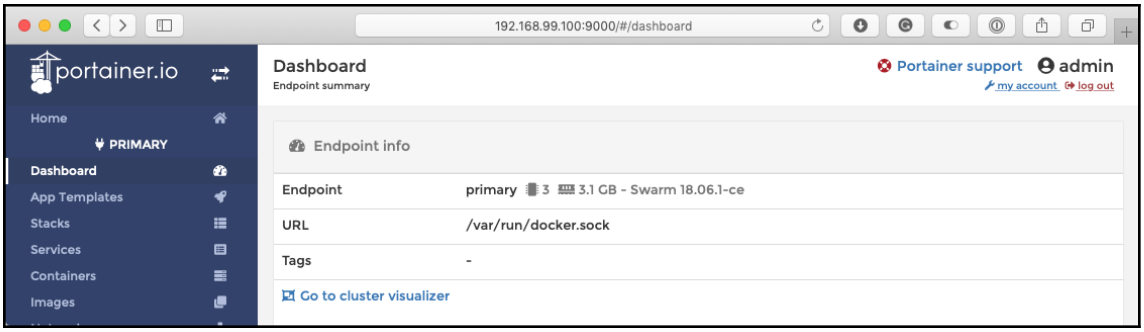


primary up 2018-09-28 14:54:10
0 stacks 1 services 1 containers - 1 0 1 volumes 1 images
3 3.1 GB - No tags
Group: Unassigned
Swarm 18.06.1-ce

Clicking on the endpoint will take you the **Dashboard**, we will look at **Endpoints** again at the end of the section.

Dashboard and Swarm

One of the first changes you will notice is that the Dashboard now displays information on the Swarm cluster, for example:



Dashboard
Endpoint summary

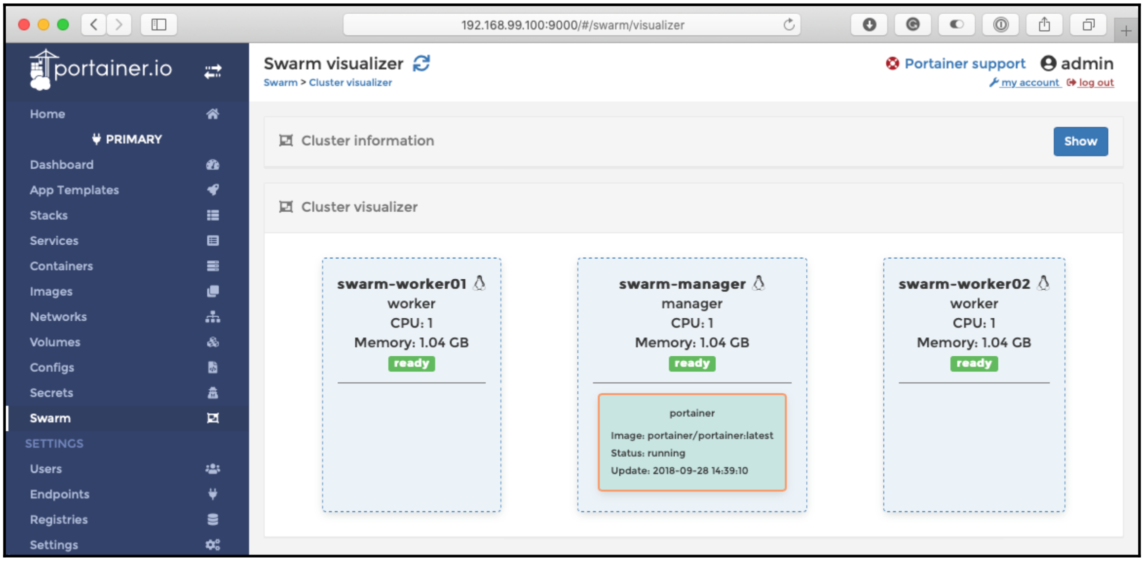
Endpoint info

Endpoint	primary 3 3.1 GB - Swarm 18.06.1-ce
URL	/var/run/docker.sock
Tags	-

Go to cluster visualizer

Notice how the CPU says 3 and the total RAM is 3.1 GB, each node within the cluster has 1 GB of RAM and 1 CPU, so these values are the cluster totals.

Clicking on **Go to cluster vizualizer** will take you to the Swam page, this gives you a visual overview of the cluster, where the only running service is currently Portainer:



Swarm visualizer
Swarm > Cluster visualizer

Cluster information

Cluster visualizer

swarm-worker01
worker
CPU: 1
Memory: 1.04 GB
ready

swarm-manager
manager
CPU: 1
Memory: 1.04 GB
ready

portainer
Image: portainer/portainer:latest
Status: running
Update: 2018-09-28 14:39:10

swarm-worker02
worker
CPU: 1
Memory: 1.04 GB
ready

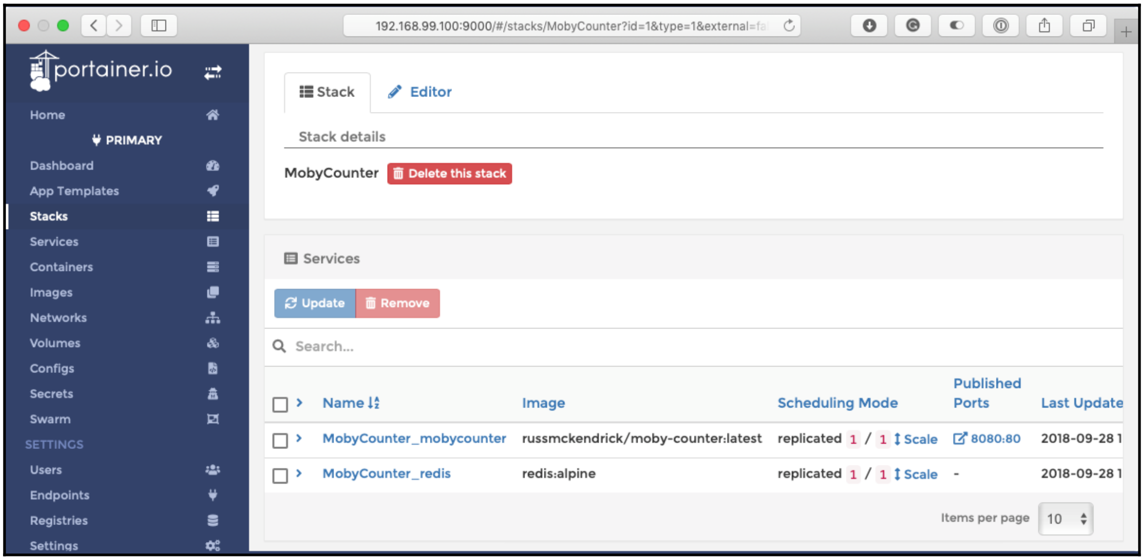
Stacks

The one item we didn't cover 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 contents above into the web-editor, enter a name of `MobyCounter`, do not add any spaces or special characters to the name as this is used by Docker for and then click on **Deploy the stack**.

Once deployed you will be able to click on **MobyCounter** and manage the stack:



Stack details

MobyCounter Delete this stack

Services

	Name	Image	Scheduling Mode	Published Ports	Last Update
□	MobyCounter_mobycounter	russmckendrick/moby-counter:latest	replicated 1 / 1 Scale	8080:80	2018-09-28 1
□	MobyCounter_redis	redis:alpine	replicated 1 / 1 Scale	-	2018-09-28 1

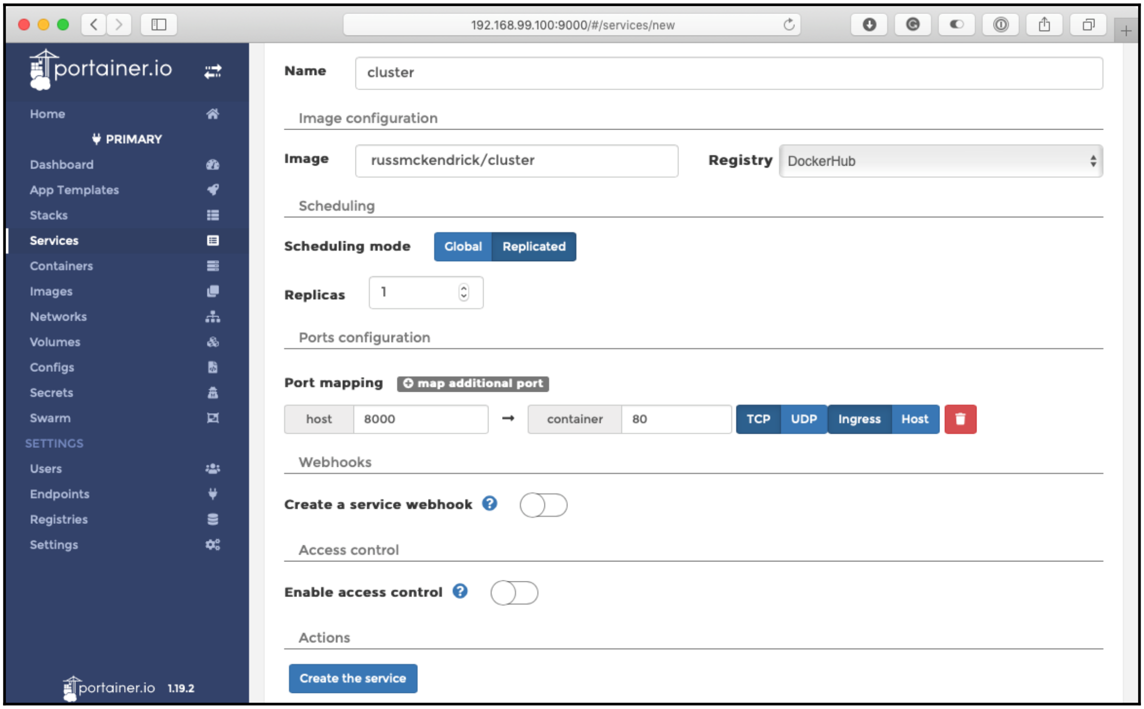
Stacks are a collection of services, so let's take a look at them next.

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 `8000` on the host to map to port `80` to the container, this is because the stack we launched in the last section is already using port `8080` on the host:



Name: cluster

Image configuration

Image: russmckendrick/cluster
Registry: DockerHub

Scheduling

Scheduling mode: Global [✓ Replicated]

Replicas: 1

Ports configuration

Port mapping ⊕ map additional port

host 8000 → container 80 [✓ TCP] UDP [✓ Ingress] Host

Webhooks

Create a service webhook ? ○

Access control

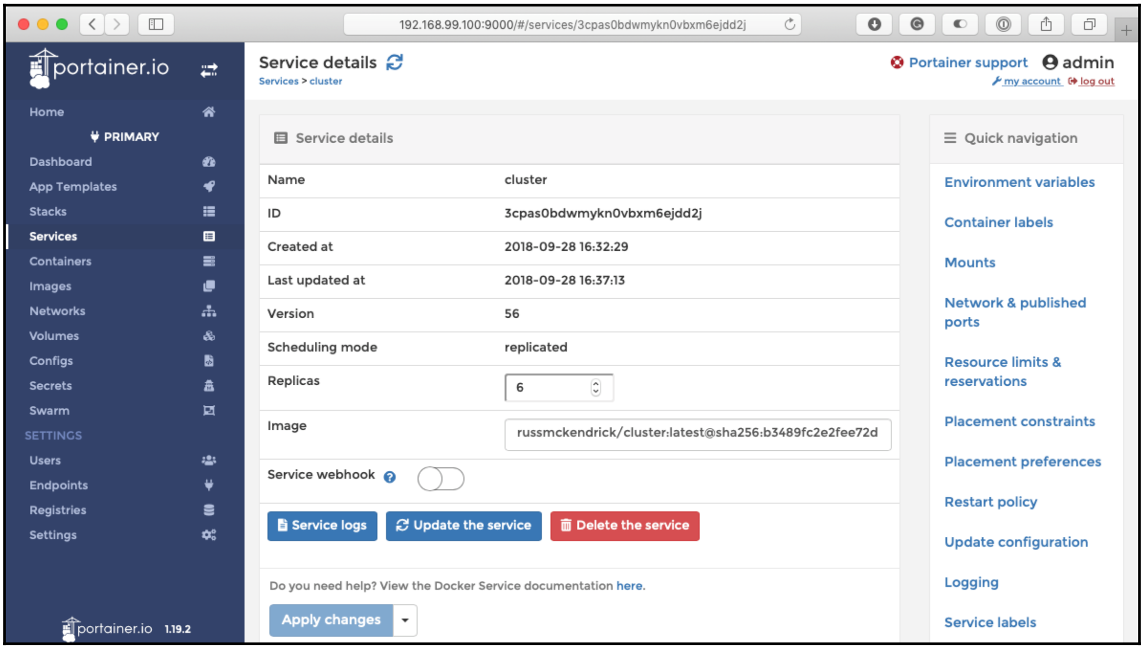
Enable access control ? ○

Actions

Create the 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 column, 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** column takes us to an overview of the service. As you can see, there is a lot of information on the service:



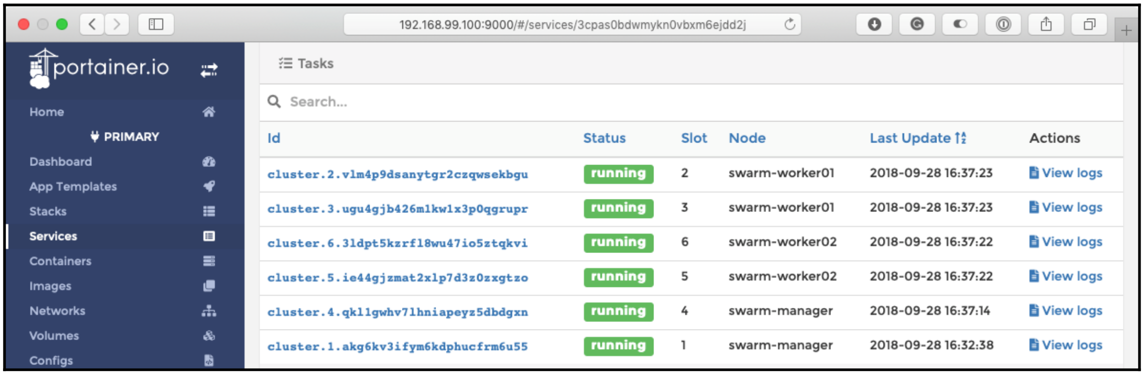
Service details

Service details

Name	cluster
ID	3cpas0bdwmykn0vbxm6ejdd2j
Created at	2018-09-28 16:32:29
Last updated at	2018-09-28 16:37:13
Version	56
Scheduling mode	replicated
Replicas	6
Image	russmckendrick/cluster:latest@sha256:b3489fc2e2fee72d
Service webhook	○

Do you need help? View the Docker Service documentation here.

