Russ McKendrick, Pethuru Raj,
Jeeva S. Chelladhurai, Vinod Singh

Docker Bootcamp

Less is more with Docker

Packt>

Docker Bootcamp

Less is more with Docker

Russ McKendrick

Pethuru Raj

Jeeva S. Chelladhurai

Vinod Singh

BIRMINGHAM - MUMBAI

Docker Bootcamp

First published: April 2017

Production reference: 1250417

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

ISBN 978-1-78728-698-6

www.packtpub.com

Credits

Authors
Russ McKendrick
Pethuru Raj
Jeeva S. Chelladhurai
Vinod Singh

Reviewer
Jeeva S. Chelladhurai

Commissioning Editor
Kartikey Pandey

Acquisition Editor
Prachi Bisht

Content Development Editor
Mamata Walkar

Technical Editors
Naveenkumar Jain

Copy Editor
Safis Editing

Project Coordinator
Kinjal Bari

Proofreader
Safis Editing

Indexer
Tejal Daruwale Soni

Graphics
Kirk D'Penha

Production Coordinator
Shantanu Zagade

Cover Work
Shantanu Zagade

About the Authors

Russ McKendrick is an experienced solution architect who has been working in IT and related industries for the better part of 24 years. During his career, he has had varied responsibilities in many different sectors, ranging from looking after an entire IT infrastructure to providing first-line, second-line, and senior support in both client-facing and internal teams for small and large organizations.

Russ works almost exclusively with Linux, using open source systems and tools across both dedicated hardware and virtual machines hosted in public and private clouds at Node4 Limited, where he heads up the Open Source Solutions team.

In his spare time, he has written three books (including this one) on Docker. Monitoring Docker and Extending Docker which are both available now from Packt, as well contributing to Monitoring and Management With Docker and Containers which was published by The New Stack. He also buys way too many vinyl records.

LinkedIn: `https://in.linkedin.com/in/russmckendrick`

GitHub: `https://github.com/russmckendrick`

Personal Blog: `https://media-glass.es/`

Dockerhub: `https://hub.docker.com/u/russmckendrick/`

Packt: `https://www.packtpub.com/books/info/authors/russ-mckendrick`

Pethuru Raj, PhD has been working as a cloud architect in the IBM Global Hybrid Cloud Center of Excellence (CoE), IBM India Bangalore for the last four years. Previously he worked as TOGAF-certified enterprise architecture (EA) consultant in Wipro Consulting Services (WCS) Division, Bangalore for 10 years. He also had a fruitful stint (2 years) as a lead architect in the corporate research (CR) division of Robert Bosch, India. He has gained more than 16 years of IT industry experience. He finished the CSIR-sponsored PhD degree in Anna University, Chennai and continued the UGC-sponsored postdoctoral research in the department of Computer Science and Automation, Indian Institute of Science, Bangalore. Thereafter, he was granted a couple of international research fellowships (JSPS and JST) to work as a research scientist for 3.5 years in two leading Japanese universities. Totally he gained 8 years of research experience. He has authored 7 books thus far and he focuses on some of the emerging technologies such as:

- Software-defined Clouds (SDC)
- Big, Fast, Streaming and IoT Data Analytics
- Docker-enabled containerization
- Microservices architecture (MSA)
- Cognitive Clouds
- Smarter Cities Technologies and Tools
- IoT Edge/Fog Analytics

He has published more than 30 research papers in peer-reviewed journals such as IEEE, ACM, Springer-Verlag, Inderscience, etc.

Home Page: `www.peterindia.net`

LinkedIn Profile: `https://www.linkedin.com/in/peterindia`

Personal Email: `peterindia@gmail.com`

Jeeva S. Chelladhurai has been working as a DevOps specialist at the IBM GTS Labs for the last 9 years. He has more than 20 years of IT industry experience. He has technically managed and mentored diverse teams across the globe in envisaging and building pioneering telecommunication products. He specializes in DevOps, Automation and cloud solution delivery, with a focus on data center optimization, software-defined environments (SDEs), and distributed application development, deployment, and delivery using the newest Docker technology. Jeeva is also a strong proponent of the agile methodologies, DevOps, and IT automation. He holds a master's degree in computer science from Manonmaniam Sundaranar University and a graduation certificate in project management from Boston University, USA. Besides his official responsibilities, he writes book chapters and authors research papers. He has been instrumental in crafting reusable technical assets for IBM solution architects and consultants. He speaks in technical forums on DevOps technologies and tools. His Linked in profile can be found at `https://www.linkedin.com/in/JeevaChelladhurai`

Vinod Singh is a seasoned technical professional who has worked for two decades with software industry. Currently he is a senior cloud architect with IBM's cloud flagship offering Bluemix supporting customers across the world. Vinod's experience with networking and data communication spans software design, development and testing. The Cloud, Cognitive, and Linux are his passions and he feels Cognitive computing is once again going to change the world. Vinod's experience with the latest design thinking techniques, agile & lean methods, and extreme programing was very fruitful and has been a tremendous help in making cloud deals across the world.

Vinod is a regular speaker at IBM's internal conferences, IEEE conferences, and technology meetups. Vinod's latest day job revolves around IBM BlueMix, Cloud Foundry, Softlayer, OpenStack, Amazon AWS.

Vinod wants to acknowledge his wife for regularly reminding him to complete the chapters of the book. His wife's extra ordinary support at home enables Vinod to run that extra mile in professional life.

About the Reviewer

Jeeva S. Chelladhurai has been working as a DevOps specialist at the IBM GTS Labs for the last 9 years. He has more than 20 years of IT industry experience. He has technically managed and mentored diverse teams across the globe in envisaging and building pioneering telecommunication products. He specializes in DevOps, Automation and cloud solution delivery, with a focus on data center optimization, software-defined environments (SDEs), and distributed application development, deployment, and delivery using the newest Docker technology. Jeeva is also a strong proponent of the agile methodologies, DevOps, and IT automation. He holds a master's degree in computer science from Manonmaniam Sundaranar University and a graduation certificate in project management from Boston University, USA. Besides his official responsibilities, he writes book chapters and authors research papers. He has been instrumental in crafting reusable technical assets for IBM solution architects and consultants. He speaks in technical forums on DevOps technologies and tools. His Linked in profile can be found at `https://www.linkedin.com/in/JeevaChelladhurai`.

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://www.amazon.com/dp/1787286983`.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

It's not very often a technology comes along, which is adopted so widely across an entire industry. Since its first public release in March 2013, Docker has not only gained the support of both end users, like you and I, but also industry leaders such as Amazon, Microsoft, and Google.

Docker is currently using the following sentence on their website to describe why you would want to use it:

> *Docker provides an integrated technology suite that enables development and IT operations teams to build, ship, and run distributed applications anywhere.*

As simple as Docker's description sounds, it's been the ultimate goal for most development and IT operations teams for several years to have a tool, which can ensure that an application can consistently work across all stages of an application lifecycle, from development all the way through to production.

You will learn how to install Docker on your Operating System of choice. You will see that once Docker is installed, no matter which operating system you are using, you will get the same results when running containers.

We will then extend our Docker installation to public clouds and you will learn that no matter where you deploy your Docker hosts, the experience remains consistent and simple.

By the final chapter, you should have an idea on how Docker can be integrated into your day-to-day workflow and what your next steps with containers are going to be.

What this book covers

Chapter 1, *Installing Docker Locally*, works through installing the core Docker Engine as well as supporting tools on macOS, Windows 10, and Linux desktops so that you are ready for the forthcoming chapters.

Chapter 2, *Launching Applications Using Docker*, uses the Docker installation we installed in the previous chapter and launches containers. By the end of the chapter, we will launch a WordPress installation both manually and by using Docker Compose to define your multi-container application. We will also look at how you can publish your own images to Docker Hub.

Chapter 3, *Docker in the Cloud*, explains how to move away from your local installation of Docker and into public clouds. Here, we will look at launching Docker hosts in various public clouds and also deploy our applications onto them.

Chapter 4, *Docker Swarm*, continues to use public clouds; but rather than working with single isolated Docker hosts, we will deploy and configure a Docker Swarm cluster.

Chapter 5, *Docker Plugins*, speaks of the phrases used when describing Docker, which is *Batteries included but removable*. In this chapter, we will look at third-party plugins, which extend coreDocker functionality by adding persistent storage and multi-host networking.

Chapter 6, *Troubleshooting and Monitoring*, questions that, now that we have containers running locally, remotely, and within a cluster, what can go wrong? In this chapter, we will look at some of the problems you can come across. Also, we will learn how we can deploy tools to get metrics such as CPU, memory, and HDD utilization from your containers using both, first and third-party tools.

Chapter 7, *Putting It All Together*, emphasizes that you should now have a good understanding of what Docker is, how it works, and some possible use cases. In this chapter, we will explore how you can share container experience with colleagues as well what steps to take next.

What you need for this book

You will have to install and configure Docker17.03 (CE) on the following platforms:

- Windows 10 Professional
- macOS Sierra
- Ubuntu 16.04 LTS desktop

Also, you should have access to a public cloud platform such as Digital Ocean, Amazon Web Service, or Microsoft Azure.

Who this book is for

This book targets developers, IT professionals, and DevOps engineers who like to gain intensive, hands-on knowledge, and skills with Docker without spending hours and hours in learning. If you have been struggling to find the time to gain proficiency and confidence with Docker containers and everyday Docker tasks, you have come to the right place!

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
docker container run -d \
    --name mysql \
    -e MYSQL_ROOT_PASSWORD=wordpress \
    -e MYSQL_DATABASE=wordpress \
    mysql
```

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

```
# Install the packages we need to run wp-cli
RUN apt-get update &&\
apt-get install -y sudo less mysql-client &&\
```

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

```
curl -L "https://github.com/docker/compose/releases/download/1.10.0/
docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
chmod +x /tmp/docker-compose
sudo cp /tmp/docker-compose /usr/local/bin/docker-compose
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

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

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Installing Docker Locally

In this, the first chapter, we are going to look at installing and configuring Docker on the following platforms:

- macOS Sierra
- Windows 10 Professional
- Ubuntu 16.04 LTS Desktop

Once installed, we will then look at how you can interact with your local Docker installation.

Before we start our installation, I would like to take a moment to quickly talk about the version of Docker which we will be installing.

At the time of writing, Docker 17.03 has just been released and like most updates, introduces new features as well as changes to existing features. This book has been written with this version of Docker so some of the commands listed may not work or have the same effect when using older versions.

If you already have Docker installed, I would recommend that you check that you are running Docker 17.03 by running the following command:

```
docker version
```

If your version of Docker is older that 17.03 then please refer to the upgrade instructions in each of the following sections before proceeding with the rest of the chapters.

Docker for Mac and Windows

As we have already touched upon in the preface, the version of the Docker Engine we are going to be covering in this book is very much a Linux-based tool, so how does it work on macOS and Windows?

It is easy to assume that because macOS is an operating system built on-top of a UNIX like kernel called XNU that Docker will just run as it would do on a Linux machine, unfortunately, a lot of the features which allow Docker to run are not present in the Kernel used by macOS.

While there is the recently launched Windows Subsystem for Linux which is currently in beta, Docker for Windows does not currently take advantage of this, meaning that there is even less of a Linux-like kernel for Docker to use.

The Windows Subsystem for Linux exposes an Ubuntu shell which allows you to run native Linux command-line tools on your Windows installation; for more information, please see `https://msdn.microsoft.com/en-gb/commandline/wsl/about`.

So how does Docker for Mac and Windows work? The latest versions of macOS and Windows 10 Professional ship with hypervisors which are built into the operating systems kernel, macOS has **Hypervisor framework** while Windows 10 uses Hyper-V.

Hypervisor framework allows developers to build applications without the need to install third-party kernel extensions, meaning they can leverage full hardware virtualization but remain purely in user space meaning that virtual machines remain sandboxed as if they were running as a native application. The following URL gives a technical overview: `https://developer.apple.com/reference/hypervisor`

For Docker for Mac Docker have built their own open source framework which works with the Hypervisor framework called **HyperKit**: you can find out more about HyperKit at `https://github.com/docker/HyperKit/`.

Hyper-V has been the native hypervisor for Windows-based operating systems since Windows Server 2008; it has also been part of the desktop version of Windows since Windows 8 (Professional and Enterprise editions), it allows users and developers to launch Windows and Linux virtual machines with hardware virtualization in a sandboxed environment. For more information on Hyper-V, please see `https://www.microsoft.com/en-us/cloud-platform/server-virtualization`.

Docker for Mac and Windows uses these native virtualisation technologies to launch a virtual machine running their MobyLinux distribution, MobyLinux is a light-weight distribution based on Alpine Linux who's only function is to run Docker.

The ISO for Alpine Linux currently weighs in at 26 MB, and a fully functioning minimal installation requires a footprint of ~130MB, while the distribution is extremely small it is as useable and secure as more common Linux distributions. You can find out more at `https://alpinelinux.org/`.