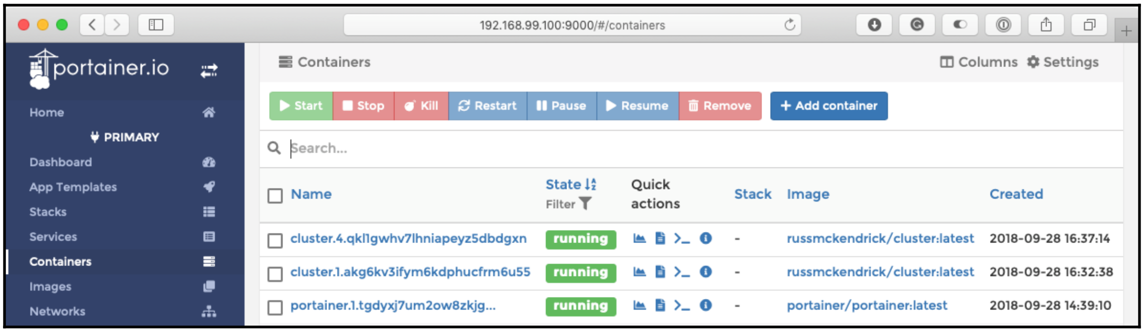
You can make a lot of changes to the **Service** on the fly, including placement constraints, the restart policy, adding service labels, and more. Right at the bottom of the page is a list of the tasks associated with the service:



Tasks

Id	Status	Slot	Node	Last Update	Actions
cluster.2.vlm4p9dsanytgr2czqwsekbgu	running	2	swarm-worker01	2018-09-28 16:37:23	View logs
cluster.3.ugu4gjb426mlkwlx3p0qgrupr	running	3	swarm-worker01	2018-09-28 16:37:23	View logs
cluster.6.3ldpt5kzrfl8wu47io5ztqkvi	running	6	swarm-worker02	2018-09-28 16:37:22	View logs
cluster.5.ie44gjzmat2xlp7d3z0zxgtzo	running	5	swarm-worker02	2018-09-28 16:37:22	View logs
cluster.4.qkl1gwhv7lhniapeyz5dbdgxn	running	4	swarm-manager	2018-09-28 16:37:14	View logs
cluster.1.akg6kv3ifym6kdphucfrm6u55	running	1	swarm-manager	2018-09-28 16:32:38	View logs

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
□	cluster.4.qkl1gwhv7lhniapeyz5dbdgxn	running		-	russmckendrick/cluster:latest	2018-09-28 16:37:14
□	cluster.1.akg6kv3ifym6kdphucfrm6u55	running		-	russmckendrick/cluster:latest	2018-09-28 16:32:38
□	portainer.1.tgdyxj7um2ow8zkjg...	running		-	portainer/portainer:latest	2018-09-28 14:39:10

There are only three containers listed, and one of them is for the Portainer service. Why is that?

Well, if you remember in the Docker Swarm chapter, we learned that `docker container` commands only really apply to the node you are running them against, and as Portainer is only talking to our manager node, that is the only node which the Docker container commands are executed against. Remember that Portainer is only a web interface for the Docker API, so it mirrors the same results as you get running `docker container ls` on the command line.

Adding endpoints

However, we can add our two remaining cluster nodes to Portainer. To do this, click on the **Endpoint** entry in the left-hand menu.

To add the endpoint, we will need to know the endpoint URL and have access to the certificates so that Portainer can authenticate itself against the Docker daemon running on the node. Luckily, as we launched the hosts using Docker Machine, this is a simple task. To get the endpoint URLs, run the following command:

```
$ docker-machine ls
```

For me, the two endpoint URLs were `192.168.99.101:2376` and `192.168.99.102:2376`; yours may be different. The certificates we need to upload can be found in the `~/.docker/machine/certs/` folder on your machine. I recommend running the following commands to open the folder in your finder:

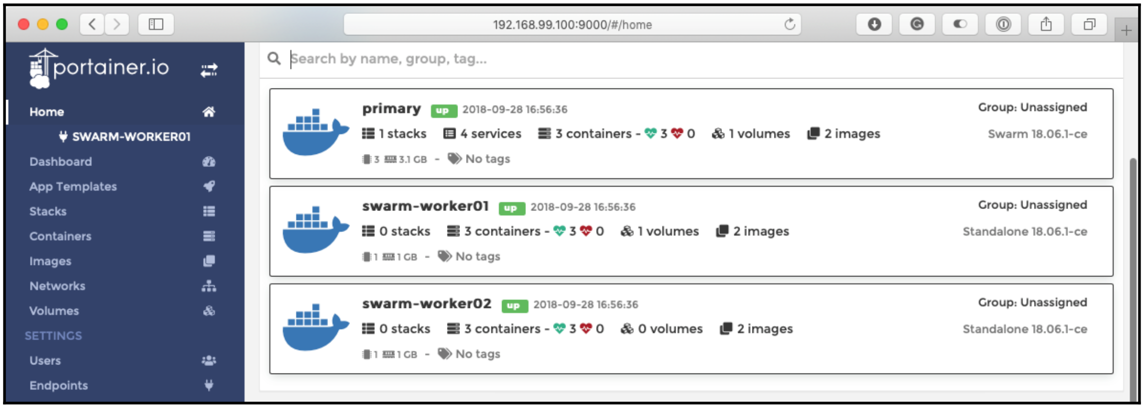
```
$ cd ~/.docker/machine/certs/
$ open .
```

Once you have added the node, you will be able to change to it using the **+ Add Endpoint** button in the **Settings / Endpoints** page.

From here enter the following information:

- **Name**: `swarm-worker01`
- **Endpoint URL**: `192.168.99.101:2376`
- **Public IP:** `192.168.99.101`
- **TLS**: On
- **TLS with server and client verification**: Ticked
- Upload the certs from `~/.docker/machine/certs/`

Then click on the **+ Add endpoint** button, clicking on **Home** will take you to the Endpoint overview screen we first saw at the start of this section of the chapter. As you can see from the following screenshot, we can see that the workers are running three containers each and that they are marked as standalone rather than Swarm:



primary up 2018-09-28 16:56:36 Group: Unassigned
1 stacks 4 services 3 containers - 3 0 1 volumes 2 images Swarm 18.06.1-ce
3 3.1 GB - No tags

swarm-worker01 up 2018-09-28 16:56:36 Group: Unassigned
0 stacks 3 containers - 3 0 1 volumes 2 images Standalone 18.06.1-ce
1 1 GB - No tags

swarm-worker02 up 2018-09-28 16:56:36 Group: Unassigned
0 stacks 3 containers - 3 0 0 volumes 2 images Standalone 18.06.1-ce
1 1 GB - No tags

You will also notice that other than the Swarm being mention in the Endpoint, there's no mention of Swarm services. Again, this is because Portainer only knows as much as your Docker nodes, and Swarm mode only allows nodes with the role of manager to launch services and tasks and interact with the other nodes in your cluster.

Don't forget to remove your local Docker Swarm cluster by running:

```
$ docker-machine rm swarm-manager swarm-worker01 swarm-worker02
```

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 take a look at how to secure your Docker host as well as how to run scans against your container images.

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 at here:

- Main website: `https://portainer.io/`
- Portainter on GitHub: `https://github.com/portainer/`
- Latest documentation: `https://portainer.readthedocs.io/en/latest/index.html`
- Template documentation: `http://portainer.readthedocs.io/en/latest/templates.html`

12
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 application
- Third-party security services

Technical requirements

In this chapter, we will be using Docker on the desktop, and we will be using Docker Machine to launch a Docker host in the cloud. Like 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:
`http://bit.ly/2AnEv5G`

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. This is something that we discussed in `Chapter 2`, *Building Container Images*; you should never do it, as it is considered a bad practice.

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, the Docker Engine does a lot of work behind the scenes. One of the tasks that the 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 you saw in previous chapters, being able to launch your containers into 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 biggest 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 host

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 he needs access to all of the VMs. Most virtual machine management tools allow you to grant role-based access to your VMs.

With Docker, you have a little disadvantage, because whoever has access to the Docker Engine on your Docker host, either through being granted sudo access or by having their user added to the Docker Linux group, has access to every Docker container that you are running. They can run new containers, they can stop existing containers, and they can delete images, as well. Be careful with who you grant permission to access the Docker Engine on your hosts. They essentially hold the keys to the kingdom, with respect to 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 of 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 really easy to ignore a quite crucial security consideration: Do you know what it is running inside of 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 the Docker Hub.

While containers have the protection of namespaces, control groups, and network isolation, we discussed how a poorly judged image download can introduce security concerns and risk into your environment. For example, a perfectly legitimate container running an unpatched piece of software can introduce risk to the availability of your application and data.

Docker commands

Let's take a look at the Docker commands that can be used to help tighten up security, as well as view information about the images that you might be using.

There are two commands that we will focus on. 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.

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 of 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 the applications can't write to certain locations.

Similar to the previous command, `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 into a container, it will mount as an empty volume over the top of the directory inside of the container, unless you use the `--volumes-from` switch or add data to the container in some other way after it has been launched:

```
$ 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.

diff command

Let's take another look at the `docker diff` command; since it relates to the security aspects of the 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 `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:

```
A /trying_to_write_a_file
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
```

This is a great way to spot anything untoward or unexpected that may be going on within your container.

Best practices

In this section, we will look at the best practices when it comes to Docker, as well as the *Center for Internet Security* guide, 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:

- **One application per container**: Spread out your applications to one 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, only install what you need in your container images. If you have to install more 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 that whoever has root or sudo access to your Docker hosts has access to manipulate all of the images and containers on the host.
- **Use the latest version**: 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, and also Red Hat.
- **Use the resources**: Use the resources available if you need help. The community within Docker 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 14`, *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, which 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 230-page PDF, released under the Creative Commons license, and it covers Docker CE 17.06 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:

- The host configuration
- The Docker daemon configuration
- The Docker daemon configuration files
- Container images/runtime
- Docker security operations

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, cover 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 Benchmark Security application that you can install and run. The tool will inspect the following:

- The host configuration
- The Docker daemon configuration
- The Docker daemon configuration files
- Container images and build files
- Container runtime
- The Docker security operations
- Docker Swarm configuration

Look 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.

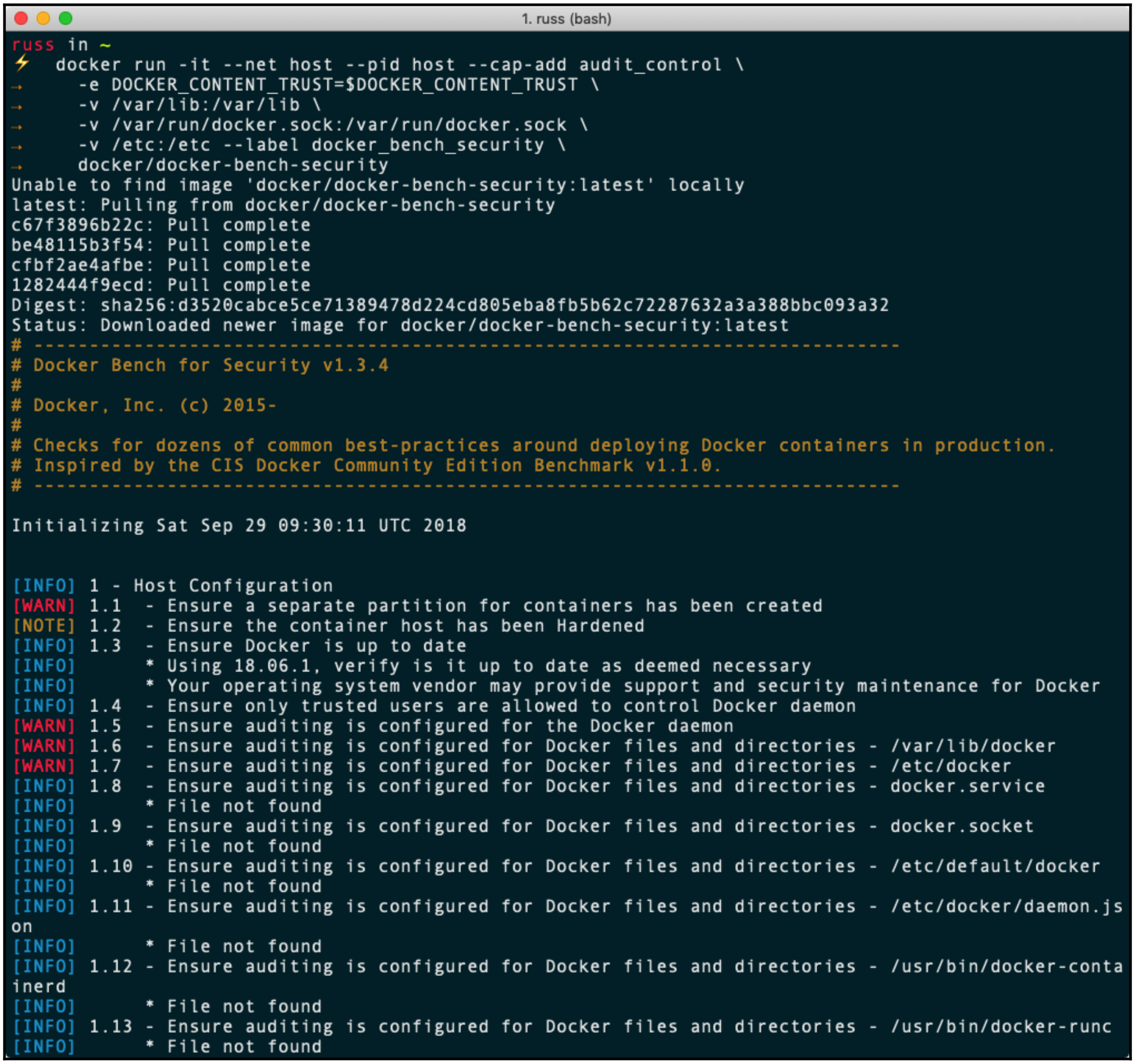
Running the tool on Docker for macOS and Docker for Windows

Running the tool is simple. It's already been packaged for us, inside of 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 the `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` based system shortly:

```
$ docker 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:



```
russ in ~
⚡  docker 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
Unable to find image 'docker/docker-bench-security:latest' locally
latest: Pulling from docker/docker-bench-security
c67f3896b22c: Pull complete
be48115b3f54: Pull complete
cfbf2ae4afbe: Pull complete
1282444f9ecd: Pull complete
Digest: sha256:d3520cabce5ce71389478d224cd805eba8fb5b62c72287632a3a388bbc093a32
Status: Downloaded newer image for docker/docker-bench-security:latest
# ------------------------------------------------------------------------------
# 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 Sat Sep 29 09:30:11 UTC 2018


[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 18.06.1, 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.js
on
[INFO]      * File not found
[INFO] 1.12 - Ensure auditing is configured for Docker files and directories - /usr/bin/docker-conta
inerd
[INFO]      * File not found
[INFO] 1.13 - Ensure auditing is configured for Docker files and directories - /usr/bin/docker-runc
[INFO]      * File not found
```

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 16.04.5 LTS server in DigitalOcean, and perform a clean installation of Docker using Docker Machine, as follows:

```
$ DOTOKEN=0cb54091fecfe743920d0e6d28a29fe325b9fc3f2f6fccba80ef4b26d41c7224
$ docker-machine create \
    --driver digitalocean \
    --digitalocean-access-token $DOTOKEN \
    docker-digitalocean
```

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 the 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 this using the following code:

```
$ 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`.

To start the audit on our Ubuntu Docker host, I ran the following:

```
$ docker 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 three 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 that 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
- Container runtime
- Docker security operations
- Docker Swarm configuration

Let's take a look at what we are seeing 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/learn/customized-file-monitoring-auditd/`.