Docker for Mac and Windows takes care of launching, configuring, and maintaining the virtual machine as well as functions such as networking and mounting filesystems from your local machine to a MobyLinux virtual machine.

Docker for Mac

Docker for Mac has the following system requirements; if your machine does not meet them then Docker for Mac will fail to install:

- Your Mac must be a 2010 or later model, with support for Intel's hardware support for **memory management unit** (**MMU**) virtualization.
- You must be running OS X El Capitan 10.11 or newer. I recommend that you are running the latest macOS.
- You must have at least 4GB of RAM.
- Versions of VirtualBox 4.3.30 or lower must NOT be installed as this causes problems with Docker for Mac.

To check that your machine can support Docker for Mac you can run the following command:

```
sysctl kern.hv_support
```

This should return a `1` when you run the command; this means the virtualization is enabled in your kernel as it is available on your CPU.

Downloading Docker for Mac

Docker for Mac is available from the following URL:

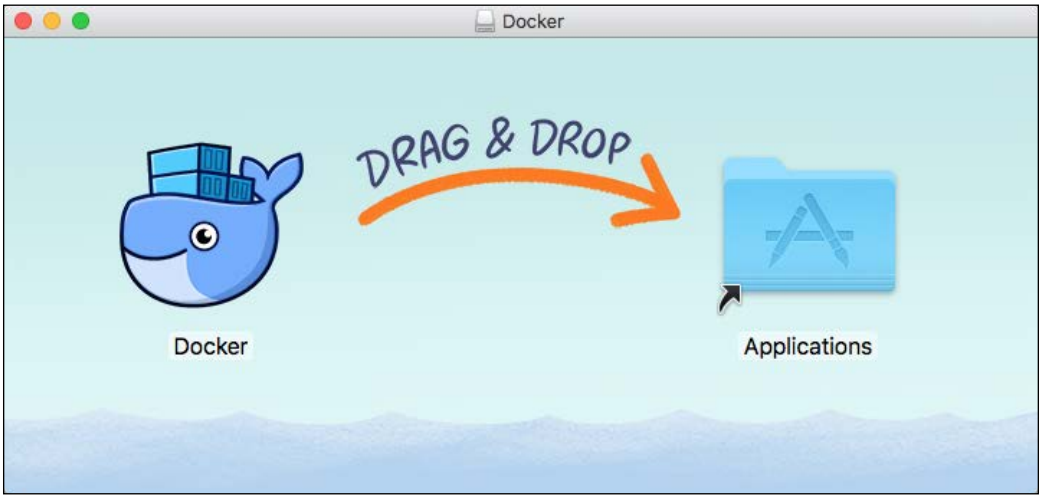
```
https://store.docker.com/editions/community/docker-ce-desktop-mac
```

I would recommend sticking with the **Stable channel** for now as this is the version we will be installing on remote machines in later chapters. Clicking on **Get Docker for Mac (stable)** will kick off a download of a disk image (DMG) file, once downloaded double-click on the file to mount it.

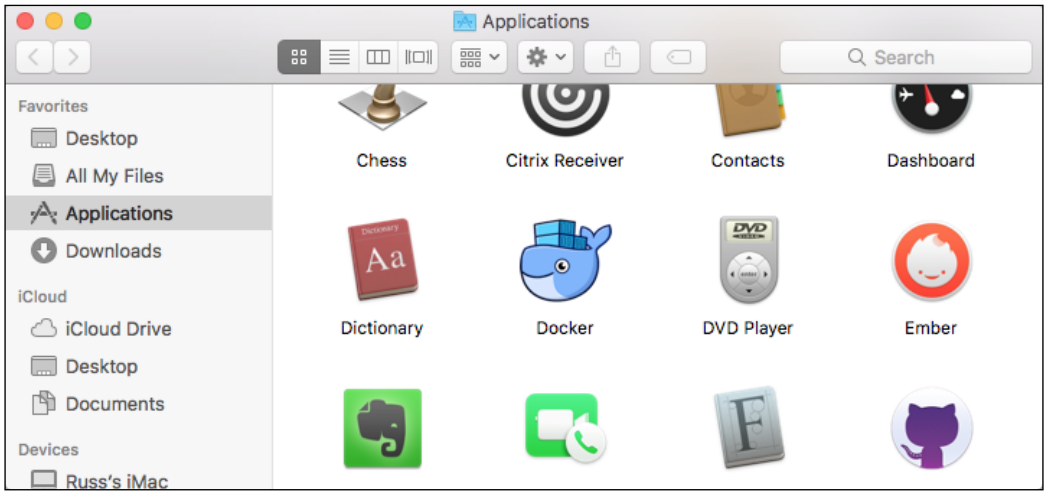
Installing Docker for Mac

Like most macOS apps, all you have to do is the following:

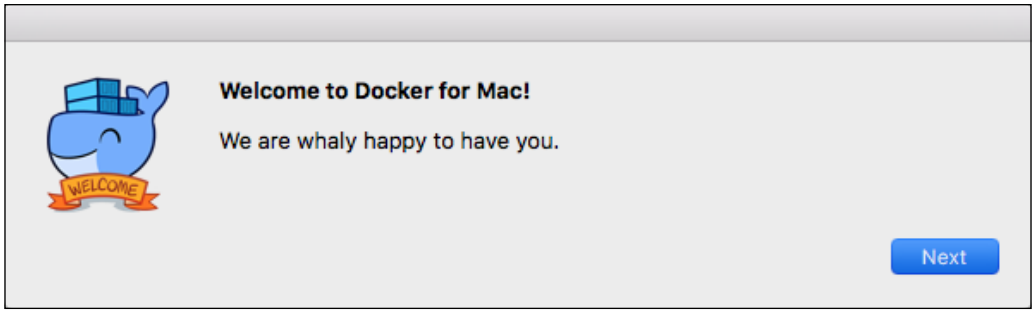
1. Drag the Docker application from the mounted disc image to your applications folder; opening the mounted image in the macOS finder by double clicking on it makes this task easy, as you can see from the following screenshot:



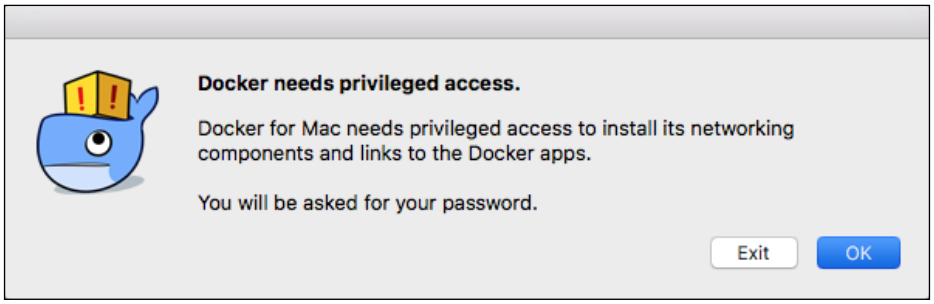
2. Once the application has been copied, you can close the finder window and open your **Applications**, find Docker, and open it:



3. When you open Docker for the first time you will be walked through the initial installation:

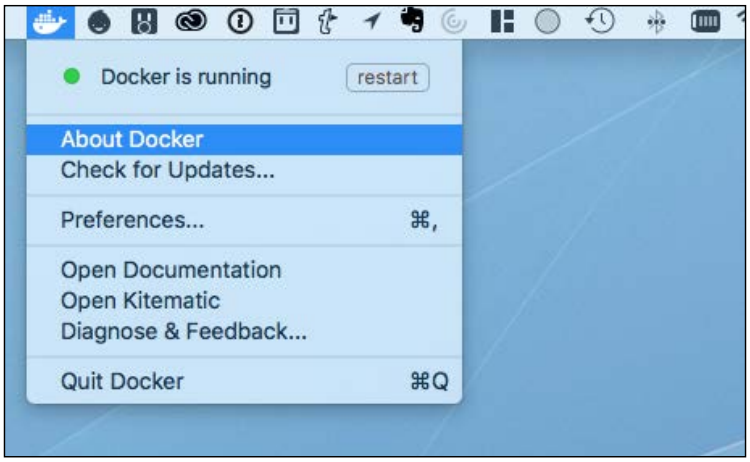


4. Clicking **Next** will tell you that Docker will ask for your password, it needs this to complete the installation.



5. After clicking **OK** and entering your password when prompted a whale icon will appear in the menu bar, and while Docker starts you should see something which looks like the following popup:

6. Clicking **Got it!** will close the pop-up. You can tell that Docker has started as the small boxes on the whales back in the icon will stop animating; also left-clicking over the icon will bring up a menu which shows the status of your Docker installation:

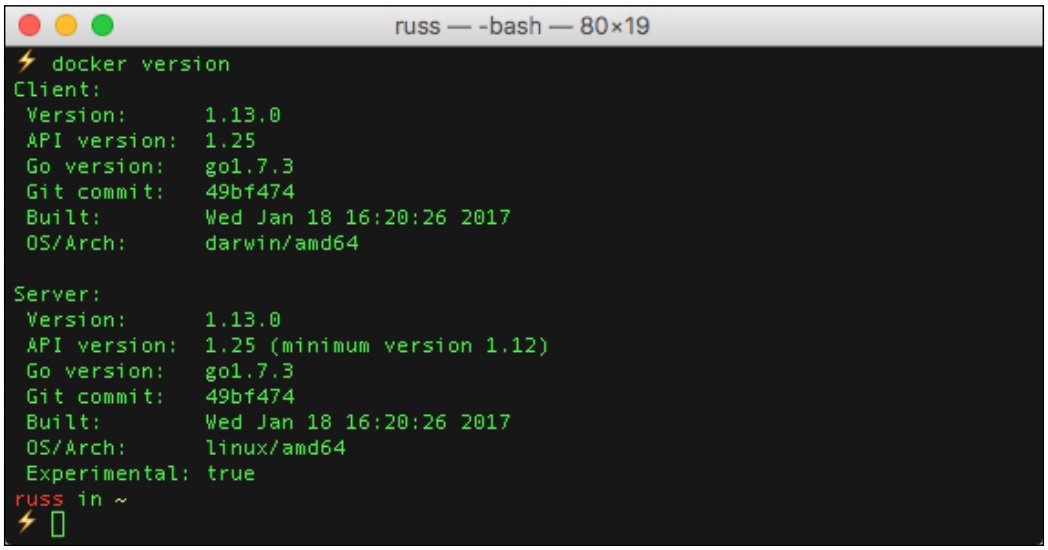


7. Selecting **About Docker** from the menu will open the following window:

8. Running the following command in a terminal shows additional information about your Docker installation:

```
docker version
```

You should see something like the following output:



As you can see, it gives details on the Docker client which is installed on your macOS host and the MobyLinux virtual machine the client is connecting to.

Docker for Windows

Docker for Windows has the following system requirements; if your machine does not meet them then Docker for Windows will inform you before exiting:

- You must be running a 64bit Windows 10 Pro, Enterprise and Education (1511 November update, build 10586 or later) installation or later (there are plans to support other versions in the future)
- Hyper-V must be enabled, though the installer will enable it for you if needed
- You must have at least 4GB of RAM

Downloading Docker for Windows

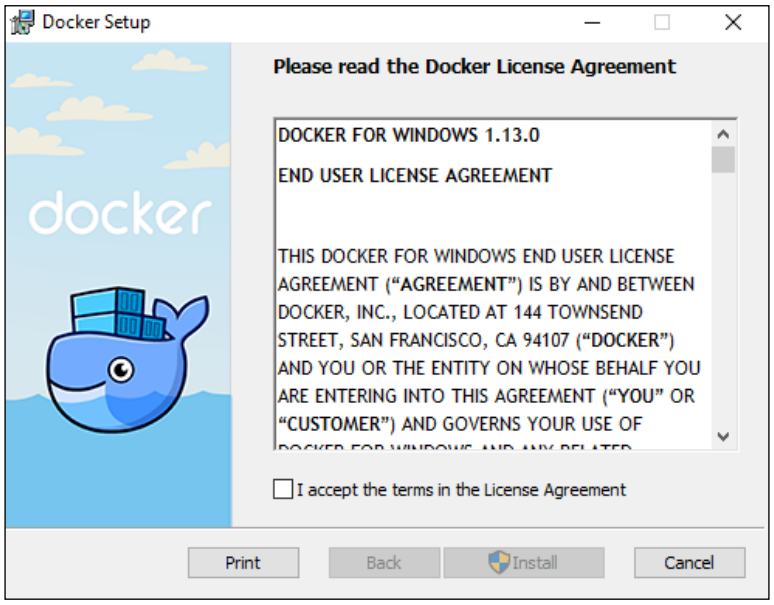
Docker for Windows is available from the following URL:

`https://store.docker.com/editions/community/docker-ce-desktop-windows`

Like Docker for Mac, I would recommend sticking with the **Stable channel**. Clicking on **Get Docker for Windows (stable)** will download an installer; once the installer has finished downloading, you will be given the option to **Run** it.

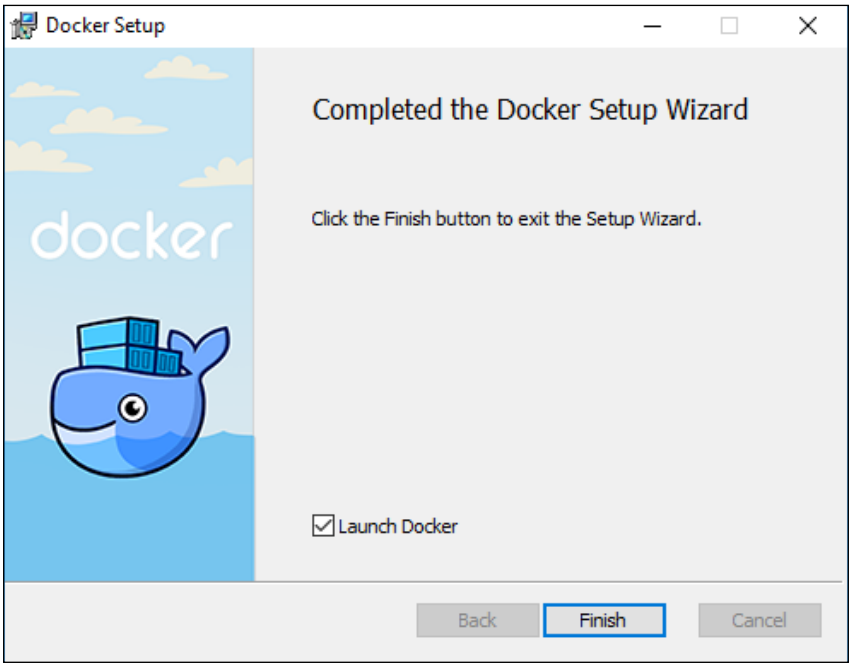
Installing Docker for Windows

When the Docker for Windows installer first opens, you will be greeted by the Docker License Agreement:

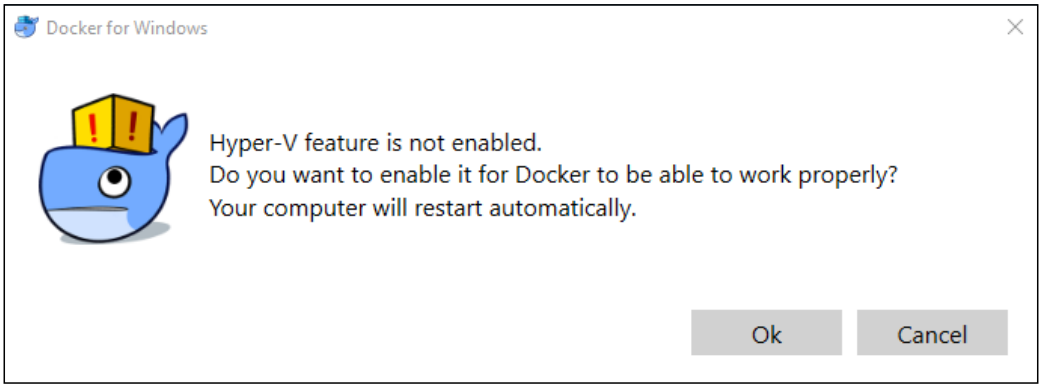


Clicking **I accept the terms of the License Agreement** will enable the **Install** button, clicking **Install** with immediately start the installation.

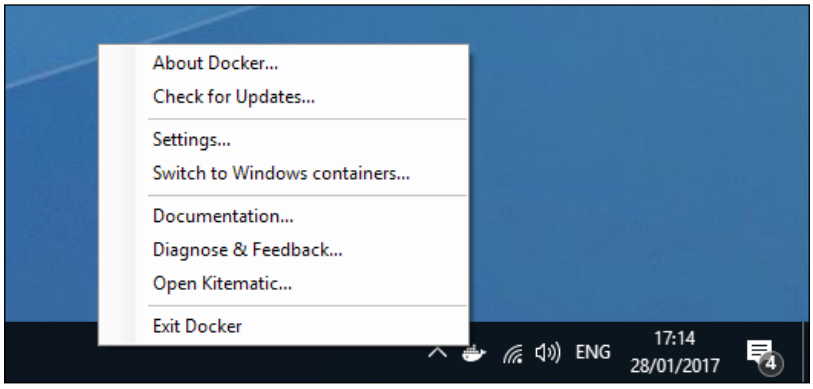
After a minute or two, you should receive confirmation that the installation has been completed.



Making sure that **Launch Docker** is ticked (it should be by default), click on **Finish** to open Docker. If you do not have Hyper-V enabled, then you will receive the following prompt:



Clicking **Ok** will reboot your computer so ensure that you have saved any open documents you may have. Once rebooted, Docker should launch automatically, and like Docker for Mac, you will notice that there is an icon of a whale in your menu bar:

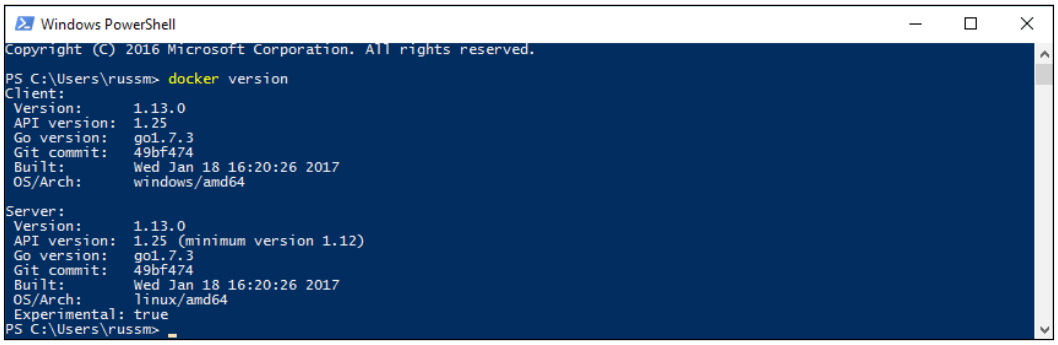


Once Docker has started, selecting **About Docker** from the menu will open the following window:

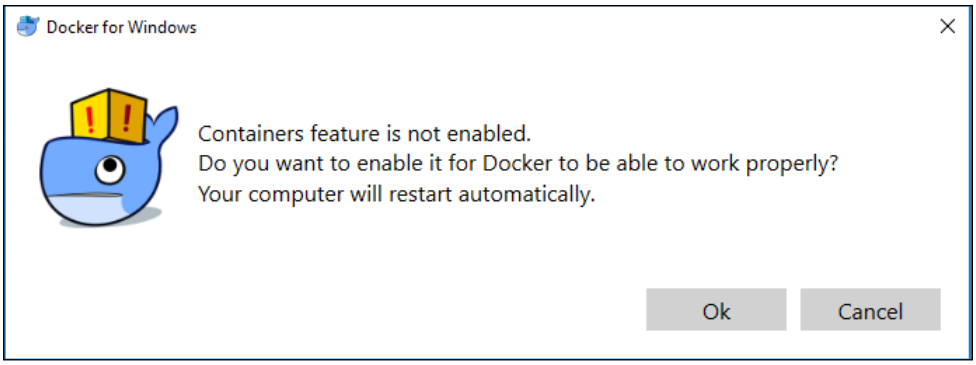
Finally, open Windows PowerShell and entering the following command:

```
docker version
```

This will return similar information on Docker for Mac, showing both the client information and details on the MobyLinux virtual machine:



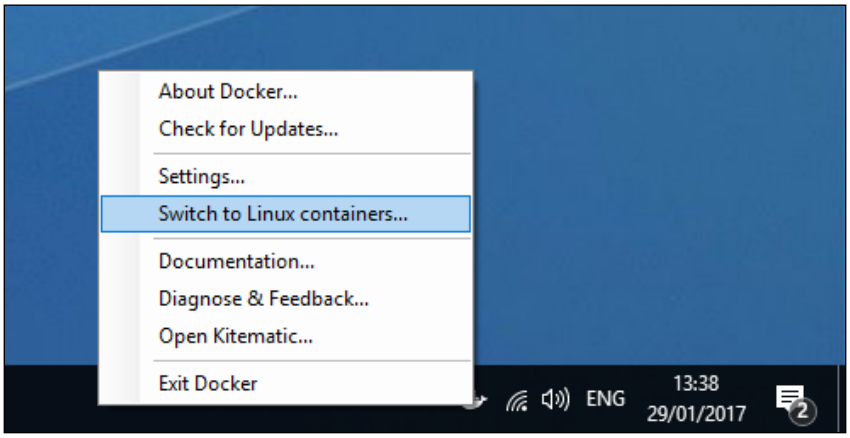
There is one more thing with Docker for Windows: you can run native Windows containers. You can enable this feature by selecting the **Switch to Windows containers ...** option from the menu; if it is your first time enabling this feature then you will get the following dialog popup:



Clicking **Ok** will reboot your machine. Once rebooted, selecting the menu option again will switch you over from using Linux to Windows; this is reflected when you run the following:

```
docker version
```

As you can see, the server **OS/Arch** has changed from **linux/amd64** to **windows/amd64**. We will not be looking at Windows containers in this book; you can change back to Linux containers by using the menu option:



If you have any problems running the commands in later chapters on Docker for Windows, check that you are using Linux containers by running `docker version` or using the menu.

Upgrading Docker for Mac and Windows

Both Docker for Mac and Windows allow you to easily update your installed version of Docker. If you have an old version of Docker for Mac or Windows installed, you should have been prompted that there is a later version of Docker available when you first open Docker. I you haven't had a prompt then selecting **Check for Updates…** from the menu will kick off the upgrade process, which is similar to the installation process we have already covered for each of the versions.

If there are no updates, then you will receive a notification confirming you are on the latest version.

Docker on Ubuntu 16.04

If you have been looking at the Docker website, you will notice that there is not a Docker for Linux Desktop download, that is because there is no need for one. Docker is a Linux tool and will run natively on most Linux desktops and servers.

While Docker is available in the main Ubuntu repositories, I would recommend installing Docker using the official repository. You can do this by running the following command:

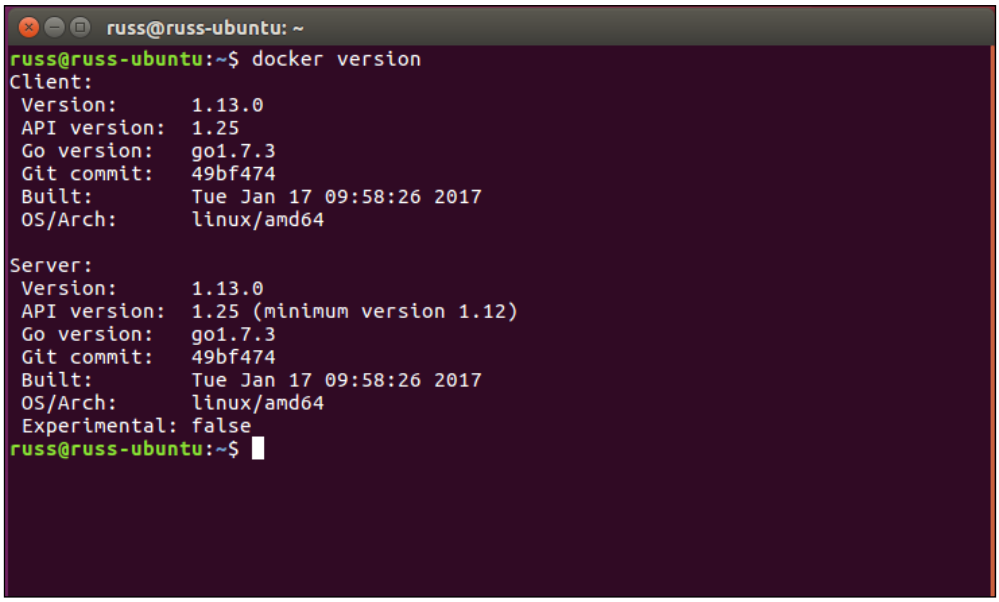
```
curl -sSL https://get.docker.com/ | sh
```

This will configure and install the latest version of Docker Engine. Once installed, you will receive a command to run to give your user permission to run Docker, run the command and then log out.

When you log back in, you will be able to run the following command:

```
docker version
```

You should see something like the following:



One thing I haven't mentioned so far is that when we installed Docker for Mac and Windows two additional components were installed. These were Docker Machine and Docker Compose, we will be covering these in *Chapter 2, Launching Applications Using Docker* and *Chapter 3, Docker in the Cloud*.