Docker daemon configuration

My Docker daemon configuration flagged up eight `[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/engine/userguide/networking/`.

```
[WARN] 2.5 - Ensure aufs storage driver is not used
```

AUFS was used quite a lot in Docker's early days; however, it is no longer considered a best practice, as it could be responsible for issues in the host machine's Kernel:

```
[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/engine/admin/logging/overview/` 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/engine/admin/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/engine/userguide/networking/default_network/binding/`:

```
[WARN] 2.18 - Ensure containers are restricted from acquiring new
privileges
```

This stops the processes within the containers potentially can't gain 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 Docker Machine.

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.

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 two `[WARN]` statuses are 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 the Docker Engine running on your Docker host; for example, when you are running a GUI (such as Portainer), which we covered in `Chapter 11`, *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 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/engine/userguide/networking/ 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 the 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 http://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, browse to 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 of your container. This has the potential to crash you 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 - Perform regular security audits of your host system and
containers
[INFO] 6.2 - Monitor Docker containers usage, performance and metering
[INFO] 6.3 - Backup container data
[INFO] 6.4 - Avoid image sprawl
[INFO]     * There are currently: 4 images
[INFO] 6.5 - Avoid container sprawl
[INFO]     * There are currently a total of 8 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:

```
[PASS] 7.1 - Ensure swarm mode is not Enabled, if not needed
[PASS] 7.2 - Ensure the minimum number of manager nodes have been created
in a swarm (Swarm mode not enabled)
[PASS] 7.3 - Ensure swarm services are binded to a specific host interface
(Swarm mode not enabled)
[PASS] 7.5 - Ensure Docker's secret management commands are used for
managing secrets in a Swarm cluster (Swarm mode not enabled)
[PASS] 7.6 - Ensure swarm manager is run in auto-lock mode (Swarm mode not
enabled)
[PASS] 7.7 - Ensure swarm manager auto-lock key is rotated periodically
(Swarm mode not enabled)
[PASS] 7.8 - Ensure node certificates are rotated as appropriate (Swarm
mode not enabled)
[PASS] 7.9 - Ensure CA certificates are rotated as appropriate (Swarm mode
not enabled)
[PASS] 7.10 - Ensure management plane traffic has been separated from data
plane traffic (Swarm mode not enabled)
```

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 230-page document.

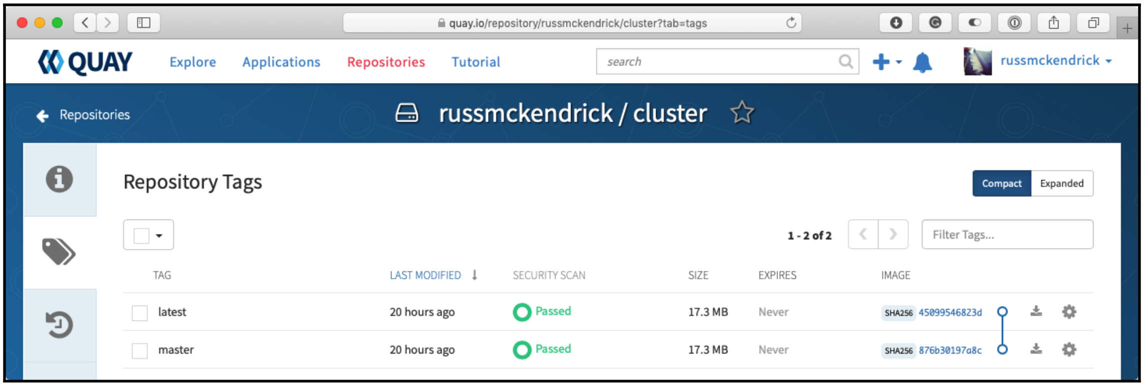
Third-party security services

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 service by CoreOS, which was purchased by Red Hat, is similar to the Docker Hub/Registry; one difference is that Quay actually performs a security scan of each image after it is pushed/built.

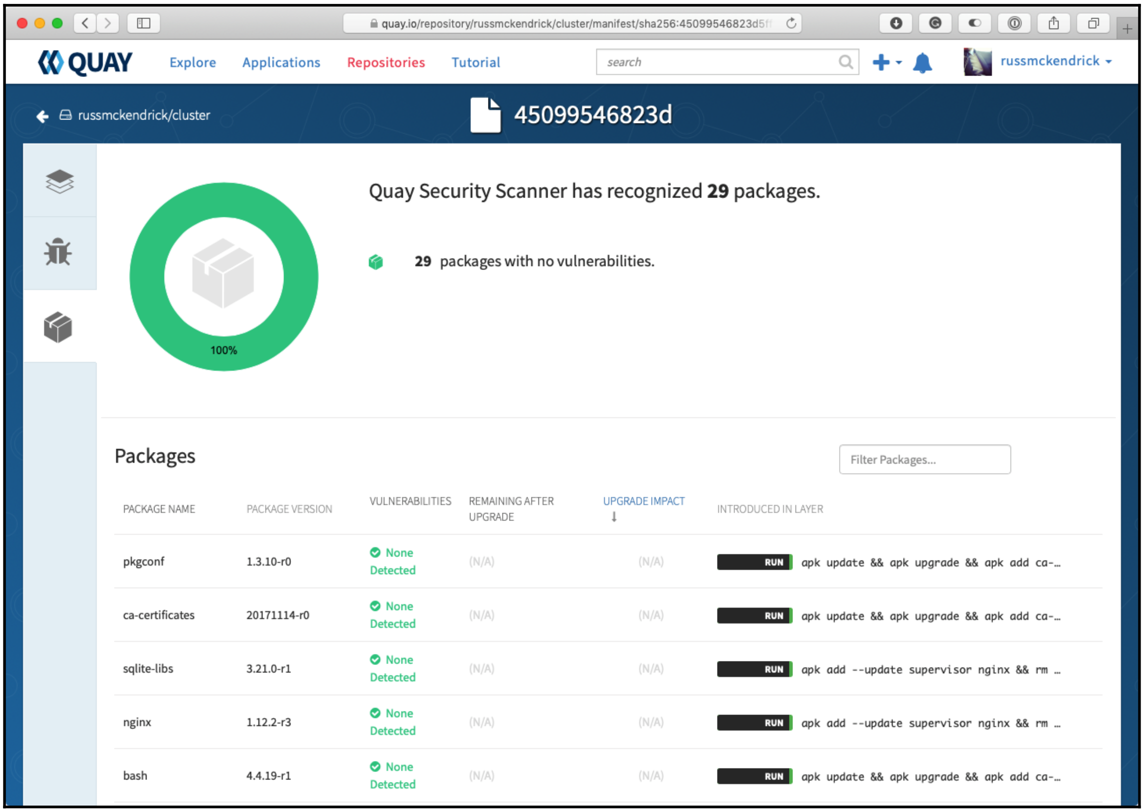
You can see the results of the scan by viewing the **Repository Tags** for your chosen image; here, you will see a column for **Security Scan**. As you can see in the following screenshot, in the example image that we created, there are no problems:



Repository Tags

TAG	LAST MODIFIED	SECURITY SCAN	SIZE	EXPIRES	IMAGE
latest	20 hours ago	Passed	17.3 MB	Never	SHA256 45099546823d
master	20 hours ago	Passed	17.3 MB	Never	SHA256 876b30197a8c

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 29 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 **29** packages.

29 packages with no vulnerabilities.

Packages

Filter Packages...

PACKAGE NAME	PACKAGE VERSION	VULNERABILITIES	REMAINING AFTER UPGRADE	UPGRADE IMPACT ↓	INTRODUCED IN LAYER
pkgconf	1.3.10-r0	✓ None Detected	(N/A)	(N/A)	RUN `apk update && apk upgrade && apk add ca-…`
ca-certificates	20171114-r0	✓ None Detected	(N/A)	(N/A)	RUN `apk update && apk upgrade && apk add ca-…`
sqlite-libs	3.21.0-r1	✓ None Detected	(N/A)	(N/A)	RUN `apk add --update supervisor nginx && rm …`
nginx	1.12.2-r3	✓ None Detected	(N/A)	(N/A)	RUN `apk add --update supervisor nginx && rm …`
bash	4.4.19-r1	✓ None Detected	(N/A)	(N/A)	RUN `apk update && apk upgrade && apk add ca-…`

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 CoreOS. 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://coreos.com/clair/docs/latest/api_v1.html`.

The installation instructions can be found at the project's GitHub page, at `https://github.com/coreos/clair/`.

Also, you can find a list of tools that support Clair on its integration page, at `https://coreos.com/clair/docs/latest/integrations.html`.

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-premise 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 shall start by creating the folders that we need, and we will also download the Docker Compose and basic configuration file from the project GitHub repository:

```
$ mkdir anchore anchore/config
$ cd anchore
$ curl
https://raw.githubusercontent.com/anchore/anchore-engine/master/scripts/doc
ker-compose/docker-compose.yaml -o docker-compose.yaml
$ curl
https://raw.githubusercontent.com/anchore/anchore-engine/master/scripts/doc
ker-compose/config.yaml -o config/config.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 to install the command-line client. If you are running macOS, then you have to run the following commands, ignoring the first if you already have `pip` installed:

```
$ sudo easy_install pip
$ pip install --user anchorecli
$ export PATH=${PATH}:${HOME}/Library/Python/2.7/bin
```

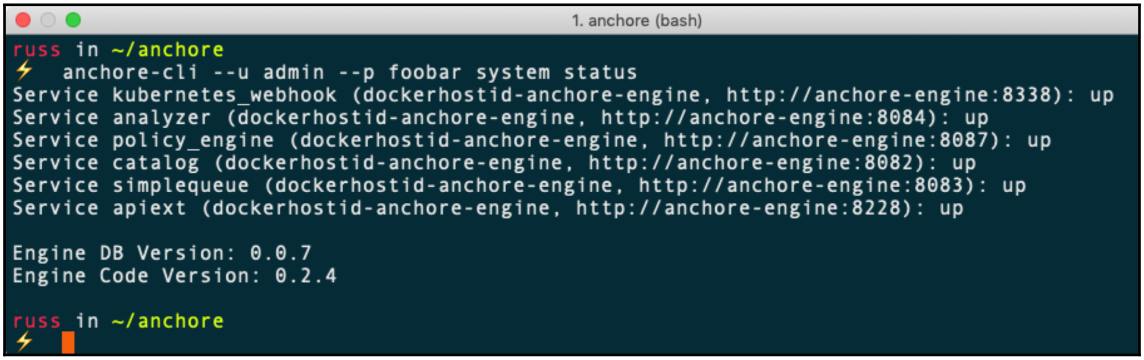
For Ubuntu users, you should run the following commands, this time ignoring the first two commands if you already have `pip`:

```
$ sudo apt-get update
$ sudo apt-get install python-pip
$ sudo pip install anchorecli
```

Once it has installed, you can run the following commands to check the status of your installation:

```
$ anchore-cli --u admin --p foobar 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`:

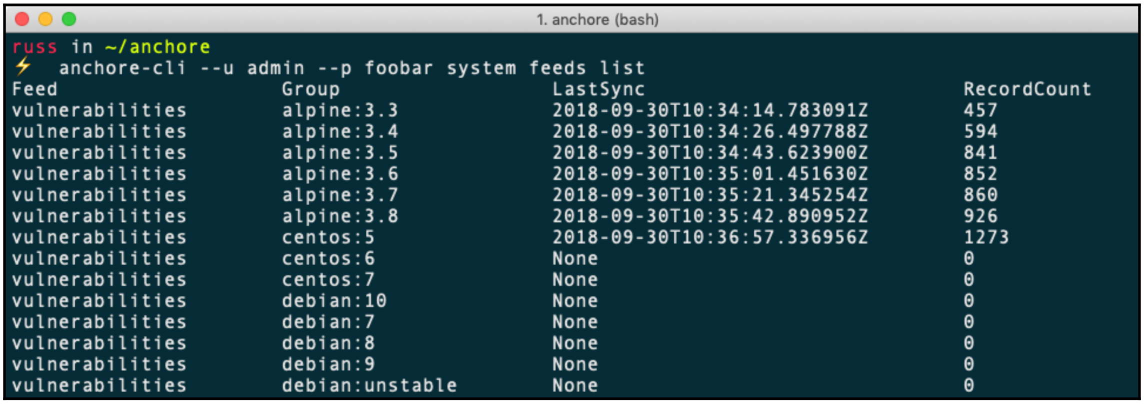


```
russ in ~/anchore
⚡  anchore-cli --u admin --p foobar system status
Service kubernetes_webhook (dockerhostid-anchore-engine, http://anchore-engine:8338): up
Service analyzer (dockerhostid-anchore-engine, http://anchore-engine:8084): up
Service policy_engine (dockerhostid-anchore-engine, http://anchore-engine:8087): up
Service catalog (dockerhostid-anchore-engine, http://anchore-engine:8082): up
Service simplequeue (dockerhostid-anchore-engine, http://anchore-engine:8083): up
Service apiext (dockerhostid-anchore-engine, http://anchore-engine:8228): up

Engine DB Version: 0.0.7
Engine Code Version: 0.2.4

russ in ~/anchore
⚡
```

The next command shows you where Anchore is in the database sync:

```
$ anchore-cli --u admin --p foobar system feeds list
```

As you can see in the following screenshot, my installation is currently syncing the CentOS 6 database. This process can take up to a few hours; however, for our example, we are going to be scanning an Alpine Linux based image as shown:



```
russ in ~/anchore
⚡  anchore-cli --u admin --p foobar system feeds list
Feed                   Group                  LastSync                           RecordCount
vulnerabilities        alpine:3.3             2018-09-30T10:34:14.783091Z        457
vulnerabilities        alpine:3.4             2018-09-30T10:34:26.497788Z        594
vulnerabilities        alpine:3.5             2018-09-30T10:34:43.623900Z        841
vulnerabilities        alpine:3.6             2018-09-30T10:35:01.451630Z        852
vulnerabilities        alpine:3.7             2018-09-30T10:35:21.345254Z        860
vulnerabilities        alpine:3.8             2018-09-30T10:35:42.890952Z        926
vulnerabilities        centos:5               2018-09-30T10:36:57.336956Z        1273
vulnerabilities        centos:6               None                               0
vulnerabilities        centos:7               None                               0
vulnerabilities        debian:10              None                               0
vulnerabilities        debian:7               None                               0
vulnerabilities        debian:8               None                               0
vulnerabilities        debian:9               None                               0
vulnerabilities        debian:unstable        None                               0
```

Next up, we have to grab an image to scan; let's grab an older image, as follows:

```
$ anchore-cli --u admin --p foobar image add docker.io/russmckendrick/moby-
counter:old
```

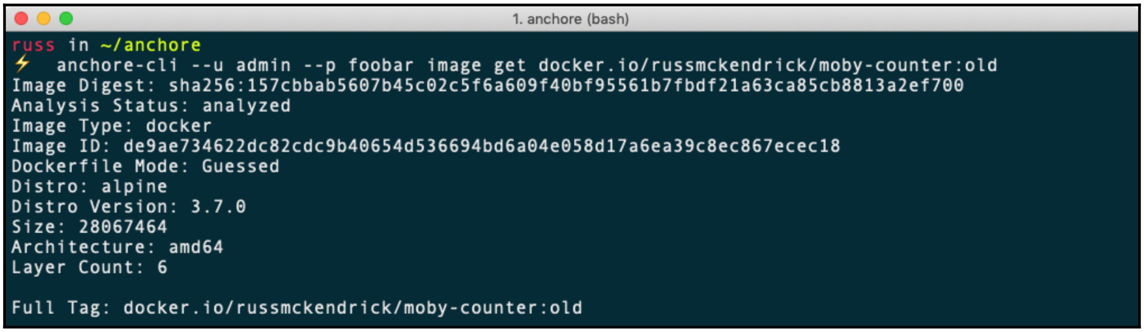
It will a minute or two to run its initial scan; you can check the status by running the following:

```
$ anchore-cli --u admin --p foobar image list
```

After a while, the status should change from `analyzing` to `analyzed`:

```
$ anchore-cli --u admin --p foobar image get docker.io/russmckendrick/moby-counter:old
```

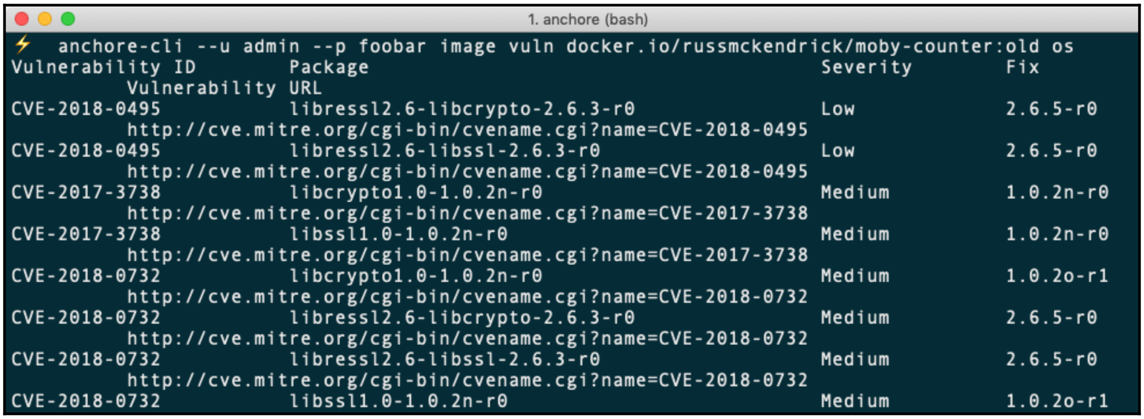
This will show you an overview of the image, as follows:



```
russ in ~/anchore
⚡  anchore-cli --u admin --p foobar image get docker.io/russmckendrick/moby-counter:old
Image Digest: sha256:157cbbab5607b45c02c5f6a609f40bf95561b7fbdf21a63ca85cb8813a2ef700
Analysis Status: analyzed
Image Type: docker
Image ID: de9ae734622dc82cdc9b40654d536694bd6a04e058d17a6ea39c8ec867ecec18
Dockerfile Mode: Guessed
Distro: alpine
Distro Version: 3.7.0
Size: 28067464
Architecture: amd64
Layer Count: 6

Full Tag: docker.io/russmckendrick/moby-counter:old
```

You can then view a list of problems (if there are any), by running the following:

```
$ anchore-cli --u admin --p foobar image vuln
docker.io/russmckendrick/moby-counter:old os
```



```
⚡  anchore-cli --u admin --p foobar image vuln docker.io/russmckendrick/moby-counter:old os
Vulnerability ID        Package                                         Severity        Fix
         Vulnerability URL
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
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
CVE-2017-3738           libcrypto1.0-1.0.2n-r0                          Medium          1.0.2n-r0
         http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-3738
CVE-2017-3738           libssl1.0-1.0.2n-r0                             Medium          1.0.2n-r0
         http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-3738
CVE-2018-0732           libcrypto1.0-1.0.2n-r0                          Medium          1.0.2o-r1
         http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-0732
CVE-2018-0732           libressl2.6-libcrypto-2.6.3-r0                  Medium          2.6.5-r0
         http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-0732
CVE-2018-0732           libressl2.6-libssl-2.6.3-r0                     Medium          2.6.5-r0
         http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-0732
CVE-2018-0732           libssl1.0-1.0.2n-r0                             Medium          1.0.2o-r1
```

As you can see, each package that is listed has the current version, a link to the CVE issue, and also, a confirmation of the version number that fixes the reported issue.

You can use the following commands to remove the Anchore containers:

```
$ docker-compose stop
$ docker-compose rm
```

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 regards 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 can minimize any potential damage an intruder can 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/`.

13
Docker Workflows

In this chapter, we will be looking at Docker and various workflows for Docker. We'll put all the pieces together so you can start using Docker in your production environments and feel comfortable doing so. Let's take a peek at what we will be covering in this chapter:

- Docker for development
- Monitoring Docker
- Extending to external platforms
- What does production look like?

Technical requirements

In this chapter, we will be using Docker on the desktop. Like 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 system.

A full copy of the code used in this chapter can be found in the GitHub repository at `https://github.com/PacktPublishing/Mastering-Docker-Third-Edition/tree/master/chapter14`.

Check out the following video to see the Code in Action:
`http://bit.ly/2SaG0uP`

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.

For 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 with 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 (`WP-CLI`). This allows you to stop, start, and even remove containers without losing your work.

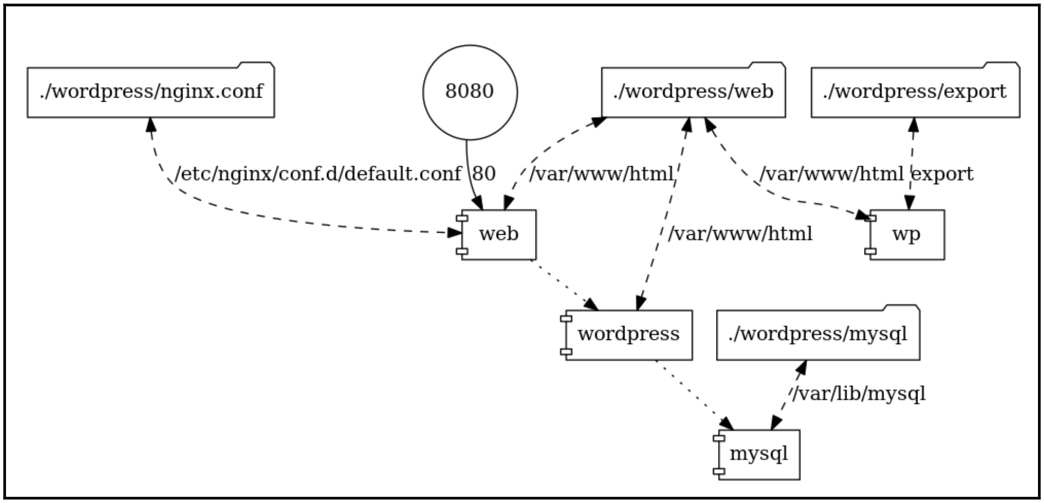
Before we launch our WordPress installation, let's take a look at the Docker Compose file and what services we have running:

```
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:
      - mysql
  mysql:
      image: mysql:5
      environment:
        MYSQL_ROOT_PASSWORD: "wordpress"
        MYSQL_USER: "wordpress"
        MYSQL_PASSWORD: "wordpress"
        MYSQL_DATABASE: "wordpress"
      volumes:
        - "./wordpress/mysql:/var/lib/mysql"
  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:



You can use `docker-compose-viz` to give yourself a visual representation of any Docker Compose file. As you can see from ours, we have four services defined.

The first is called `web`. This service is the only one of the four 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. Before we look at these, take a look at the following nginx configuration:

```
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 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`.

This means 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.

The second service is `wordpress`; this is the official WordPress image from `https://store.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.

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 approach of running a single service on each container.

We are mounting the same web root as we are doing 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 any 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 work on.

The next service is MySQL, which uses the official MySQL image (`https://store.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:stretch-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.

You will notice that when the container starts, `wordpress/mysql` is populated with a few files. I do not recommend editing them using your local IDE.

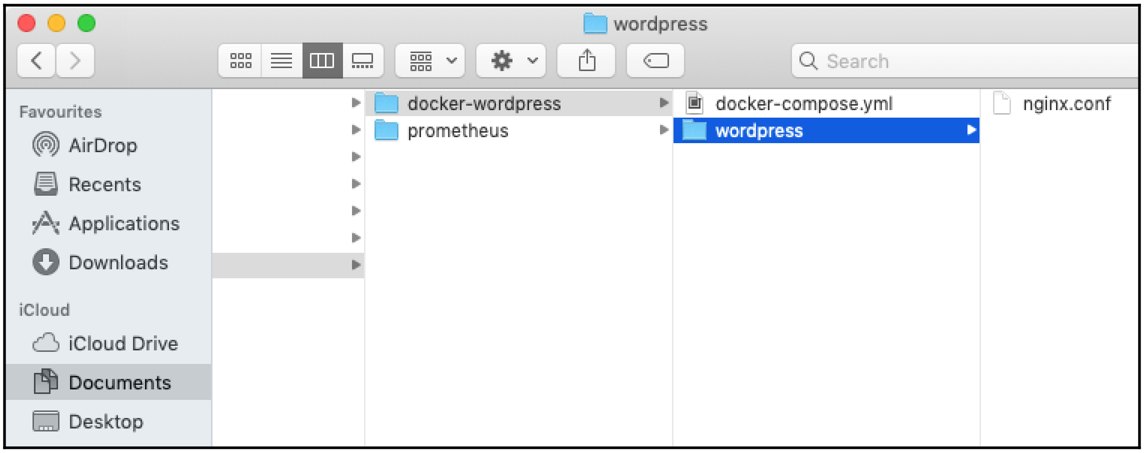
The final service is simply called `wp`. It differs from the other three services: this service will immediately exit when executed because there is no long-running process within the container. Instead of a long-running process, it provides access to the WordPress command-line tool in an environment that exactly matches our main `wordpress` container.

You will notice that we are mounting the web root as we have done on web and WordPress 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:

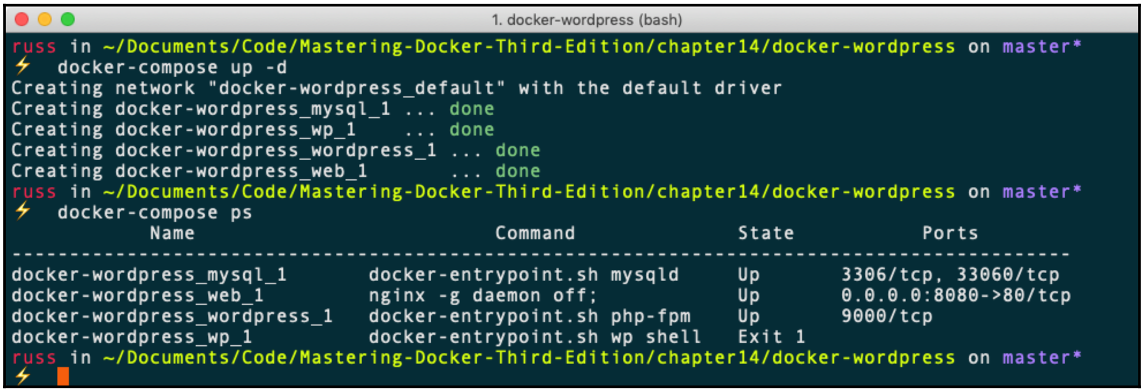
```
$ docker-compose pull
```

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:



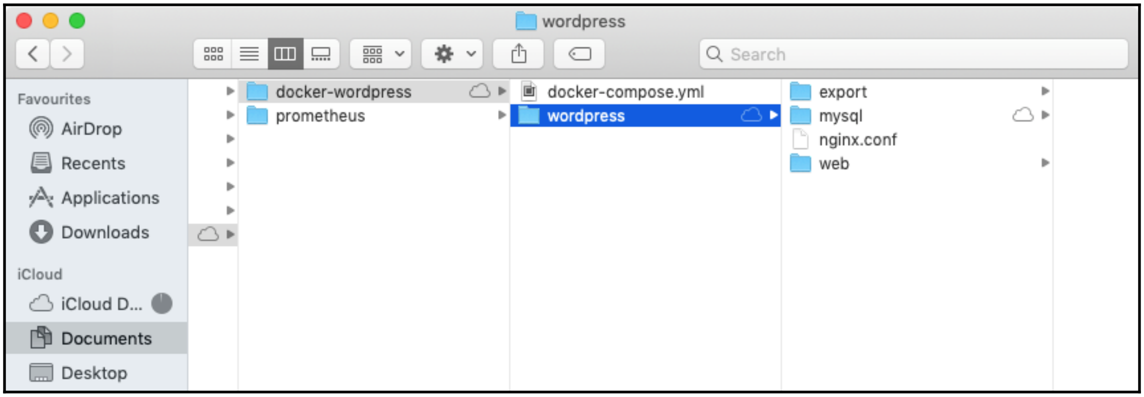
As you can see, we only have `nginx.conf` in there, which is part of the Git repository. Then we can use the following commands to start the containers and check their status:

```
$ docker-compose up -d
$ docker-compose ps
```

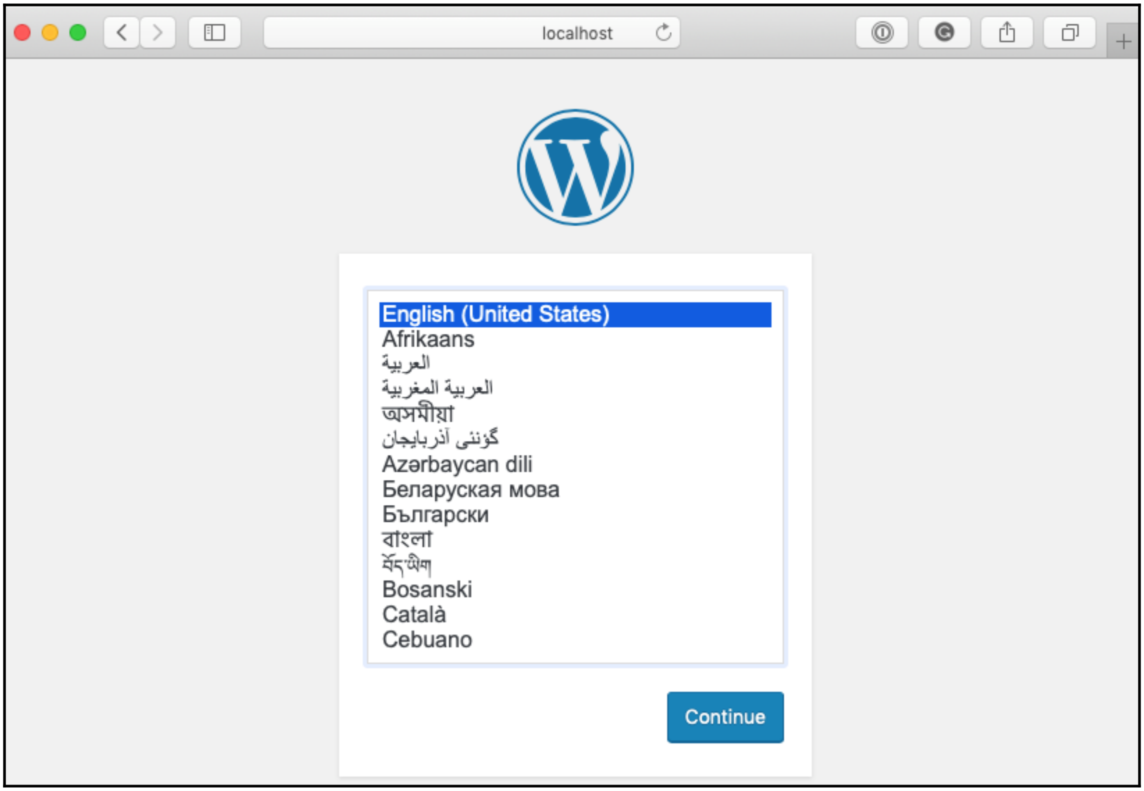


```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡  docker-compose up -d
Creating network "docker-wordpress_default" with the default driver
Creating docker-wordpress_mysql_1 ... done
Creating docker-wordpress_wp_1    ... done
Creating docker-wordpress_wordpress_1 ... done
Creating docker-wordpress_web_1       ... done
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡  docker-compose ps
            Name                             Command              State             Ports
---------------------------------------------------------------------------------------------
docker-wordpress_mysql_1       docker-entrypoint.sh mysqld     Up      3306/tcp, 33060/tcp
docker-wordpress_web_1         nginx -g daemon off;            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
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡
```

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, so that's fine:



Opening a browser and going to `http://localhost:8080/` should show you the standard WordPress pre-installation welcome page, where you can select the language you wish to use for your installation:

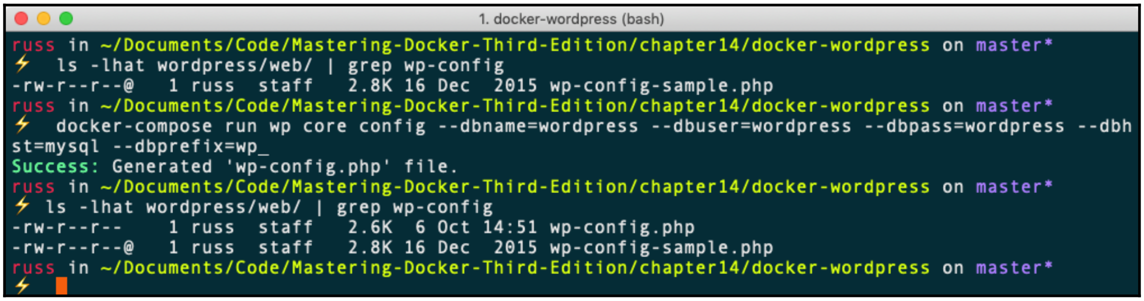


Do not click on **Continue**, as it will take you to the next screen of the GUI-based installation. Instead, return to your Terminal.

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:



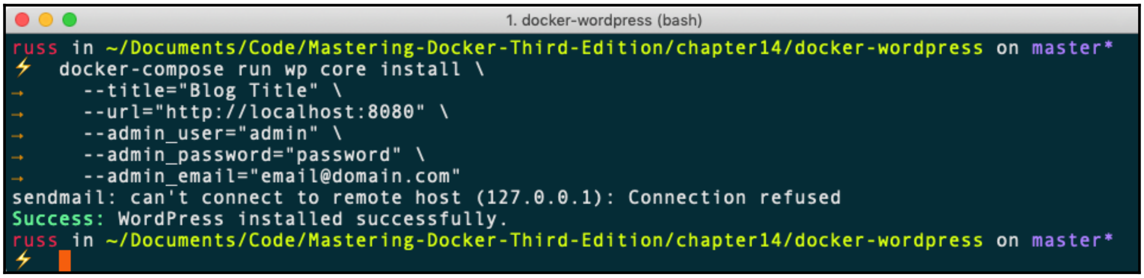
```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡  ls -lhat wordpress/web/ | grep wp-config
-rw-r--r--@   1 russ  staff   2.8K 16 Dec  2015 wp-config-sample.php
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡  docker-compose run wp core config --dbname=wordpress --dbuser=wordpress --dbpass=wordpress --dbh
st=mysql --dbprefix=wp_
Success: Generated 'wp-config.php' file.
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡ ls -lhat wordpress/web/ | grep wp-config
-rw-r--r--    1 russ  staff   2.6K  6 Oct 14:51 wp-config.php
-rw-r--r--@   1 russ  staff   2.8K 16 Dec  2015 wp-config-sample.php
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡
```

You will notice that in the command, we are passing the database details 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 a 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 about the email service; do not worry about that message, as this is only a local development environment. We are not too worried about emails leaving our WordPress installation:

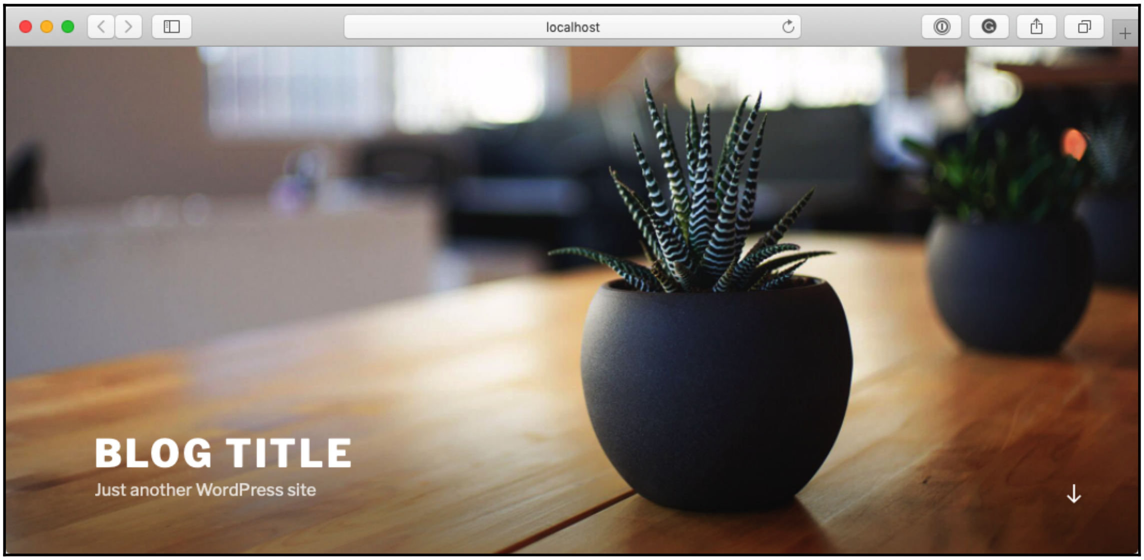


```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡  docker-compose run wp core install \
→     --title="Blog Title" \
→     --url="http://localhost:8080" \
→     --admin_user="admin" \
→     --admin_password="password" \
→     --admin_email="email@domain.com"
sendmail: can't connect to remote host (127.0.0.1): Connection refused
Success: WordPress installed successfully.
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡
```

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 password is `password`, and the user has an email of `email@domain.com`

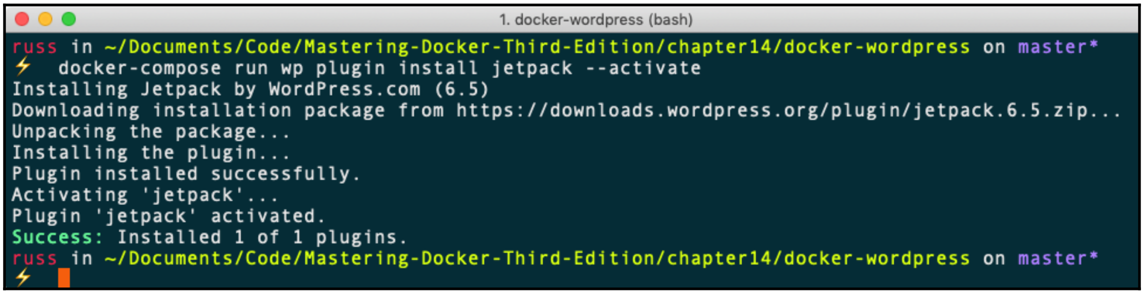
Going back to your browser and entering `http://localhost:8080/` should present you with a vanilla WordPress site:



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:

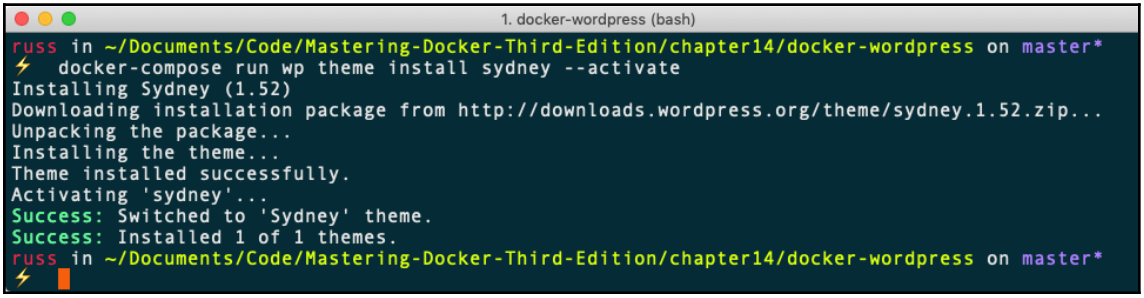


```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡  docker-compose run wp plugin install jetpack --activate
Installing Jetpack by WordPress.com (6.5)
Downloading installation package from https://downloads.wordpress.org/plugin/jetpack.6.5.zip...
Unpacking the package...
Installing the plugin...
Plugin installed successfully.
Activating 'jetpack'...
Plugin 'jetpack' activated.
Success: Installed 1 of 1 plugins.
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡
```

Then, install and enable the `sydney` theme:

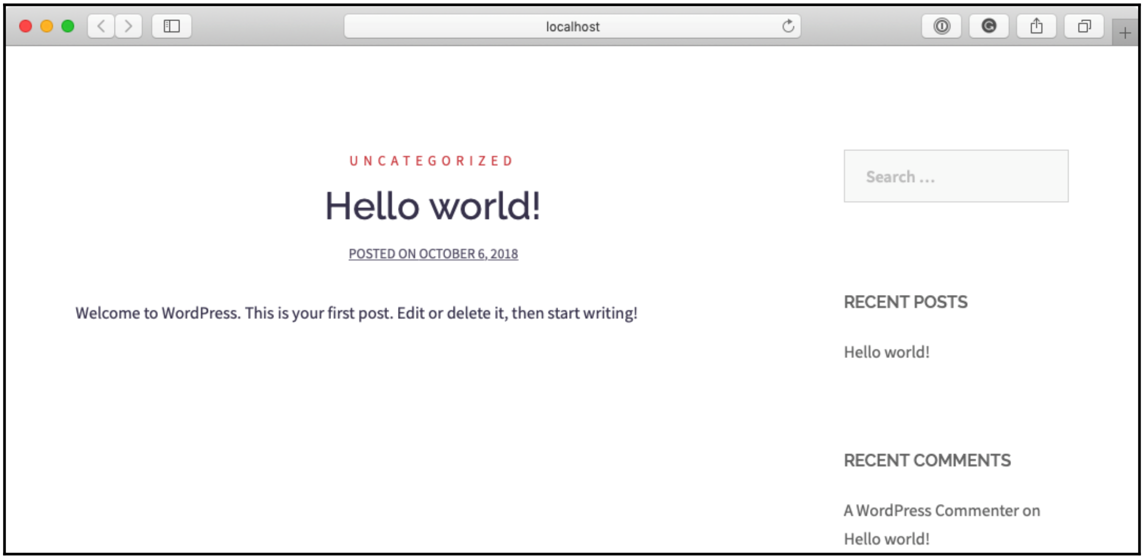
```
$ docker-compose run wp theme install sydney --activate
```

The output of the command is given here:



```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡  docker-compose run wp theme install sydney --activate
Installing Sydney (1.52)
Downloading installation package from http://downloads.wordpress.org/theme/sydney.1.52.zip...
Unpacking the package...
Installing the theme...
Theme installed successfully.
Activating 'sydney'...
Success: Switched to 'Sydney' theme.
Success: Installed 1 of 1 themes.
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡
```

Refreshing our WordPress page at `http://localhost:8080/` should show something like the following:



UNCATEGORIZED

Hello world!

POSTED ON OCTOBER 6, 2018

Welcome to WordPress. This is your first post. Edit or delete it, then start writing!

RECENT POSTS

Hello world!

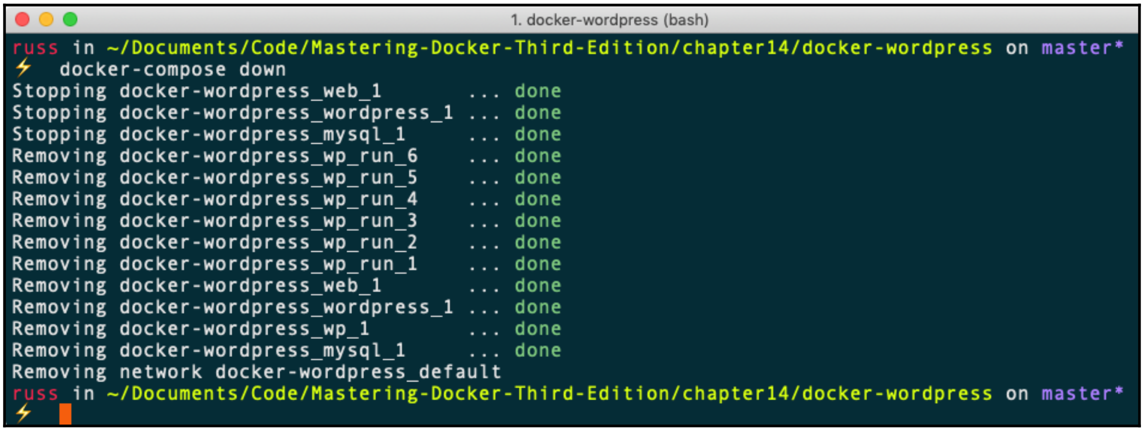
RECENT COMMENTS

A WordPress Commenter on Hello world!

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 in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡  docker-compose down
Stopping docker-wordpress_web_1       ... done
Stopping docker-wordpress_wordpress_1 ... done
Stopping docker-wordpress_mysql_1     ... done
Removing docker-wordpress_wp_run_6    ... done
Removing docker-wordpress_wp_run_5    ... done
Removing docker-wordpress_wp_run_4    ... done
Removing docker-wordpress_wp_run_3    ... done
Removing docker-wordpress_wp_run_2    ... done
Removing docker-wordpress_wp_run_1    ... 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
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡
```