To install Docker Machine run the following commands:

```
curl -L "https://github.com/docker/machine/releases/download/v0.9.0/
docker-machine-$(uname -s)-$(uname -m)" -o /tmp/docker-machine
chmod +x /tmp/docker-machine
sudo cp /tmp/docker-machine /usr/local/bin/docker-machine
```

To install Docker Compose, run the following commands:

```
curl -L "https://github.com/docker/compose/releases/download/1.10.0/
docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
chmod +x /tmp/docker-compose
sudo cp /tmp/docker-compose /usr/local/bin/docker-compose
```

Once installed you should be able to run the following two commands:

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

Testing your installation

Now that we have Docker installed, we are going to quickly test our installation by downloading, running and connecting to a NGINX container.

NGINX is a free, open source, high-performance HTTP server and reverse proxy. NGINX is known for its high performance, stability, rich feature set, simple configuration, and low resource consumption.

A note on Docker commands

Docker 1.13 introduced a slightly altered set of command line instructions for interacting with containers and images. As this syntax will eventually become the new standard we will be using it throughout this book. For more information on the CLI restructure, please see the Docker 1.13 announcement blog post at `https://blog.docker.com/2017/01/whats-new-in-docker-1-13/`

To download and launch the container all you need to do run the following commands from your terminal prompt;

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

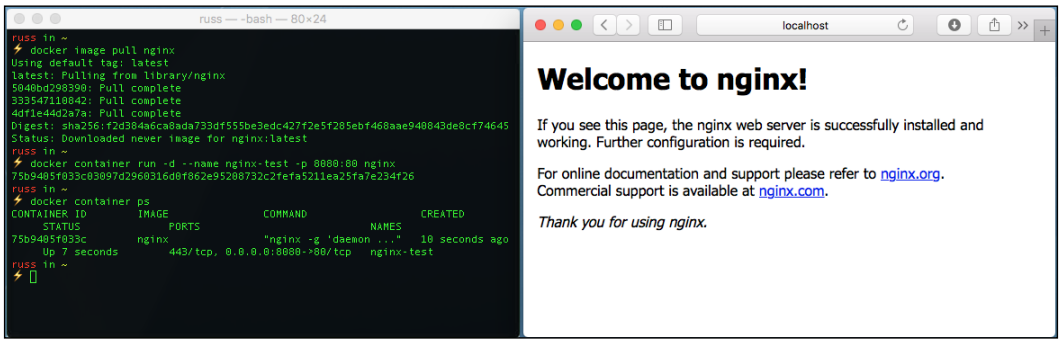
The first command pulls the NGINX container image from the Docker Hub, and the second command launches our NGINX container, naming it `nginx-test` mapping port `8080` on your machine to port `80` on the container.

You can check that the container is running using the following command:

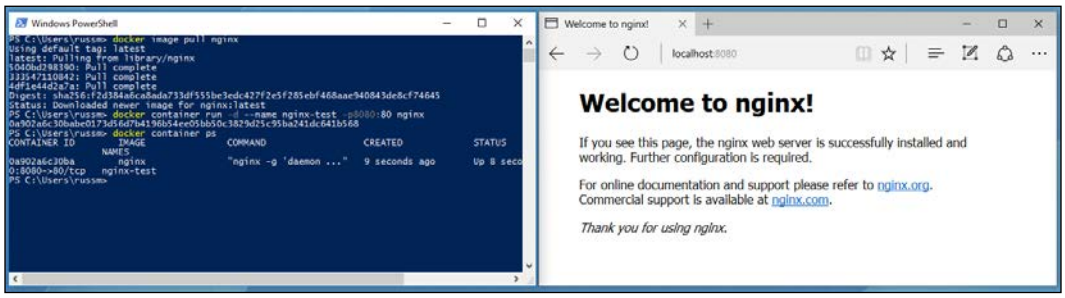
```
docker container ps
```

Opening your browser and going to `http://localhost:8080/` should show you the default **Welcome to NGINX** page.

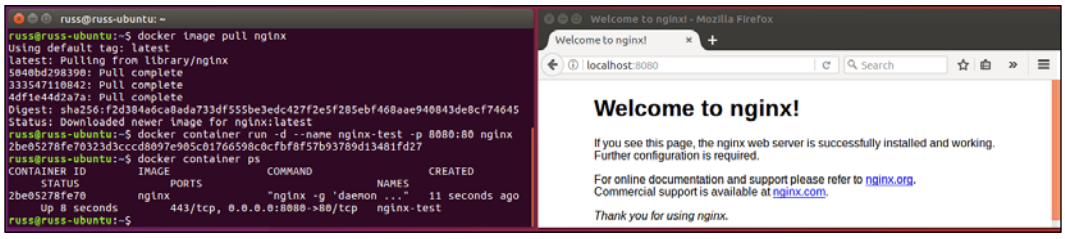
As you can see from the following screens, the process is the same when using Docker for Mac:



Docker for Windows:



Or Docker on Ubuntu 16.04:



As you can see from the screens above, the result of us running the same command on each of the three platforms is exactly the same.

Once you have tested launching a container you can tidy up afterwards by running the following commands to stop and remove the container and then delete the image:

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

Summary

In this chapter, we have worked through installing Docker for Mac, Docker for Windows and Docker on Ubuntu 16.04. Hopefully, you will have followed along with one or more of the installations on your local machine. We have also launched our first container and connected to it using our web browser.

In the next chapter, we will go into a lot more detail on the commands we used to launch our test container as well as using Docker Compose to launch multi-container applications.

2
Launching Applications Using Docker

In this chapter, we are going to be looking at launching more than just a simple web server using our local Docker installation. We will look at the following topics:

- Using Docker on the command-line to launch applications
- How to use the Docker `build` command
- Using Docker Compose to make multi-container applications easier to launch

We will then look at using all the techniques above to launch a WordPress and Drupal application stack.

Docker terminology

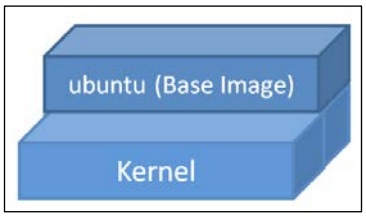
Before we start learning how to launch containers, we should quickly discuss some of the more common terminology we are going to be using in this chapter.

Please note, the Docker commands in this chapter have been written for use with Docker 1.13 and later. Trying to run commands such as `docker image pull nginx` in older versions will fail with an error. Please refer to *Chapter 1, Installing Docker Locally* for details on how to install the latest version of Docker.

Docker images

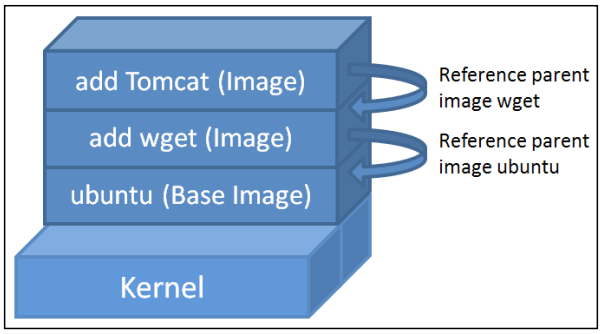
A **Docker image** is a collection of all the files that make up an executable software application. This collection includes the application plus all the libraries, binaries, and other dependencies such as deployment descriptors and so on. just needed just to run the application everywhere without any hitch or hurdle. These files in the Docker image are read-only and hence the content of the image cannot be altered. If you choose to alter the content of your image, the only option Docker allows is to add another layer with the new changes. In other words, a Docker image is made up of layers which you can review using `docker image history` subcommand.

The Docker image architecture effectively leverages this layering concept to seamlessly add additional capabilities to the existing images to meet the varying business requirements and increase the reuse of images. In other words, capabilities can be added to existing images by adding additional layers on top of that image and deriving a new image. The Docker images have a parent/child relationship and the bottom-most image is called the **base image**. The base image is the special image that doesn't have any parent:



In the previous diagram, Ubuntu is a base image and it does not have any parent image.

The Ubuntu Docker image is a minimalist bundle of software libraries and binaries that are critical to run an application. It does not include Linux Kernel, Diver Drivers, and various other services a full-fledged Ubuntu operating system would provide.



As you can see in the above diagram, everything starts with a base image and here in this example, it is Ubuntu. Further on, the `wget` capability is added to the image as a layer and the `wget` image is referencing Ubuntu image as its parent. And in the next layer, an instance of Tomcat application server is added and it refers the `wget` image as its parent. Each addition that is made to the original base image is stored in a separate layer (a kind of hierarchy gets generated here to retain the original identity).

Precisely speaking, any Docker image has to originate from a base image and an image gets continuously enriched in its functionality by getting fresh modules and this is accomplished by adding an additional module as a new layer on the existing Docker image one by one as vividly illustrated in the above diagram.

The Docker platform provides a simple way of building new images or extending existing images. You can also download the Docker images that the other people have already created and deposited in Docker image repositories (private or public).

Docker Registry

A **Docker Registry** is a place where Docker images can be stored in order to be publicly or privately found, accessed, and used by worldwide software developers for quickly crafting fresh and composite applications without any risks. Because, all the stored images will have gone through multiple validations, verifications, and refinements, the quality of those images are really high.

Using the `dockerimage push` subcommand, you can dispatch your Docker image to the registry so that it is registered and deposited. Using the `dockerimage pull` subcommand, you can download a Docker image from the registry.

A Docker Registry could be hosted by a third party as a public or private registry, such as one of the following registries:

- Docker Hub (`https://hub.docker.com/`)
- Quay (`https://quay.io/`)
- Google Container Registry (`https://cloud.google.com/container-registry/`)
- AWS Container Registry (`https://aws.amazon.com/ecr/`)

Every institution, innovator and individual can have their own Docker Registry to stock up their images for internal and/or external access and usage.

Docker Hub

In the previous chapter, when you ran the `dockerimage pull` subcommand, the `nginx` image got downloaded mysteriously. In this section, let's unravel the mystery around the `docker image pull` subcommand and how the Docker Hub immensely contributed toward this unintended success.

The good folks in the Docker community have built a repository of images and they have made it publicly available at a default location, `index.docker.io`. This default location is called the Docker Hub. The `docker image pull` subcommand is programmed to look for the images at this location. Thus, when you `pull` a `nginx` image, it is effortlessly downloaded from the default registry. This mechanism helps in speeding up the spinning of the Docker containers.

The Docker Hub is the official repository that contains all the painstakingly curated images that are created and deposited by the worldwide Docker development community. This so-called cure is enacted for ensuring that all the images stored in the Docker Hub are secure and safe through a host of quarantine tasks. There are additional mechanisms such as creating the image digest and having content trust that gives you the ability to verify both the integrity and the publisher of all the data received from a registry over any channel.

There are proven verification and validation methods for cleaning up any knowingly or unknowingly introduced malware, adware, viruses, and so on, from these Docker images.

The digital signature is a prominent mechanism of the utmost integrity of the Docker images. Nonetheless, if the official image has been either corrupted, or tampered with, then the Docker engine will issue a warning and then continue to run the image.

In addition to the official repository, the Docker Hub Registry also provides a platform for thethird-party developers and providers for sharing their images for general consumption. The third-party images are prefixed by the user ID of their developers or depositors.

For example, `russmckendrick/cluster`is a third-party image, wherein `russmckendrick` is the user ID and `cluster` is the image repository name. You can download any third-party image by using the `docker image pull` subcommand, as shown here:

```
docker image pull russmckendrick/cluster
```

Apart from the preceding repository, the Docker ecosystem also provides a mechanism for leveraging the images from any third-party repository hub other than the Docker Hub Registry, and it also provides the images hosted by the local repository hubs. As mentioned earlier, the Docker engine has been programmed to look for images at `index.docker.io` by default, whereas in the case of the third-party or the local repository hub, we must manually specify the path from where the image should be pulled.

A manual repository path is similar to a URL without a protocol specifier, such as `https://`, `http://`and `ftp://`.

Following is an example of pulling an image from a third-party repository hub:

```
docker image pull registry.domain.com/myapp
```

Controlling Docker containers

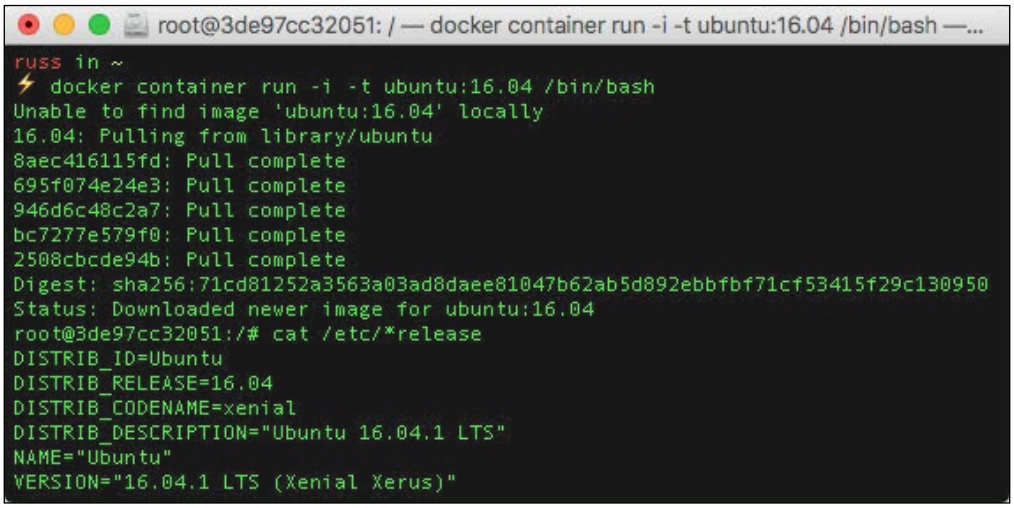
The Docker engine enables you to `start`, `stop`, and `restart` a container with a set of `docker` subcommands. Let's begin with the `docker container stop` subcommand, which stops a running container. When a user issues this command, the Docker engine sends `SIGTERM (-15)` to the main process, which is running inside the container. The **SIGTERM** signal requests the process to terminate itself gracefully.

Most of the processes would handle this signal and facilitate a graceful exit. However, if this process fails to do so, then the Docker engine will wait for a grace period. Even after the grace period, if the process has not been terminated, then the Docker engine will forcefully terminate the process. The forceful termination is achieved by sending `SIGKILL (-9)`.

The **SIGKILL** signal cannot be caught or ignored and hence, it will result in an abrupt termination of the process without a proper cleanup.

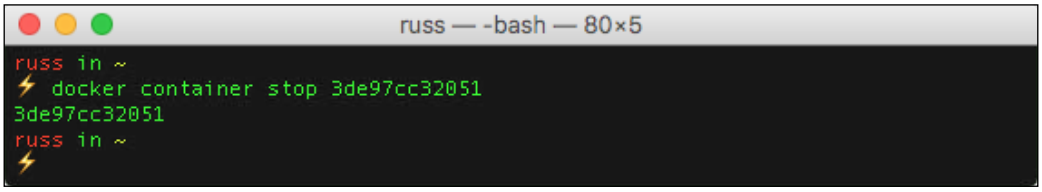
Now, let's launch our container and experiment with the `docker container stop` subcommand, as shown here:

```
dockercontainer run -i -t ubuntu:16.04 /bin/bash
```

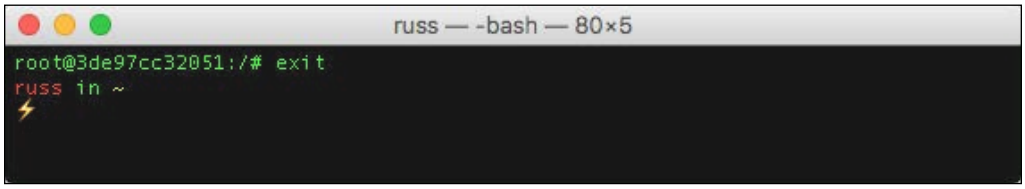


Having launched the container, let's run the `docker container stop` subcommand on this container by using the container `ID` that was taken from the prompt. Of course, we have to use a second screen/terminal to run this command, and the command will always echo back to the container `ID`, as shown here:

```
docker container stop 3de97cc32051
```



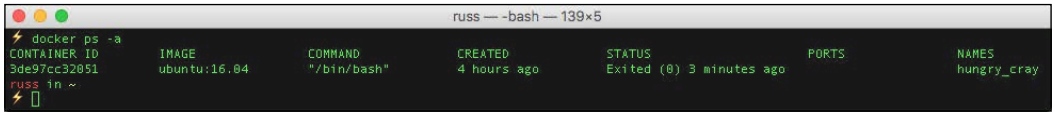
Now, if you switch to the screen/terminal where you were running the container, you will notice that the container is being terminated. If you observe a little more keenly, then you will also notice the text `exit` next to the container prompt. This happened due to the SIGTERM handling mechanism of the bash shell, as shown here:



If we take it one step further and run the `docker container ps` subcommand, then we will not find this container anywhere in the list. The fact is that the `docker container ps` subcommand, by default, always lists the container that is in the running state. Since our container is in the stopped state, it was comfortably left out of the list. Now, you might ask, how do we see the container that is in the stopped state? Well, the `docker container ps` subcommand takes an additional argument `-a`, which will list all the containers in that Docker host irrespective of its status.

This can be done by running the following command:

```
docker container ps -a
```



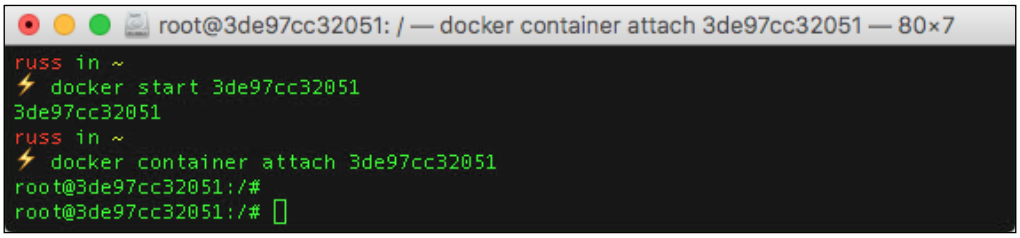
Next, let's look at the `docker container start` subcommand, which is used for starting one or more stopped containers. A container could be moved to the stopped state either by the `docker container stop` subcommand or by terminating the main process in the container either normally or abnormally. On a running container, this subcommand has no effect.

Let's start the previously stopped container by using the `docker container start` subcommand, as follows:

```
docker start 3de97cc32051
```

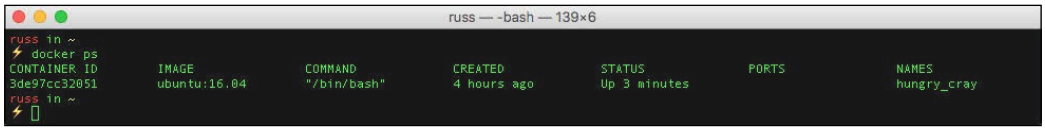
By default, the `docker container start` subcommand will not attach to the container. You can attach it to the container either by using the `-a` option in the `docker container start` subcommand or by explicitly using the `docker container attach` subcommand, as shown here:

```
docker container attach 3de97cc32051
```



Now, let's run the `docker containerps` command and verify the container's running status, as shown here:

```
docker container ps
```



The `restart` command is a combination of the `stop` and the `start` functionality. In other words, the `restart` command will `stop` a running container by following the precise steps followed by the `docker conatiner stop` subcommand and then it will initiate the `start` process. This functionality will be executed by default through the `docker conatiner restart` subcommand.

The next important set of container controlling subcommands are the following:

- `docker container pause`
- `docker container unpause`

The `docker container pause` subcommands will essentially freeze the execution of all the processes within that container. Conversely, the `docker container unpause` subcommand will unfreeze the execution of all the processes within that container and resume the execution from the point where it was frozen.

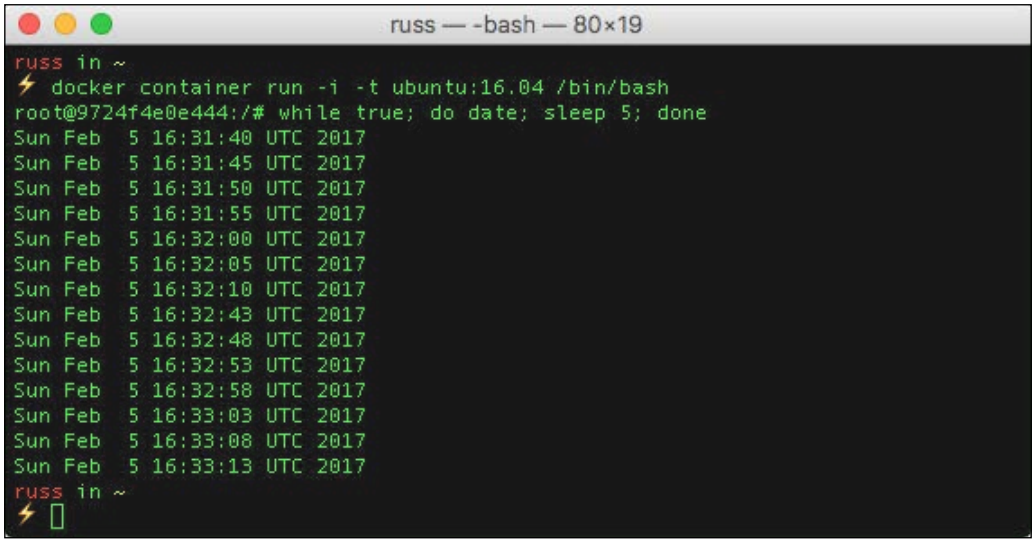
Having seen the technical explanation of `pause`/`unpause`, let's see a detailed example for illustrating how this feature works. We have used two screen/terminal scenarios. On one terminal, we have launched our container and used an infinite while loop for displaying the date and time, sleeping for 5 seconds, and then continuing the loop. We will run the following commands:

```
docker container run -i -t ubuntu:16.04 /bin/bash
```

Once you are within the container, run the following:

```
while true; do date; sleep 5; done
```

Our little script has very faithfully printed the date and time every `5` seconds apart from when it was paused:



As you can see from the terminal output above, we encountered a delay of around 30 seconds, because this is when we initiated the `docker container pause` subcommand on our container on the second terminal screen, as shown here:

```
docker container pause 9724f4e0e444
```

When we paused our container, we looked at the process status by using the `docker containerps` subcommand on our container, which was on the same screen, and it clearly indicated that the container had been paused, as shown in this command result:

```
docker container ps
```

We continued onto issuing the `docker conatiner unpause` subcommand, which unfroze our container, continued its execution, and then started printing the date and time, as we saw in the preceding command, shown here:

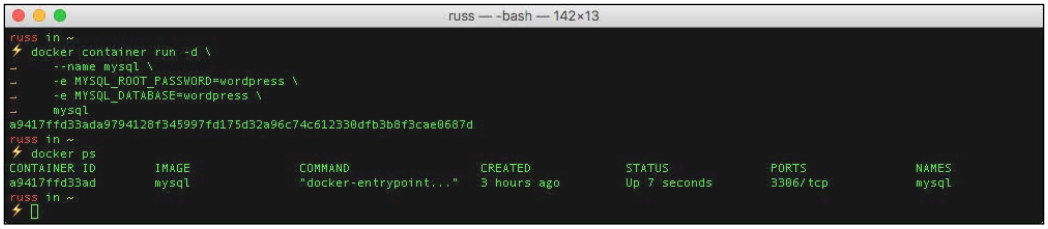
```
docker container unpause 9724f4e0e444
```

We explained the `pause` and the `unpause` commands at the beginning of this section.

Lastly, the container and the script running within it had been stopped by using the `docker container stop` subcommand, as shown here:

```
docker container stop 9724f4e0e444
```

You can see everything we ran in our second terminal below:



Let's look at doing something a little more complex now.

Running a WordPress container

Almost everyone at some point will have installed, used, or read about WordPress, so for our next example, we will be using the official WordPress container from the Docker Hub. You can find details on the container at `https://hub.docker.com/_/wordpress/`.

WordPress is web software that you can use to create a beautiful website, blog, or app. We like to say that WordPress is both free and priceless at the same time. For more information, check out `https://wordpress.org/`.

To launch WordPress, you will need to download and run two containers, the first of which is the database container, for this I recommend using the official MySQL container which you can find at `https://hub.docker.com/_/mysql/`.

To download the latest MySQL container image run the following command on your Mac, Windows or Linux machine:

```
docker image pull mysql
```

Now that you have the pulled a copy of the image you can launch MySQL by running the following command:

```
docker container run -d \
    --name mysql \
    -e MYSQL_ROOT_PASSWORD=wordpress \
    -e MYSQL_DATABASE=wordpress \
    mysql
```

The command above launches (`docker container run`) the MySQL in a detached state (using `-d`), meaning that it is running in the background, we are calling the container `mysql` (`--namewordpress`) and we are using two different environment variables (using `-e`) to set the MySQL root password to `wordpress (-e MYSQL_ROOT_PASSWORD=wordpress`) and to create a database called `wordpress` (`-e MYSQL_DATABASE=wordpress`).

Once launched, you should receive the container ID. You can check the container is running as expected by using the following command:

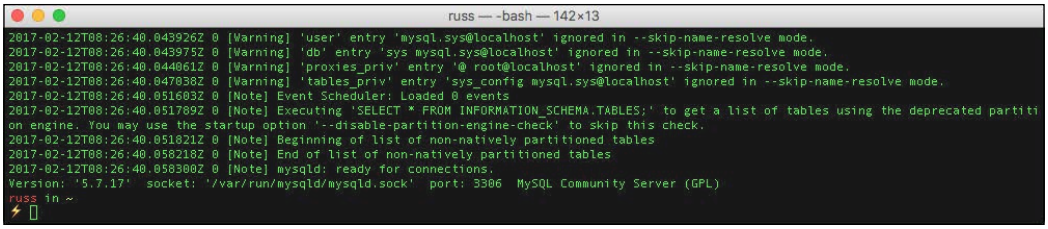
```
docker container ps
```

Now, at this point, although the container is running that doesn't really mean that MySQL is ready. If you were to launch your WordPress container now, you might find that it runs for a short while and then stops.