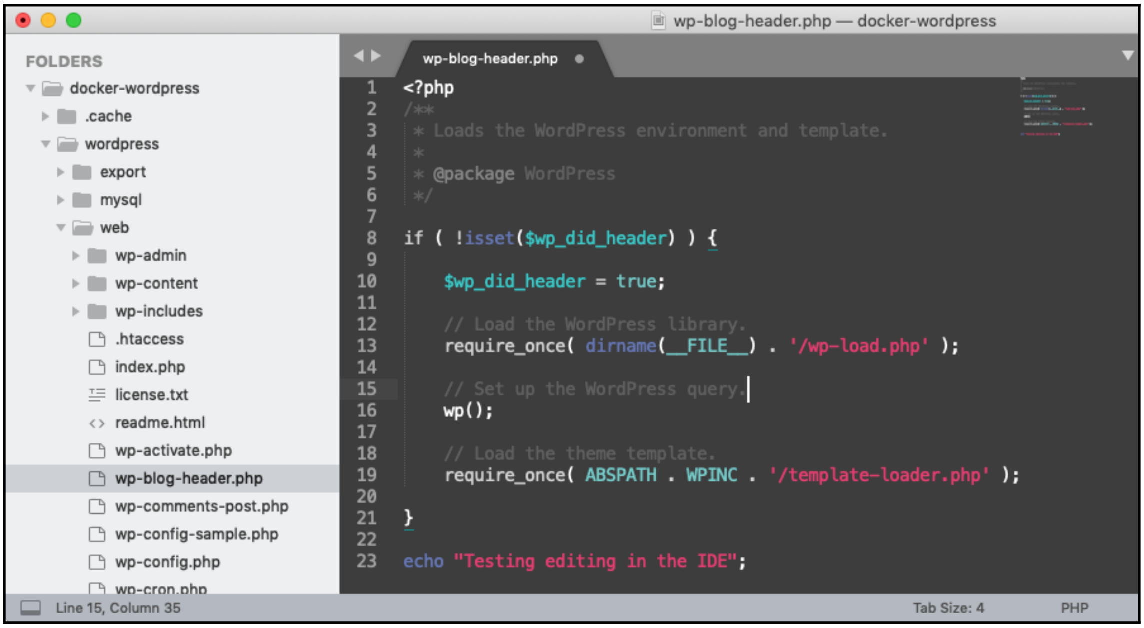
As our entire WordPress installation, including all of the files and database, is stored on our local machine, we should be able to run the following command to return to our WordPress site where we left it:

```
$ docker-compose up -d
```

Once you have confirmed it is up and running as expected by going to `http://localhost:8080/`, open the `docker-wordpress` folder in your desktop editor. I used Sublime Text. In your editor, open the `wordpress/web/wp-blog-header.php` file and add the following line to the opening PHP statement and save it:

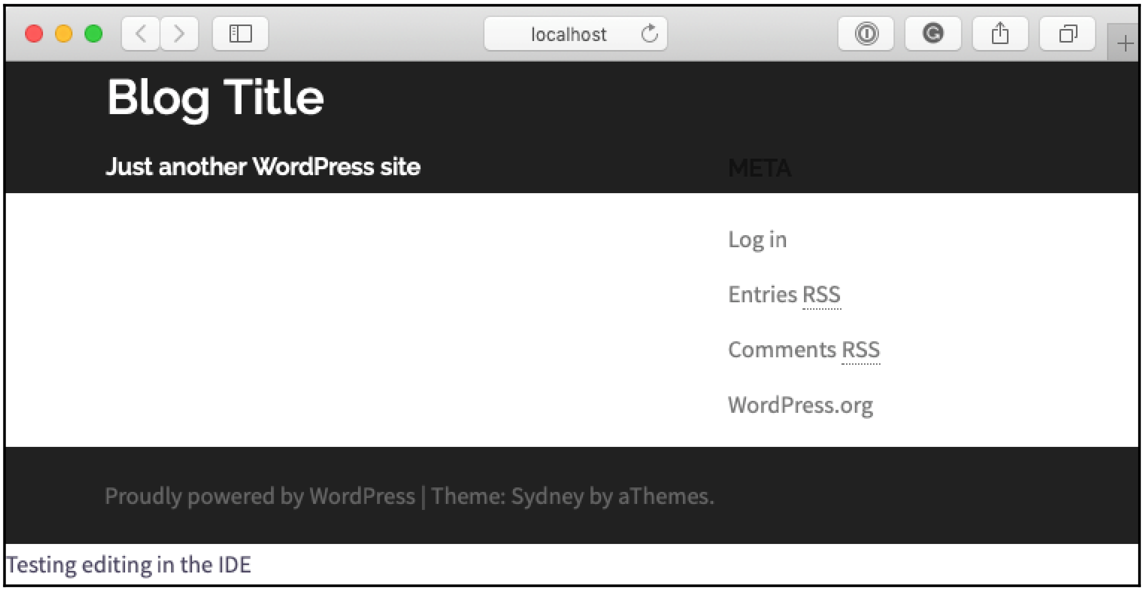
```
echo "Testing editing in the IDE";
```

The file should look something like the following:



```
<?php
/**
 * Loads the WordPress environment and template.
 *
 * @package WordPress
 */

if ( !isset($wp_did_header) ) {

    $wp_did_header = true;

    // Load the WordPress library.
    require_once( dirname(__FILE__) . '/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";
```

Once saved, refresh your browser and 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, as the text is is quite small):



localhost
Blog Title
Just another WordPress site
META
Log in
Entries RSS
Comments RSS
WordPress.org
Proudly powered by WordPress | Theme: Sydney by aThemes.

Testing editing in the IDE

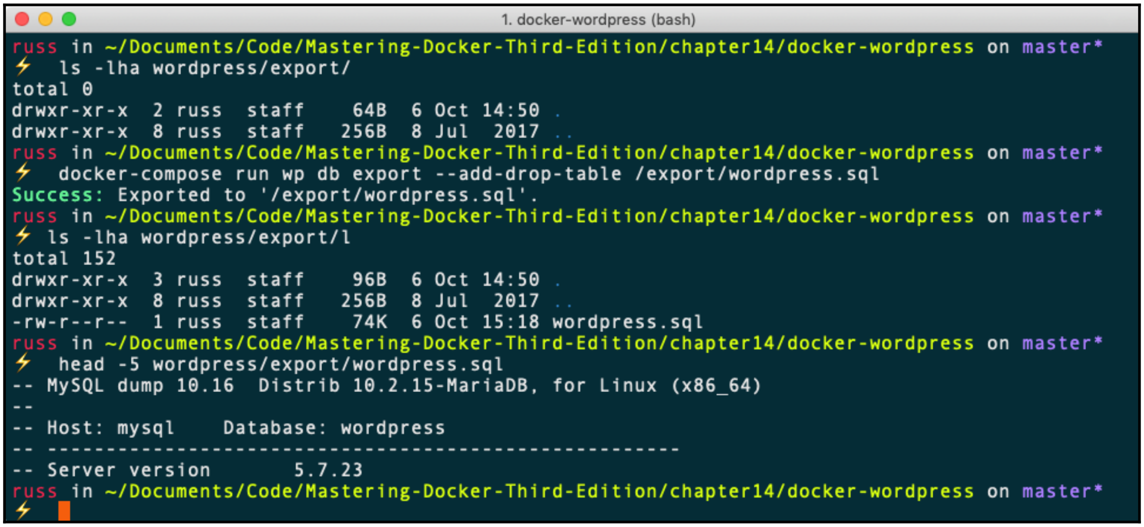
The final thing we are going to look at is why we had the `wordpress/export` folder mounted on the `wp` container.

As already mentioned earlier in the chapter, you shouldn't be really 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, it is not considered as best practice. Because of this, we have mounted the export folder to allow us to use WP-CLI 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
```

The following Terminal output shows the export and also the contents of `wordpress/export` before and after, and finally, the top few lines of the MySQL dump:

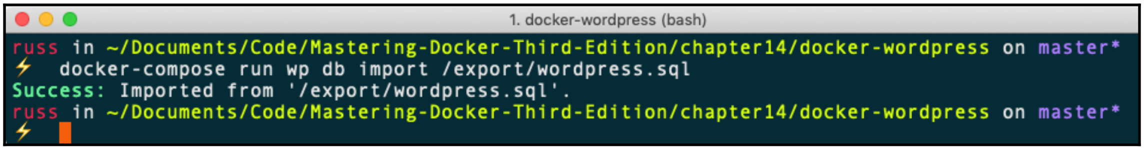


```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡  ls -lha wordpress/export/
total 0
drwxr-xr-x  2 russ  staff    64B  6 Oct 14:50 .
drwxr-xr-x  8 russ  staff   256B  8 Jul  2017 ..
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡  docker-compose run wp db export --add-drop-table /export/wordpress.sql
Success: Exported to '/export/wordpress.sql'.
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡ ls -lha wordpress/export/l
total 152
drwxr-xr-x  3 russ  staff    96B  6 Oct 14:50 .
drwxr-xr-x  8 russ  staff   256B  8 Jul  2017 ..
-rw-r--r--  1 russ  staff    74K  6 Oct 15:18 wordpress.sql
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡  head -5 wordpress/export/wordpress.sql
-- MySQL dump 10.16  Distrib 10.2.15-MariaDB, for Linux (x86_64)
--
-- Host: mysql    Database: wordpress
-- ------------------------------------------------------
-- Server version       5.7.23
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡
```

If I needed to, because, say, I had made a mistake during development, I could roll back to that version of the database by running the following command:

```
$ docker-compose run wp db import /export/wordpress.sql
```

The output of the command is given here:



```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡  docker-compose run wp db import /export/wordpress.sql
Success: Imported from '/export/wordpress.sql'.
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/docker-wordpress on master*
⚡
```

As you have seen, we have installed WordPress, interacted with it both using WP-CLI and the browser, edited the code, and also backed up and restored the database, all without having to install or configure nginx, PHP, MySQL, or WP-CLI. 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 it is running in the exact environment as our own installation.

Finally, as we're using official images from the Docker Store, we know we can safely ask to have them deployed into production as they have been built with Docker's best practices in mind.

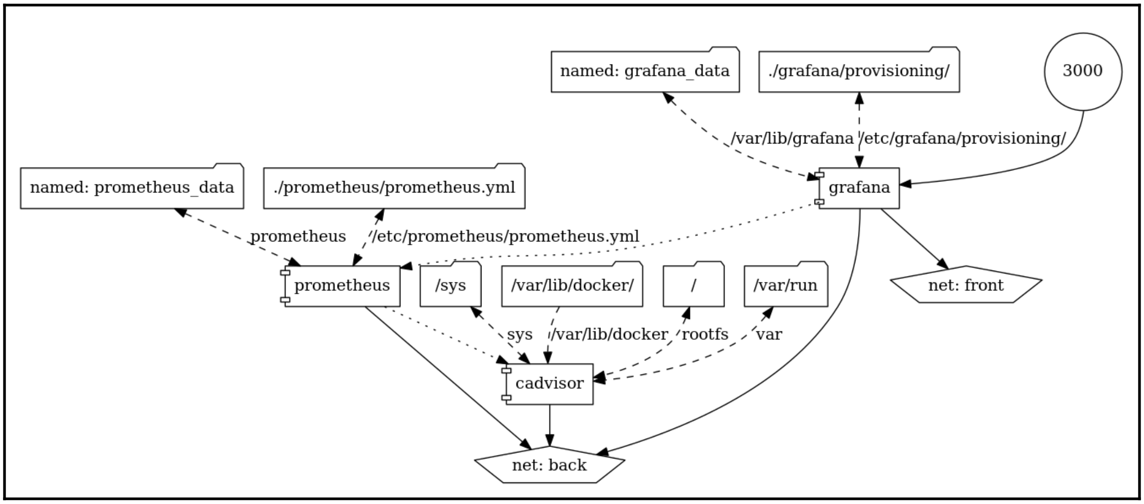
TIP Don't forget to stop and remove your WordPress containers by running `docker-compose down`.

Monitoring

Next, we are going to take a look at monitoring our containers and also Docker hosts. 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 kept.

While 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, it is not too helpful if you need to look back at a problem: maybe you 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 the visualization:



As you can see, there is a lot going on. The three services we are running are:

- **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` is a project released by Google. As you can see from 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 11`, *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.

`cadvisor` 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 from an easy-to-use interface. However, it doesn't keep more than 5 minutes' worth of metrics.

As we are attempting to record metrics that can be available hours or even days later, having no more than 5 minutes of metrics means that we are going to have to use additional tools to record the metrics it processes. `cadvisor` exposes the information we want to record our containers as structured data at the following endpoint: `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 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 also 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 in there as well as Prometheus itself defined. 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:

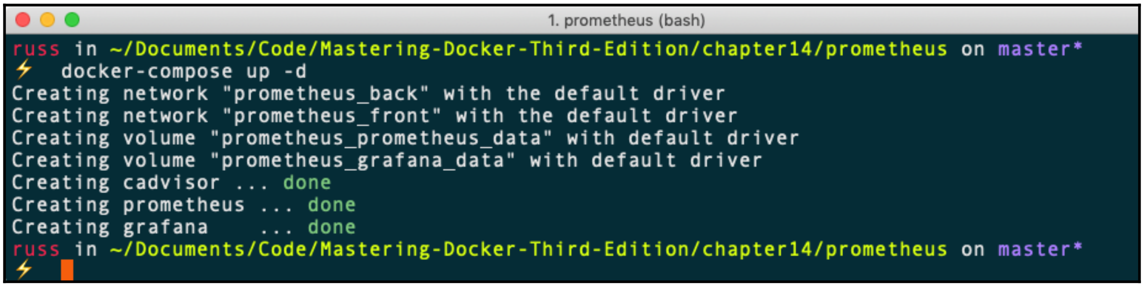
```
GF_SECURITY_ADMIN_USER=admin
GF_SECURITY_ADMIN_PASSWORD=password
GF_USERS_ALLOW_SIGN_UP=false
```

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 four 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 four services:

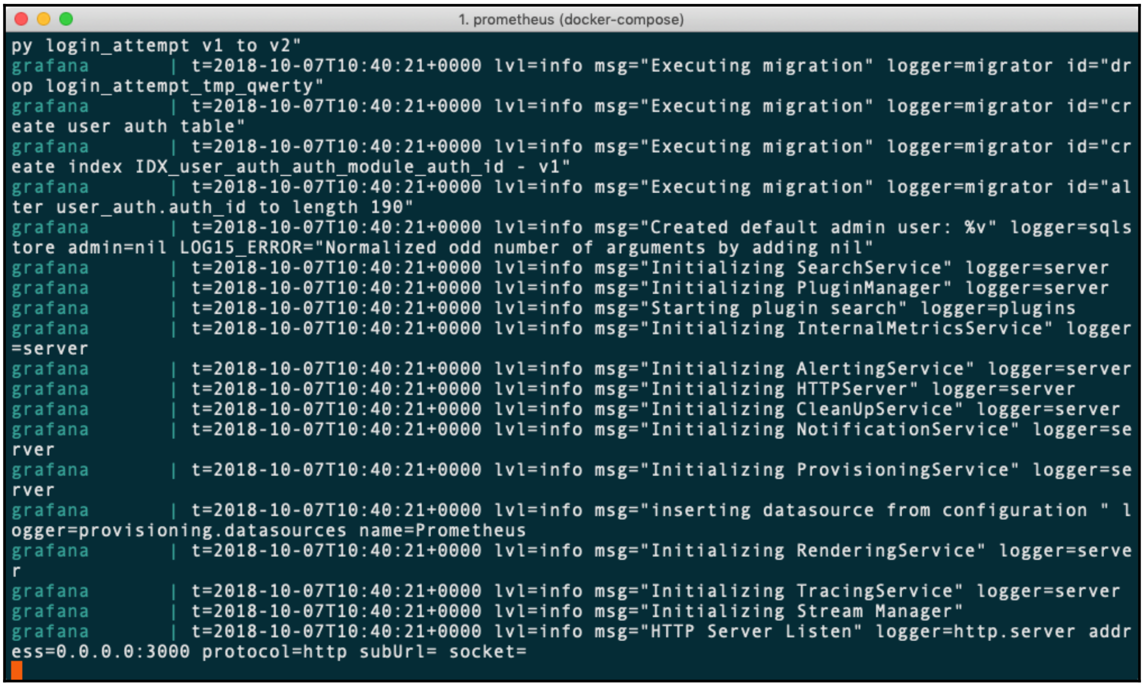