Don't worry, this is expected. As there is no MySQL data within the container it takes a little while to get itself into a state where it is available to accept incoming connections.

To check the status of your MySQL container you can run the following command:

```
docker container logs mysql
```



Once you see the message **mysqld: ready for connections**, you are good to launch your WordPress container; you may find yourself having to check the logs a few times.

Next up, we to down the WordPress container image; to do this, run the following command:

```
docker image pull wordpress
```

Once downloaded run the following command to launch the WordPress container:

```
docker container run -d \
    --name wordpress \
--link mysql:mysql\
    -p 8080:80 \
    -e WORDPRESS_DB_PASSWORD=wordpress \
    wordpress
```

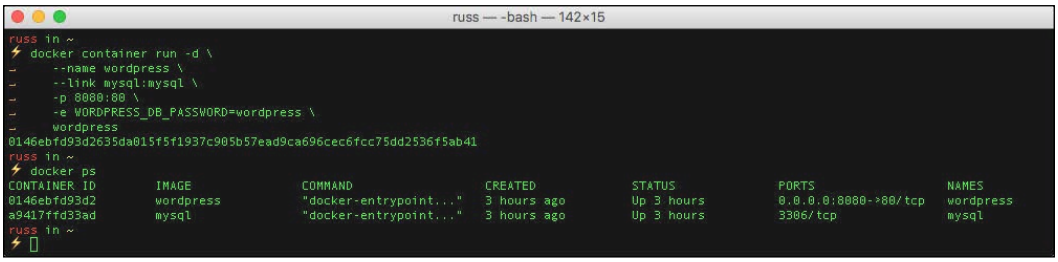
Again, we are launching the container in the background (using `-d`), calling the container the `wordpress` (with `--name wordpress`). This is where things differ slightly between the MySQL and WordPress containers, we need to let the WordPress container know where our MySQL container is accessible, to do this are using the link flag (in our case by running `--link mysql:mysql`) this will create an alias within the WordPress container pointing it at the IP address of the MySQL container.

Next up we are opening port `8080` on our machine and mapping it to port `80` on the container (using `-p 8080:80`) and then letting WordPress know what the password is (with `-e WORDPRESS_DB_PASSWORD=wordpress`).

Check the running containers using the following command:

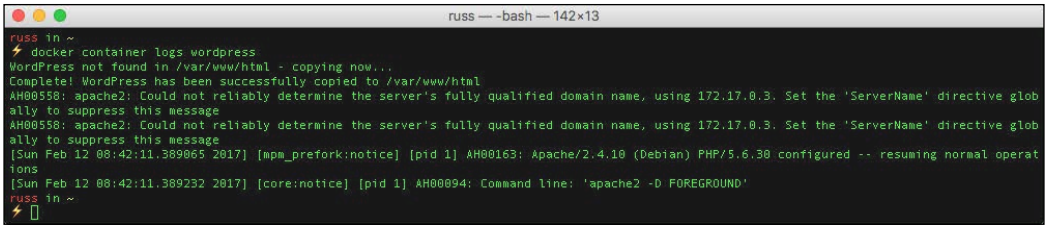
```
docker container ps
```

Should show you that you now have two running containers, MySQL and WordPress:

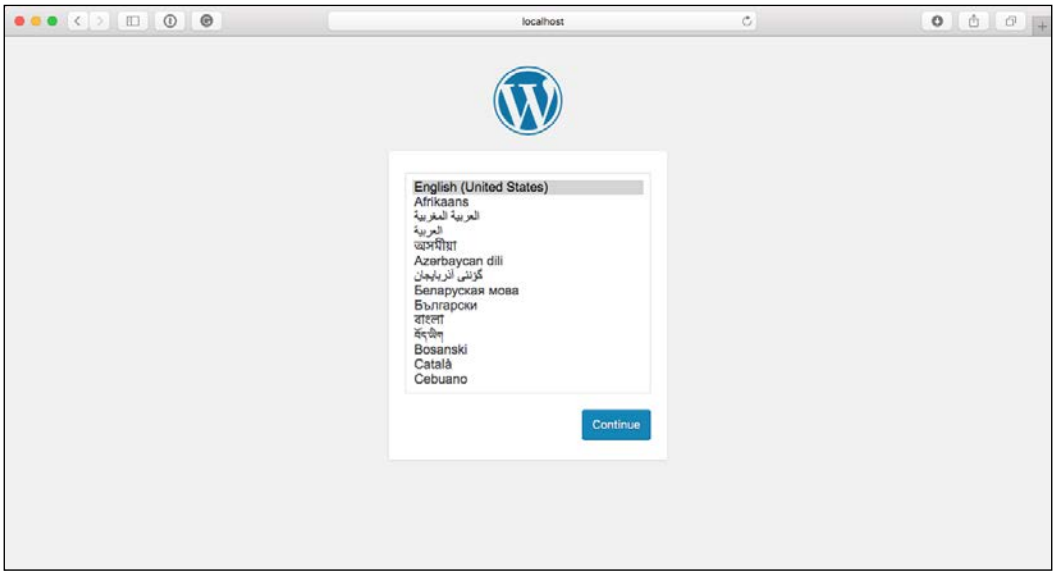


Unlike the MySQL container, there isn't much the WordPress container needs to do before it is accessible, you can check the logs by running the following command:

```
docker container logs wordpress
```



If you open your browser and go to `http://localhost:8080/` you should see your WordPress installation sitting at an installation prompt like the following:



If you like, you can work through the installation and get WordPress up and running by clicking on **Continue** and following the onscreen prompts; however, the next set of commands we will be running will destroy our two containers.

To remove everything we have just launched ahead of the next exercise, run the following commands:

```
docker container stop wordpressmysql
docker container rmwordpressmysql
docker image rm wordpress mysql
```

So far we have used the Docker client to easily launch, stop, start, pause, unpause and remove containers as well as downloading and removing container images from the Docker Hub, while this is great to quickly launch a few containers it can get complicated to manage once you have more than a few containers running at once, this is where the next tool we are going to look at comes in.

Docker Compose

If you were following along with the Linux installation in *Chapter 1*, *Installing Docker Locally* then you should have already installed Docker Compose manually, for those of you that skipped that part then you will glad to know that Docker Compose is installed and maintained as part of Docker for Mac and Windows.

I am sure that you will agree that so far Docker has proved to be quite intuitive, Docker Compose is no different. It started off life as third-party software called Fig and was written by Orchard Labs (the project's original website is still available at `http://fig.sh/`).

The original project's goal was the following:

> "*Provide fast, isolated development environments using Docker*"

Since Orchard Labs became part of Docker, they haven't strayed too far from the original projects goal:

> "*Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a Compose file to configure your application's services. Then, using a single command, you create and start all the services from your configuration.*"

Before we start looking at Compose files and start containers up, let's think of why a tool such as Compose is useful.

Why Compose?

Launching individual containers is as simple as running the following command:

```
docker container run -i -t ubuntu:16.04 /bin/bash
```

This will launch and then attach to an Ubuntu container. As we have already touched upon, there is a little more to it than just launching simple containers though. Docker is not here to replace virtual machines, it is here to run a single application.

This means that you shouldn't really run an entire LAMP stack in single container, instead, you should look at running Apache and PHP in one container, which is then linked with a second container running MySQL.

You could take this further, running NGINX container, a PHP-FPM container, and a MySQL container. This is where it gets complicated. All of sudden, your simple single command for launching a container is now several lines, all of which must executed in the correct order with the correct flags to expose ports, link them together and configure the services using environment variables.

This is exactly the problem Docker Compose tries to fix. Rather than several long commands, you can define your containers using a YAML file. This means that you will be able to launch your application with a single command and leave the logic of the order in which the containers will be launched to Compose.

YAML Ain't Markup Language (**YAML**) is a human-friendly data serialization standard for all programming languages.

It also means that you can ship your application's Compose file with your code base or directly to another developer/administrator and they will be able to launch your application exactly how you intended it be executed.

Compose files

Let's start by getting a launching WordPress again. First of all, if you haven't already clone the GitHub repository which accompanies this book. You can find it at the following URL: `https://github.com/russmckendrick/bootcamp`

For more information on how to clone the repository please see the introduction. Once you have repo cloned run the following commands from the top level of the repo:

```
cd chapter2/compose-wordpress
```

The `compose-wordpress` folder contains the following `docker-compose.yml` file:

```
version: "3"

services:
mysql:
```

```
      image: mysql
      volumes:
        - db_data:/var/lib/mysql
      restart: always
      environment:
        MYSQL_ROOT_PASSWORD: wordpress
        MYSQL_DATABASE: wordpress
wordpress:
depends_on:
        - mysql
      image: wordpress
      ports:
        - "8080:80"
      restart: always
      environment:
        WORDPRESS_DB_PASSWORD: wordpress

volumes:
db_data:
```

As you can see, the `docker-compose.yml` file is easy to follow; our initial `docker-compose.yml` file is split into three sections:

- **Version**: This tells Docker Compose which file format we are using; the current version is 3
- **Services**: These are where our containers are defined, you can define several containers here
- **Volumes**: Any volumes for persistent storage are defined here, we will go into this in more detail in later chapters

For the most part, the syntax is pretty similar to that we used to launch our WordPress containers using the Docker command-line client. There are, however a few changes:

- `volumes`: In the `mysql` container we are taking a volume called `db_data` and mounting it to `/var/lib/mysql` within the container
- `restart`: This is set to `always`, meaning that if our containers stop responding any reason, like the `wordpress` container will do until the `mysql` container is accepting connections, then it will be restarted automatically meaning we don't have to manually intervene
- `depends_on`: Here we are telling the `wordpress` container not to start until the `mysql` container is running

You may notice that we are not linking our containers, this is because Docker Compose automatically creates a network to launch the services in, each container within the network created by Docker Compose automatically has its host file updated to include aliases for each of the containers within the service, meaning that our WordPress container will be able to connect to our MySQL container using the default host of `mysql`.

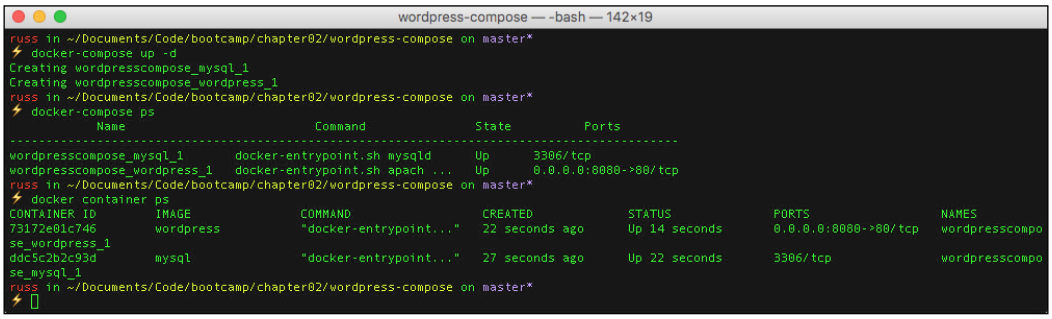
To launch our WordPress installation, all we need to do is run the following commands:

```
docker-compose pull
docker-compose up -d
```

Using the `-d` flag at the end of the command launches the containers in detached mode, this means that they will run in the background.

If we didn't use the `-d` flag, then our containers would have launched in the foreground and we would not have been able to carry on using the same terminal session without stopping the running containers.

You will see something like the following output:



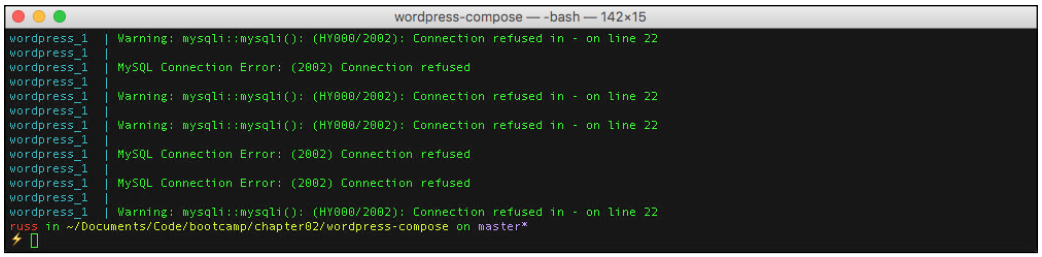
While the containers are up and running, which you can see by running the follow:

```
docker-compose ps
docker container ps
```

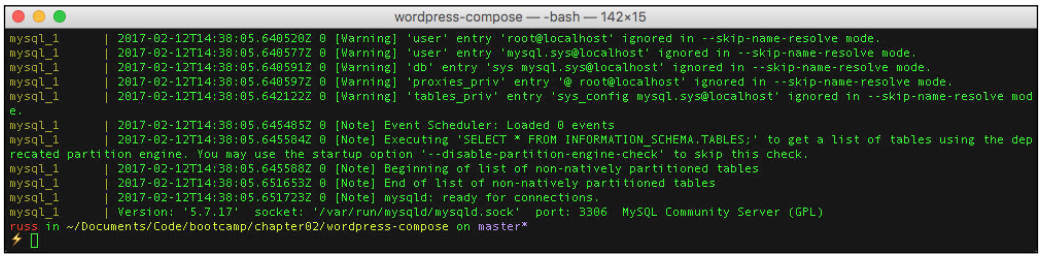
It will take a short while for the MySQL container to be ready to accept connections, you may find running:

```
docker-compose logs
```

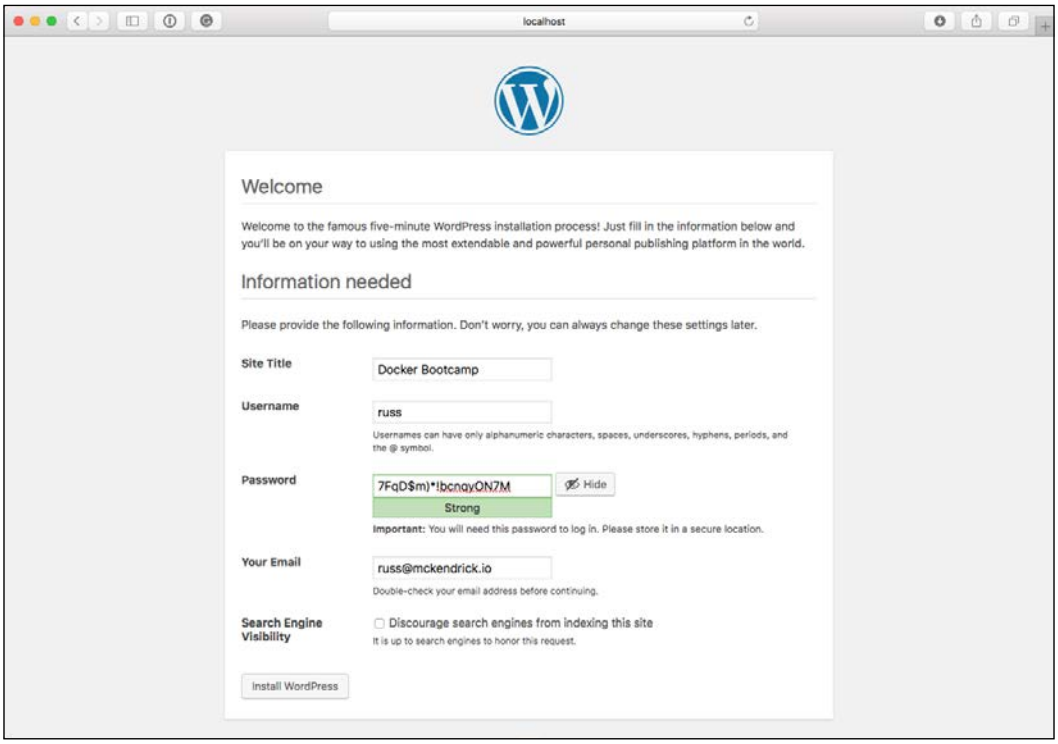
Show you connection errors like the ones below:



Don't worry, you should soon see something like the following:



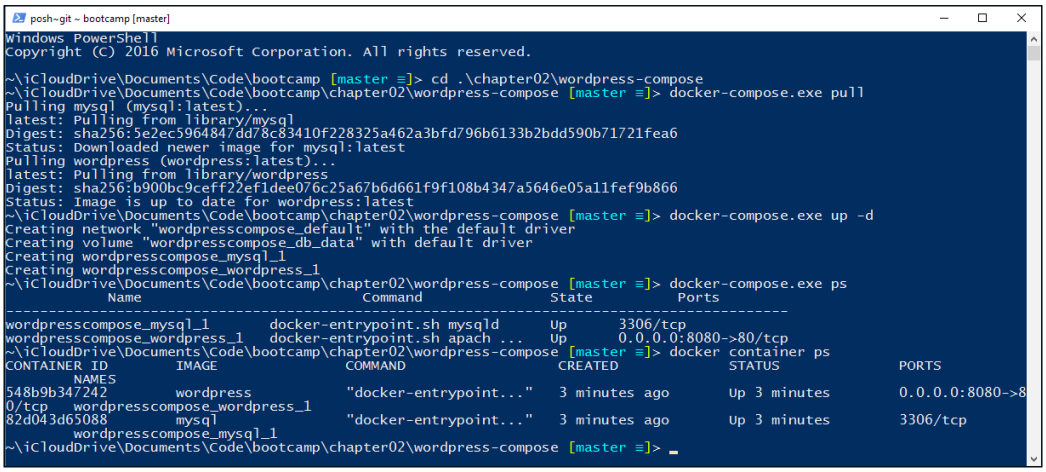
Again, opening `http://localhost:8080/` in your browser should show you the installation screen:



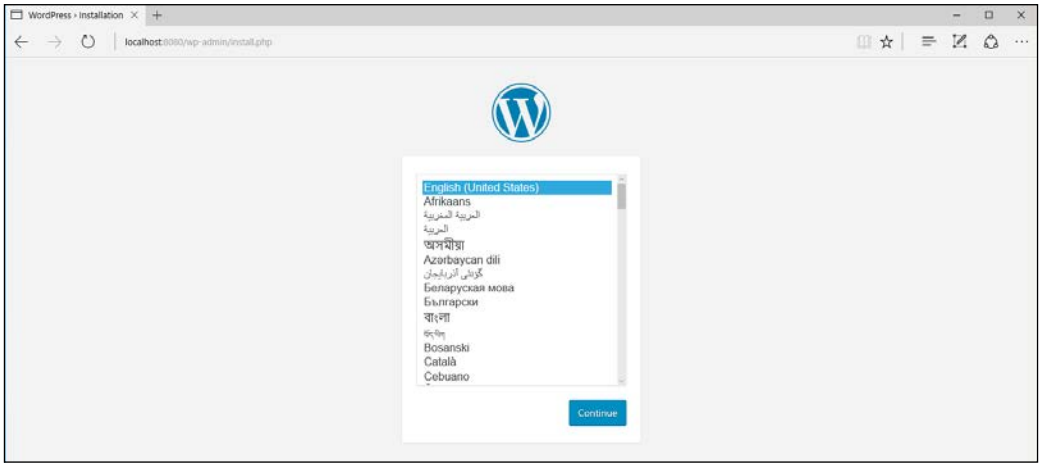
The process above works on Docker for Mac and on Linux; however for Docker for Windows you should add `.exe` to your Docker Compose commands:

```
cd .\chapter02\wordpress-compose
docker-compose.exe pull
docker-compose.exe up -d
docker container ps
```

This will give you something like the following output:



Again, opening your browser and going to `http://localhost:8080/` should show you the installation screen:



Before we move into the next section, let's stop and remove our WordPress containers by running the following commands:

```
docker-compose stop
docker-compose rm
```

Or if you are following using Docker for Windows:

```
docker-compose.exestop
docker-compose.exe rm
```

So far, we have been using images from the Docker Hub, next we will are going to take a look at customizing images.

Docker Build

Docker images are the fundamental building blocks of containers. These images could be very basic operating environments such, as `alpine` or `Ubuntu`. Or, the images could craft advanced application stacks for the enterprise and cloud IT environments. An automated approach of crafting Docker images is using a `Dockerfile`.

A `Dockerfile` is a text-based build script that contains special instructions in a sequence for building the right and the relevant images from the base images. The sequential instructions inside the `Dockerfile` can include the base image selection, installing the required application, adding the configuration and the data files, and automatically running the services as well as exposing those services to the external world. Thus, a Dockerfile-based automated build system has remarkably simplified the image-building process. It also offers a great deal of flexibility in the way in which the build instructions are organized and in the way in which they visualize the complete build process.

The Docker engine tightly integrates this build process with the help of the `docker build` subcommand. In the client-server paradigm of Docker, the Docker server (or daemon) is responsible for the complete build process and the Docker command line interface is responsible for transferring the build context, including transferring `Dockerfile` to the daemon.

To have a sneak peek into the `Dockerfile` integrated build system in this section, we introduce you to a basic `Dockerfile`. Then, we explain the steps for converting that `Dockerfile` into an image, and then launching a container from that image.

Our `Dockerfile` is made up of two instructions, as shown here (there is also a copy in the GitHub repo in the `chapter02/build_basic` folder):

```
FROM alpine:latest
CMD echo Hello World!!
```

In the following, we cover/discuss the two instructions mentioned earlier:

- The first instruction is for choosing the base image selection. In this example, we select the `apline:latest` image.
- The second instruction is for carrying out the command `CMD`, that instructs the container to `echo Hello World!!.`

Now, let's proceed towards generating a Docker image by using the preceding `Dockerfile` by calling `dockerimagebuild` along with the path of the `Dockerfile`. In our example, we will invoke the `dockerimagebuild` subcommand from the directory where we have stored the `Dockerfile,` and the path will be specified by the following command:

```
dockerimagebuild
```

After issuing the preceding command, the `build` process will begin by sending build context to the daemon and then display the text shown here:

Sending build context to Docker daemon 2.048 kB

Step 1/2 : FROM alpine:latest

The build process will continue and after completing itself, it will display the following:

Successfully built 0080692cf8db

In the preceding example, the image was built with `IMAGE ID0a2abe57c325`. Let's use this image to launch a container by using the `docker container run` subcommand as follows:

```
docker container run 0080692cf8db
```

Cool, isn't it? With very little effort, we have been able to craft an image with `alpine` as the base image, and we have been able to extend that image to produce `Hello World!!.`

This is a simple application, but the enterprise-scale images can also be realized by using the same methodology.

Now, let's look at the image details by using the `dockerimage ls` subcommand. Here, you may be surprised to see that the `IMAGE` (`REPOSITORY`) and `TAG` name have been listed as `<none>`. This is because we did not specify any image or any `TAG` name when we built this image. You could specify an `IMAGE` name and optionally a `TAG` name by using the `docker image tag` subcommand, as shown here:

```
docker image tag 0080692cf8dbbasicbuild
```

The alternative approach is to build the image with an image name during the `build` time by using the `-t` option for the `docker image build` subcommand, as shown here:

```
docker image build -t basicbuild
```

Since there is no change in the instructions in `Dockerfile`, the Docker engine will efficiently reuse the old image that has `ID0a2abe57c325` and update the image name to `basicbuild`. By default, the build system would apply `latest` as the `TAG` name. This behavior can be modified by specifying the `TAG` name after the `IMAGE` name by having a `:` separator placed in between them. That is, `<image name>:<tag name>` is the correct syntax for modifying behaviors, wherein `<image name>` is the name of the image and `<tag name>` is the name of the tag.

Once again, let's look at the image details by using the `docker image ls` subcommand, and you will notice that the image (Repository) name is `basicimage` and the tag name is `latest`. Building images with an image name is always recommended as the best practice.

Having experienced the magic of `Dockerfile`, in the subsequent sections, we will introduce you to the syntax or the format of `Dockerfile` and explain a dozen `Dockerfile` instructions.

By default `docker image build` subcommand uses the `Dockerfile` located at the build context. However `-f` option `docker image build` subcommand let's to specify an alternate `Dockerfile` in a different path or name.

A quick overview of the Dockerfile's syntax

In this section, we explain the syntax or the format of `Dockerfile`. A `Dockerfile` is made up of instructions, comments, parser directives and empty lines, as shown here:

```
# Comment

INSTRUCTION arguments
```

The instruction line of `Dockerfile` is made up of two components, where the instruction line begins with the instruction itself, which is followed by the arguments for the instruction. The instruction could be written in any case, in other words, it is case-insensitive. However, the standard practice or the convention is to use *uppercase* to differentiate it from the arguments. Let's take a relook at the content of `Dockerfile` in our previous example:

```
FROM apline:latest
CMD echo Hello World!!
```

Here, `FROM` is an instruction which has taken `apline:latest` as an argument, and `CMD` is an instruction which has taken `echo Hello World!!` as an argument.