```
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/prometheus on master*
⚡  docker-compose up -d
Creating network "prometheus_back" with the default driver
Creating network "prometheus_front" with the default driver
Creating volume "prometheus_prometheus_data" with default driver
Creating volume "prometheus_grafana_data" with default driver
Creating cadvisor ... done
Creating prometheus ... done
Creating grafana    ... done
russ in ~/Documents/Code/Mastering-Docker-Third-Edition/chapter14/prometheus on master*
⚡
```

You may be tempted to go immediately to your Grafana dashboard. If you did so, you would not see anything, as Grafana takes a few minutes to initialize itself. You can follow its progress by following the logs:

```
$ docker-compose logs -f grafana
```

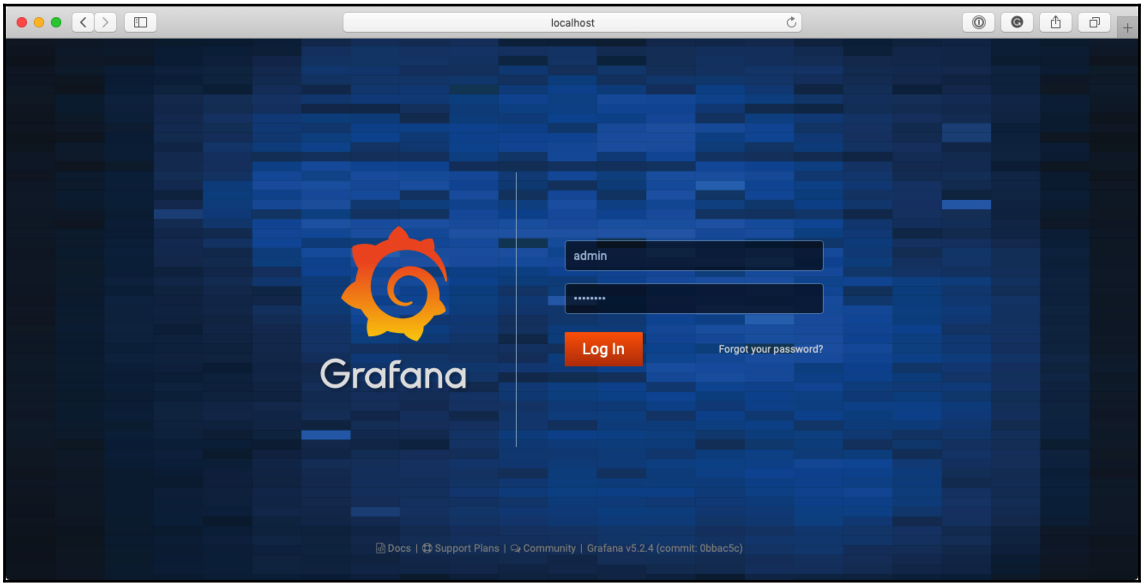
The output of the command is given here:



```
py login_attempt v1 to v2"
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="Executing migration" logger=migrator id="dr
op login_attempt_tmp_qwerty"
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="Executing migration" logger=migrator id="cr
eate user auth table"
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="Executing migration" logger=migrator id="cr
eate index IDX_user_auth_auth_module_auth_id - v1"
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="Executing migration" logger=migrator id="al
ter user_auth.auth_id to length 190"
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="Created default admin user: %v" logger=sqls
tore admin=nil LOG15_ERROR="Normalized odd number of arguments by adding nil"
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="Initializing SearchService" logger=server
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="Initializing PluginManager" logger=server
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="Starting plugin search" logger=plugins
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="Initializing InternalMetricsService" logger
=server
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="Initializing AlertingService" logger=server
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="Initializing HTTPServer" logger=server
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="Initializing CleanUpService" logger=server
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="Initializing NotificationService" logger=se
rver
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="Initializing ProvisioningService" logger=se
rver
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="inserting datasource from configuration " l
ogger=provisioning.datasources name=Prometheus
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="Initializing RenderingService" logger=serve
r
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="Initializing TracingService" logger=server
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="Initializing Stream Manager"
grafana        | t=2018-10-07T10:40:21+0000 lvl=info msg="HTTP Server Listen" logger=http.server addr
ess=0.0.0.0:3000 protocol=http subUrl= socket=
```

Once you see the `HTTP Server Listen` message, Grafana will be available. With Grafana 5 you can now import data sources and dashboards, which is why we are mounting `./grafana/provisioning/` to `/etc/grafana/provisioning/`. This folder contains the configuration which automatically configures Grafana to talk to our Prometheus service and also imports the dashboard, which will display the data that Prometheus is scraping from cadvisor.

Open your browser and enter `http://localhost:3000/`, and you should be greeted with a login screen:

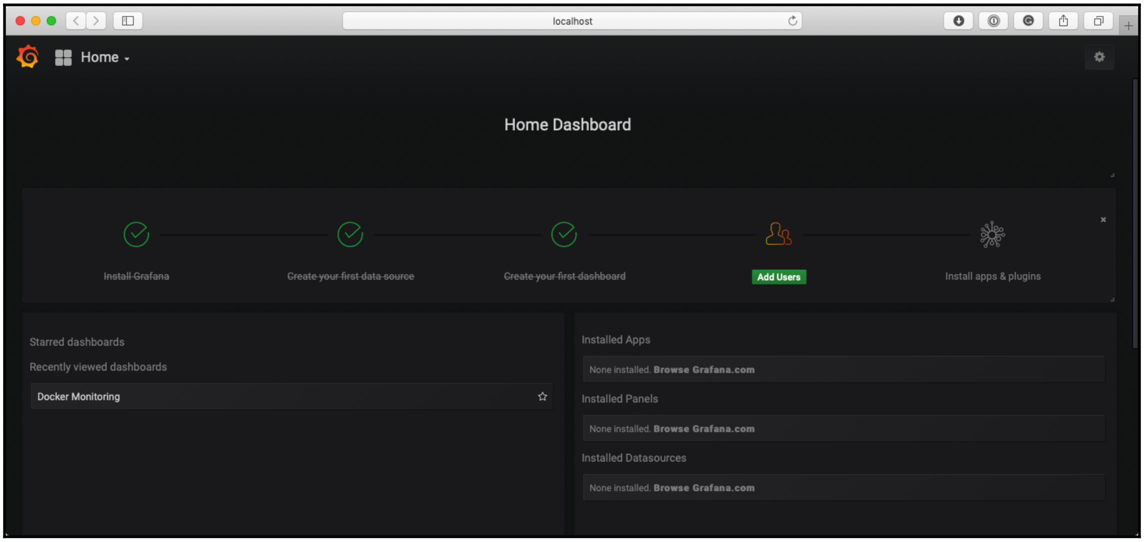


admin

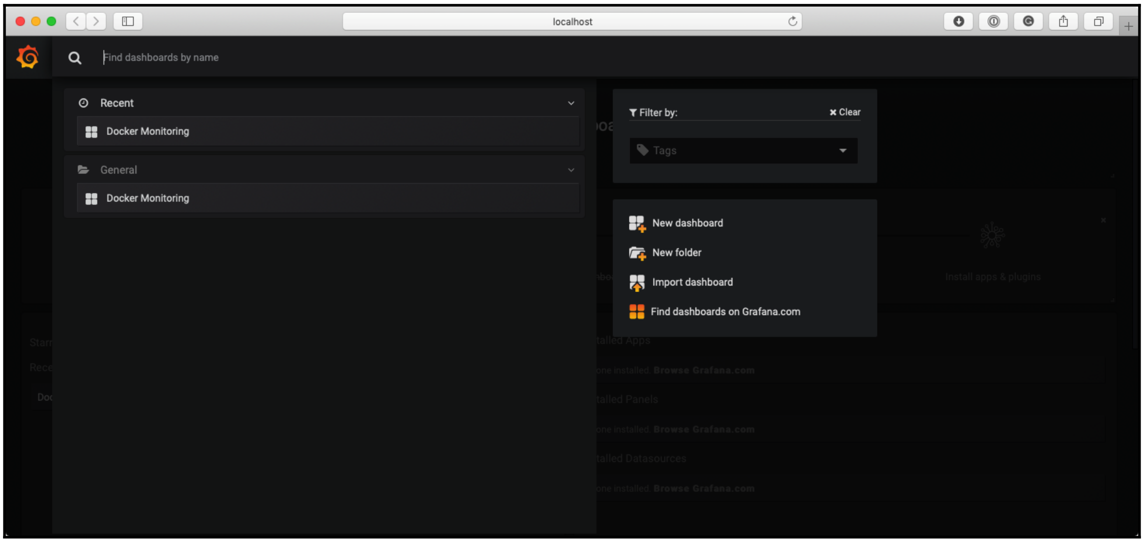
••••••••

localhost
Grafana
Log In
Forgot your password?
Docs | Support Plans | Community | Grafana v5.2.4 (commit: 0bbac5c)

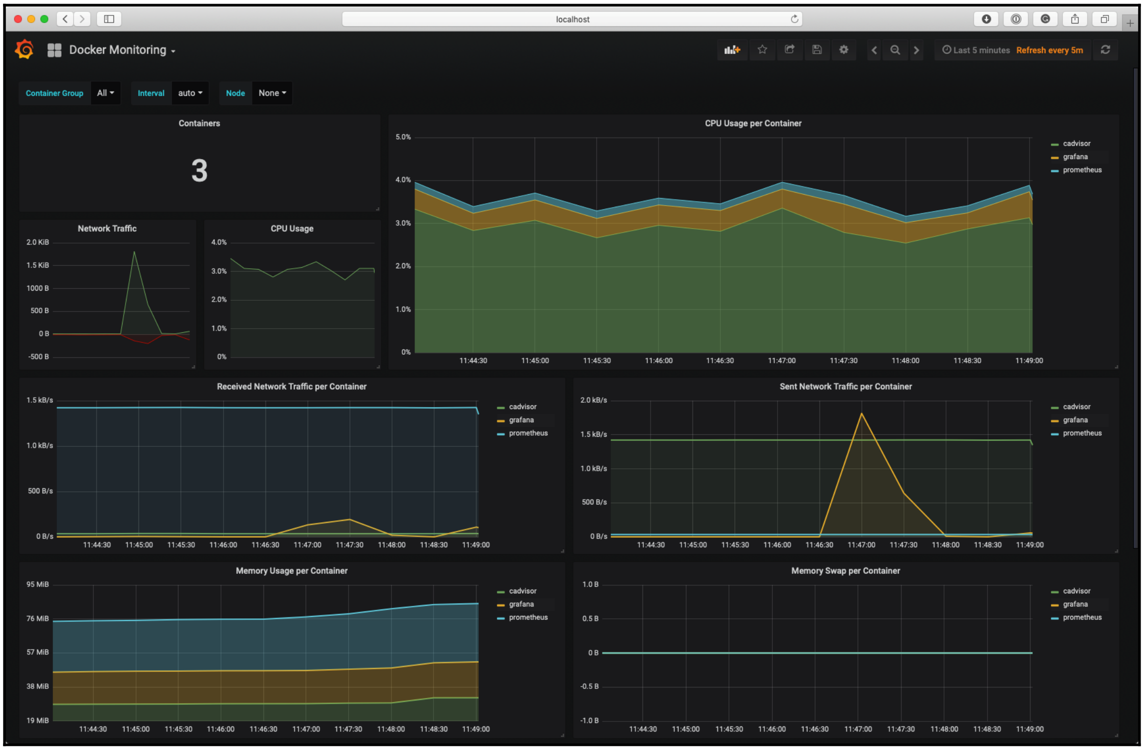
Enter the **User** as `admin` and the **Password** as `password`. Once logged in, if you have configured the data source, you should see the following page:


localhost
Home
Home Dashboard
~~Install Grafana~~
~~Create your first data source~~
~~Create your first dashboard~~
Add Users
Install apps & plugins
Starred dashboards
Recently viewed dashboards
Docker Monitoring
Installed Apps
None installed. Browse Grafana.com
Installed Panels
None installed. Browse Grafana.com
Installed Datasources
None installed. Browse Grafana.com

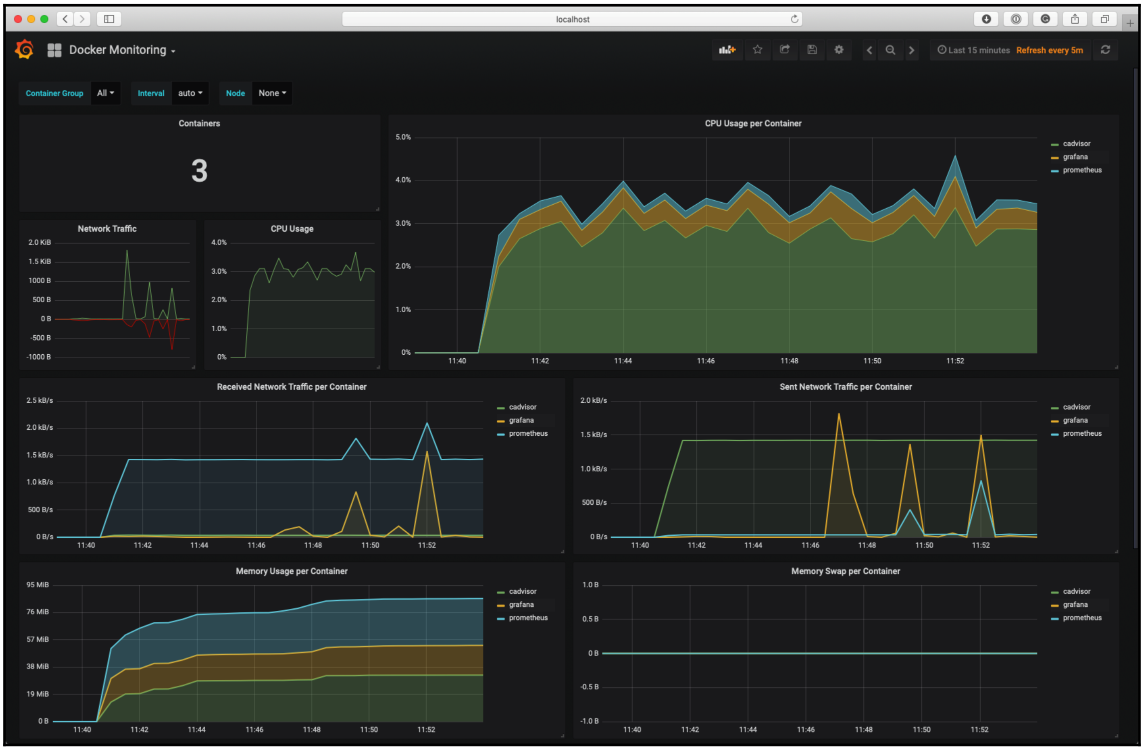
As you can see, the initial steps of **Install Grafana** | **Create your first data source** | **Create your first dashboard** have all been executed, leaving just the remaining two. For now, we will ignore these. Clicking on the **Home** button in the top left will bring up a menu that lists the available dashboards:



As you can see, we have one called **Docker Monitoring**. Clicking on it will take you to the following page:



As you can see from the timing information on 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 obviously is more than the five minutes that cadvisor is recording:



I have already mentioned that this is a complex solution; eventually, Docker will expand the recently released built-in endpoint, which presently only exposes information about the 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/thirdparty/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 listed. In fact, you may already be using them, so it would be easy for you when expanding your configuration to take into account when monitoring your containers.

Once you have finished exploring your Prometheus installation, don't forget to remove it by running the following command:

```
$ docker-compose down --volumes --rmi all
```

This removes all of the containers, volumes, images, and network.

Extending to external platforms

We have already looked at how we can extend to some other external platforms using tools such as Docker Machine, Docker Swarm, Docker for Amazon Web Services, and Rancher to launch clusters and also clusters and container services from public cloud services, such as Amazon Web Services, Microsoft Azure, and DigitalOcean.

Heroku

Heroku is a little different than the other cloud services, as it is a considered a **Platform as a service** (**PaaS**). Instead of deploying containers on it, you link your containers to that Heroku platform, from which it will be running a service, such as PHP, Java, Node.js, or Python. So, you can run your Rails application on Heroku and then attach your Docker container to that platform.

We will not be covering installing Heroku here as it is a little off topic. Please see the *Further readin*g section of the chapter for more details on Heroku.

The way you can use Docker and Heroku together is to create your application on the Heroku platform, and then in your code, you will have something similar to the following:

```
{
  "name": “Application Name",
  "description": “Application to run code in a Docker container",
  "image": “<docker_image>:<tag>”,
 "addons": [ "heroku-postgresql" ]
}
```

To take a step back, we first need to install the plugin to be able to get this functionality working. Simply run the following command:

```
$ heroku plugins:install heroku-docker
```

Now, if you are wondering what image you can or should be using from the Docker Hub, Heroku maintains a lot of images you can use in the preceding code:

- `heroku/nodejs`
- `heroku/ruby`
- `heroku/jruby`
- `heroku/python`
- `heroku/scala`
- `heroku/clojure`
- `heroku/gradle`
- `heroku/java`
- `heroku/go`
- `heroku/go-gb`

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. This is due to the sheer number of options that are available, so it would be impossible to cover them all. Also, you should already have a good idea based on the previous sections and chapters on what would work best for you.

Instead, we are going to be looking at some questions you should be asking yourself when planning your environments.

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 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 a security implication with you having a mixture of isolated and non-isolated processes on a single host, but it can also cause performance issues as you are not able to add resource limits to your non-containerized 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 get 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 are you 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?

For example, you could use just ports for a website, everything on ports `80` and `443` to the container that is configured by Docker, to accept traffic on those ports. Using virtual host routing means that you can route `domain-a.com` to `container a` and then `domainb.com` to `container b`. Both `domain-a.com` and `domain-b.com` can point toward the same IP address and port.

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

Compatibility

Even though an application works 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 out of ten times, 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 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 building a proof of concept and trying 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 all limit your trial and error.

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://traefik.io/`
- HAProxy: `https://www.haproxy.org/`
- NGINX: `https://nginx.org/`
- Heroku: `https://www.heroku.com`

Other externally hosted Docker monitoring platforms include the following:

- Sysdig Cloud: `https://sysdig.com/`
- Datadog: `http://docs.datadoghq.com/integrations/docker/`
- CoScale: `http://www.coscale.com/docker-monitoring`
- Dynatrace: `https://www.dynatrace.com/capabilities/microservices-and-container-monitoring/`
- 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://www.sysdig.org`
- Zabbix: `https://github.com/monitoringartist/zabbix-docker-monitoring`

14
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. Let's start by discussing the Moby Project.

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.

So, how did I answer? After a few days of getting puzzled looks, I settled on the following answer:

Moby Project is the collective name for an open source project that collects several libraries used to build container-based systems. The project comes with its own framework for combining these libraries into a usable system and also a reference system called Moby Origin; think of this as a "Hello World" that allows you to build and even customize your own Docker.

One of two things happened after I gave this answer; typically, the response was *but what does that actually mean?*. I responded by saying:

Moby Project is the open source playground for Docker (the company) and anyone else who wishes to contribute to the project to develop new and extend existing features to the libraries and frameworks that go to make up container-based systems in a public forum. One output of this is the bleeding-edge container system called Moby Origin and the other is Docker (the product), which is delivered as the open source community edition or the commercially supported 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 do 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, 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?* Well, at the time of writing this book, the project is still very much in its infancy. In fact, work is still ongoing to transition all of the components required for the Moby Project from the main Docker projects.

The only real, usable component of the project as I write this is *LinuxKit*, which is the framework that pulls together all of the libraries and outputs a bootable system that is capable of running containers.

Due the extremely fast pace of this project, I am not going to give any examples on how to use LinuxKit or go into any more detail about 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 exciting development:

- 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-to's, at: `https://twitter.com/moby/`
- The home of LinuxKit, which contains examples and instructions on how to get started, at: `https://github.com/linuxkit/`

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 helping with the Docker code. Since Docker is all open source, you can download the code to your local machine and work on new features and present them as pull requests back to Docker. 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/`), which we spoke about in the previous section. 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 Docker documentation at `https://docs.docker.com/project/who-written-for/`.

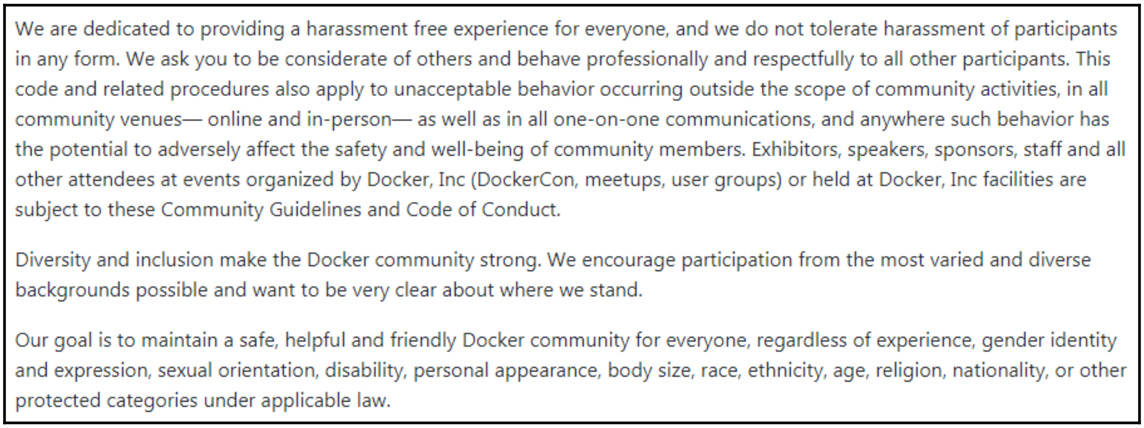
As you may have already guessed, you do not need much to get a development environment up-and-running as a lot of development is done within containers. For example, other than having a GitHub account, Docker lists the following three pieces of software as the bare minimum:

- 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://docs.docker.com/opensource/project/software-required/` and for Windows users at: `https://docs.docker.com/opensource/project/software-req-win/`.

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://docs.docker.com/opensource/code/` as well as the more detailed contribution workflow documentation at: `https://docs.docker.com/opensource/workflow/make-a-contribution/`.

Docker has 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/`.

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 back 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://community.docker.com/` ; 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.

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 9`, *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 2018, the Linux Foundation reported that CNCF had 238 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 two graduated projects, both of which we have discussed in previous chapters. These are arguably also the two most well-known out of the projects that are maintained by the foundation, and they 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 13`, *Docker Workflows*, it is a real-time monitoring and alerting system that's backed by a powerful time-series database engine.

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 publically via the `GOVERNANCE.md` and `OWNERS.md` files.
- Publically 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/`): This is the first of two tracing projects which now come under the CNCF umbrella. Rather than being an application, you download and use it is a set of libraries and APIs which let you build in behavioral tracking and monitoring into your microservices-based 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.
- **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.
- **Containerd** (`https://containerd.io`): We briefly mentioned Containerd in `Chapter 1`, *Docker Overview*, as being one of the open source projects which Docker has been working on. It is a standard container runtime which allows developers to embed a runtime that can manage both Docker and also OCI compliant images in their platforms or applications.
- **Rkt** (`https://github.com/rkt/rkt`): Rkt is an alternative to Docker's container engine. Rather than using a daemon to manage containers on the host system, Rkt uses the command line to launch and manage containers. It was donated to the CNCF by CoreOS, who is now owned by Red Hat.

- **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, Rkt, 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.
- **Envoy** (`https://www.envoyproxy.io`): Originally created inside Lyft and in use by 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.
- **Jaeger** (`https://jaegertracing.io`): This is the second tracing system in the list. Unlike OpenTracing, it is a fully distributed tracing system that was originally developed by Uber to monitor it 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.
- **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 which 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.
- **Vitess** (`https://vitess.io`): Vitess 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.
- **CoreDNS** (`https://coredns.io`): This is a small, flexible, extendablem 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.
- **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.

- **Helm** (`https://www.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.
- **Rook** (`https://rook.io`): Currently, Rook is in its early stages of development, focusing on providing an orchestration layer for managing Ceph, Red Hat's distributed storage system, on Kubernetes. Eventually, it will expand as to support other distributed blocks and object storage systems.

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 solving 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 590 cards with a total of 1,227,036 stars, a market cap of $6.52T, and funding of $16.3B.

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, which 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

1. The Docker Store: `https://store.docker.com/`
2. `$ docker image pull nginx`
3. The Moby Project
4. Seven months
5. `$ docker container help`

Chapter 2, Building Container Images

1. False; it is used to add metadata to the image
2. You can append `CMD` to an `ENTRYPOINT`, but not the overlay 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 does not map a port on the host machine

Chapter 3, Storing and Distributing Images

1. False; there is also the Docker Store
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

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 terminating 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

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 the Docker Hub or other registries, the Docker App command-line tool when can render working Docker Compose files from the data contained within the image

Chapter 6, Windows Containers

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 7, Docker Machine

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. Docker Machine is a command-line tool that can be used to launch Docker hosts on a number of platforms and technologies, in a simple and consistent way

Chapter 8, Docker Swarm

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, Docker and Kubernetes

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 10, Running Docker in Public Clouds

1. False; they launch Docker Swarm clusters
2. When using Amazon Fargate, you do not have to launch Amazon EC2 instances to run your Amazon ECS cluster on
3. The container options are listed under the Azure Web Application service
4. Using the command `kubectl create namespace sock-shop`
5. By running `kubectl -n sock-shop describe services front-end-lb`

Chapter 11, Portainer - A GUI for Docker

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 12, Docker Security

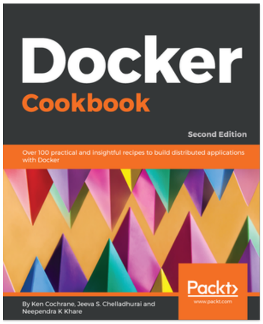
1. You would add the `--read-only` flag; or, if you want to make a volume read-only, you would add `:ro`
2. In an ideal world, you would only be running a single process per container
3. By running the Docker Bench Security application
4. The socket file for Docker, which can be found at `/var/run/docker.sock`; and also, if your host system is running Systemd, `/usr/lib/systemd`
5. False; Quay scans both public and private images

Chapter 13, Docker Workflows

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 five 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:


Docker
Cookbook
Second Edition
Over 100 practical and insightful recipes to build distributed applications with Docker
Packt>
www.packt.com
By Ken Cochrane, Jeeva S. Chelladhurai and Neependra K Khare

Docker Cookbook - Second Edition
Ken Cochrane, Jeeva S. Chelladhurai, Neependra K Khare

ISBN: 978-1-78862-686-6

- Install Docker on various platforms
- Work with Docker images and containers
- Container networking and data sharing
- Docker APIs and language bindings
- Various PaaS solutions for Docker
- Implement container orchestration using Docker Swarm and Kubernetes
- Container security
- Docker on various clouds

Gabriel N. Schenker
Learn Docker – Fundamentals of Docker 18.x
Everything you need to know about containerizing your applications and running them in production
Packt

Learn Docker - Fundamentals of Docker 18.x
Gabriel N. Schenker

ISBN: 978-1-78899-702-7

- Containerize your traditional or microservice-based application
- 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 via encapsulation, networks, and secrets
- Know your options when deploying your containerized app into the cloud

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