The comment line

The comment line in `Dockerfile` must begin with the `#` symbol. The `#` symbol after an instruction is considered as an argument. If the `#` symbol is preceded by a whitespace, then the `docker image build` system would consider that as an unknown instruction and skip the line. Now, let's understand the preceding cases with the help of an example to get a better understanding of the comment line:

- A valid `Dockerfile` comment line always begins with a `#` symbol as the first character of the line:

  ```
  # This is my first Dockerfile comment
  ```

- The `#` symbol can be a part of an argument:

  ```
  CMD echo ### Welcome to Docker ###
  ```

- If the `#` symbol is preceded by a whitespace, then it is considered as an unknown instruction by the build system:

  ```
  # this is an invalid comment line
  ```

A sample `Dockerfile` can be found at `/chapter02/build_basic/` in the repo:

```
# Example of a really bsaicDockerfile

FROM alpine:latest
CMD echo Hello World!!
```

The `docker image build` system ignores any empty line in the `Dockerfile` and hence, the author of `Dockerfile` is encouraged to add comments and empty lines to substantially improve the readability of `Dockerfile`.

The parser directives

As the name implies, the parser directives instruct the `Dockerfile` parser to handle the content of the `Dockerfile` as specified in the directives. The parser directives are optional and they must be at the very top of a `Dockerfile`. Currently escape is the only supported directive.

We use escape character to escape characters in a line or to extend a single line to multiple lines. On a UNIX like platform, `\` is the escape character whereas on windows `\` is a directory path separator and `'` is the escape character. By default, `Dockerfile` parser considers `\` as the escape character and you could override this on windows using the escape parser directive as shown below:

```
# escape='
```

The Dockerfile build instructions

So far, we have looked at the integrated build system, the `Dockerfile` syntax and a sample lifecycle, wherein how a sample `Dockerfile` is leveraged for generating an image and how a container gets spun off from that image was discussed. In this section, we will introduce the `Dockerfile` instructions, their syntax, and a few befitting examples.

The FROM instruction

The `FROM` instruction is the most important one and it is the first valid instruction of a `Dockerfile`. It sets the base image for the build process. The subsequent instructions will use this base image and build on top of it. The Docker build system lets you flexibly use the images built by anyone. You can also extend them by adding more precise and practical features to them. By default, the Docker build system looks in the Docker host for the images.

However, if the image is not found in the Docker host, then the Docker build system will pull the image from the publicly available Docker Hub Registry. The Docker build system will return an error if it cannot find the specified image in the Docker host and the Docker Hub Registry.

The `FROM` instruction has the following syntax:

```
FROM <image>[:<tag>|@<digest>]
```

In the preceding code statement, note the following:

- `<image>`: This is the name of the image which will be used as the base image.
- `<tag>` or`<digest>`: Both tag and digest are optional attributes and you could qualify a particular Docker image version using either a tag or a digest. Tag `latest` is assumed by default if both tag and digest are not present.

Here is an example of the `FROM` instruction with the image name `centos`:

```
FROM centos
```

In the above example, the Docker build system would implicitly default to tag `latest` because neither a tag nor a digest is explicitly added to the image name.

You should be strongly discouraged from using multiple `FROM` instructions in a single `Dockerfile`, as damaging conflicts could arise.

The MAINTAINER instruction

All the `MAINTAINER` instruction does is enables the authors' details to set the in an image. Docker does not place any restrictions on placing the `MAINTAINER` instruction in a `Dockerfile`. However, it is strongly recommended that you should place it after the `FROM` instruction.

The following is the syntax of the `MAINTAINER` instruction, where `<author's detail>` can be in any text. However, it is strongly recommended that you should use the image, author's name and the e-mail address as shown in this code syntax:

```
MAINTAINER <author's detail>
```

There is an example of the `MAINTAINER` instruction with the author name, and the e-mail address at `/chapter02/build_01_maintainer/` in the repo:

```
# Example Dockerfile showing MAINTAINER

FROM alpine:latest
MAINTAINER Russ McKendrick<russ@mckendrick.io>
```

The RUN instruction

The `RUN` instruction is the real workhorse during the build time, and it can run any command. The general recommendation is to execute the multiple commands by using one `RUN` instruction. This reduces the layers in the resulting Docker image because the Docker system inherently creates a layer for each time an instruction is called in `Dockerfile`.

The `RUN` instruction has two types of syntax:

- The first is the shell type, as shown here:

  ```
  RUN <command>
  ```

  Here, the `<command>` is the shell command that has to be executed during the build time. If this type of syntax is to be used, then the command is always executed by using `/bin/sh -c`.

- The second syntax type is either exec or the JSON array, as shown here:

  ```
  RUN ["<exec>", "<arg-1>", ..., "<arg-n>"]
  ```

  Wherein, the code terms mean the following:

  - `<exec>`: This is the executable to run during the build time.
  - `<arg-1>, ..., <arg-n>`: These are the variables (zero or more) number of the arguments for the executable.

Unlike the first type of syntax, this type does not invoke `/bin/sh -c`. Hence, the types of shell processing, such as the variable substitution (`$USER`) and the wild card substitution (`*`, `?`), do not happen in this type. If shell processing is critical for you, then you are encouraged to use the shell type. However, if you still prefer the exec (JSON array type) type, then use your preferred shell as the executable and supply the command as an argument.

For example, `RUN ["bash", "-c", "rm", "-rf", "/tmp/abc"]`.

Let's add a few `RUN` instructions to our `Dockerfile` to install NGINX using `apk` and then set some permissions:

```
# Example Dockerfile showing RUN

FROM alpine:latest
MAINTAINER Russ McKendrick<russ@mckendrick.io>

RUN apk add --update supervisor nginx&&rm -rf /var/cache/apk/*
```

As you can see, we are installing NGINX and Supervisor. The `&&` has been added so that we can string several commands together on a single line, as each line within the `Dockerfile` creates a layer within the image stringing commands together like this streamlines your image file.

The COPY instruction

The `COPY` instruction enables you to copy the files from your Docker host to the filesystem of the image you are building. The following is the syntax of the `COPY` instruction:

```
COPY <src> ... <dst>
```

The preceding code terms bear the explanations shown here:

- `<src>`: This is the source directory, the file in the build context, or the directory from where the `docker build` subcommand was invoked.
- `...`: This indicates that multiple source files can either be specified directly or be specified by wildcards.
- `<dst>`: This is the destination path for the new image into which the source file or directory will get copied. If multiple files have been specified, then the destination path must be a directory and it must end with a slash `/`.

Using an absolute path for the destination directory or a file has been recommended. In the absence of an absolute path, the `COPY` instruction will assume that the destination path will start from root `/`. The `COPY` instruction is powerful enough for creating a new directory and for overwriting the filesystem in the newly created image.

An example of the `copy` command can be found in the repo (`https://github.com/russmckendrick/bootcamp`) at `/chapter02/build_03_copy/`:

```
# Example Dockerfile showing COPY

FROM alpine:latest
MAINTAINER Russ McKendrick<russ@mckendrick.io>

RUN apk add --update supervisor nginx&&rm -rf /var/cache/apk/*

COPY start.sh /script/
COPY files/default.conf /etc/nginx/conf.d/
COPY files/nginx.conf /etc/nginx/nginx.conf
COPY files/supervisord.conf /etc/supervisord.conf
```

This copies the `start.sh` file to the folder in the Docker image at `/script/` and the configuration file from the `files` folder to in place on the image.

The ADD instruction

The `ADD` instruction is like the `COPY` instruction. However, in addition to the functionality supported by the `COPY` instruction, the `ADD` instruction can handle the TAR files and the remote URLs. We can annotate the `ADD` instruction as `COPY` on steroids.

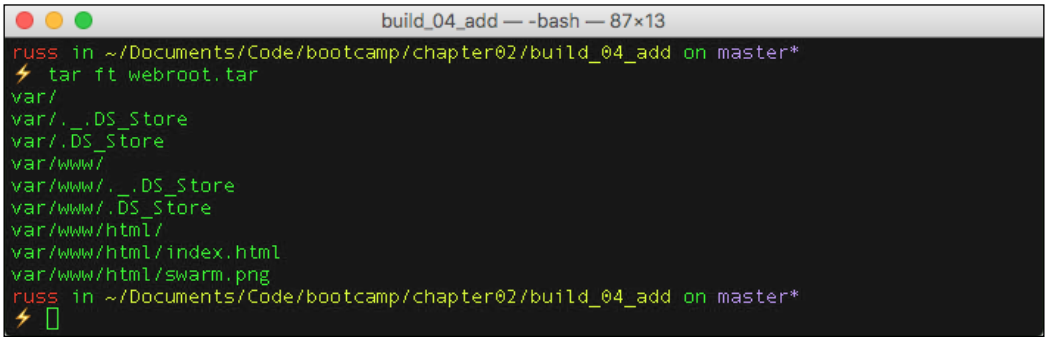
The following is the syntax of the `ADD` instruction:

```
ADD <src> ... <dst>
```

The arguments of the `ADD` instruction are very similar to those of the `COPY` instruction, as shown here:

- `<src>`: This is either the source directory or the file that is in the build context or in the directory from where the `docker build` subcommand will be invoked. However, the noteworthy difference is that the source can either be a `tar` file stored in the build context or be a remote URL.
- `...`: This indicates that the multiple source files can either be specified directly or be specified by using wildcards.
- `<dst>`: This is the destination path for the new image into which the source file or directory will be copied.

Here is an example for demonstrating the procedure for copying multiple source files to the various destination directories in the target image filesystem. In this example, we have taken a TAR file (`webroot.tar`) in the source build context with the `http` daemon configuration file and the files for the web pages are stored in the appropriate directory structure, as shown here:



The next line in the `Dockerfile` content has an `ADD` instruction for copying the TAR file (`webroot.tar`) to the target image and extracting the TAR file from the root directory (`/`) of the target image, as shown here in the example you can find in the repo at `/chapter02/build_04_add/`:

```
# Example Dockerfile showing ADD

FROM alpine:latest
MAINTAINER Russ McKendrick<russ@mckendrick.io>

RUN apk add --update supervisor nginx&&rm -rf /var/cache/apk/*

COPY start.sh /script/
COPY files/default.conf /etc/nginx/conf.d/
COPY files/nginx.conf /etc/nginx/nginx.conf
COPY files/supervisord.conf /etc/supervisord.conf

ADD webroot.tar /
RUN chown -R nginx:nginx /var/www/html
```

Thus, the TAR option of the `ADD` instruction can be used for copying multiples files to the target image, also note we have added a second `RUN` instruction to set the permissions on the folder we have just created using `ADD`.

The EXPOSE instruction

The `EXPOSE` instruction opens up a container network port for communicating between the container and the rest of the network.

The syntax of the `EXPOSE` instruction is as follows:

```
EXPOSE <port>[/<proto>] [<port>[/<proto>]...]
```

Here, the code terms mean the following:

- `<port>`: This is the network port that has to be exposed to the outside world.
- `<proto>`: This is an optional field provided for a specific transport protocol, such as TCP and UDP. If no transport protocol has been specified, then TCP is assumed to be the transport protocol.

The `EXPOSE` instruction allows you to specify multiple ports in a single line.

The following is an example of the `EXPOSE` instruction inside a `Dockerfile` exposing port `80`:

```
# Example Dockerfile showing EXPOSE

FROM alpine:latest
MAINTAINER Russ McKendrick<russ@mckendrick.io>

RUN apk add --update supervisor nginx&&rm -rf /var/cache/apk/*

COPY start.sh /script/
COPY files/default.conf /etc/nginx/conf.d/
COPY files/nginx.conf /etc/nginx/nginx.conf
COPY files/supervisord.conf /etc/supervisord.conf

ADD webroot.tar /

RUN chown -R nginx:nginx /var/www/html

EXPOSE 80/tcp
```

The ENTRYPOINT instruction

The `ENTRYPOINT` instruction will help in crafting an image for running an application (entry point) during the complete lifecycle of the container, which would have been spun out of the image. When the entry point application is terminated, the container would also be terminated along with the application and vice versa.

Thus, the `ENTRYPOINT` instruction would make the container function like an executable. Functionally, `ENTRYPOINT` is akin to the `CMD` instruction which we will look at next, but the major difference between the two is that the entry point application is launched by using the `ENTRYPOINT` instruction, which cannot be overridden by using the `docker run` subcommand arguments.

However, these `docker container run` subcommand arguments will be passed as additional arguments to the entry point application. Having said this, Docker provides a mechanism for overriding the entry point application through the `--entrypoint` option in the `docker container run` subcommand. The `--entrypoint` option can accept only word as its argument and hence, it has limited functionality.

Syntactically, the `ENTRYPOINT` instruction is very similar to the `RUN`, and the `CMD` instructions, and it has two types of syntax, as shown here:

- The first type of syntax is the shell type, as shown here:

  ```
  ENTRYPOINT <command>
  ```

  Here, `<command>` is the shell command, which is executed during the launch of the container. If this type of syntax is used, then the command is always executed by using `/bin/sh -c`.

- The second type of syntax is `exec` or the JSON array, as shown here:

  ```
  ENTRYPOINT ["<exec>", "<arg-1>", ..., "<arg-n>"]
  ```

  Wherein, the code terms mean the following:

  - `<exec>`: This is the executable, which has to be run during the container launch time.
  - `<arg-1>, ..., <arg-n>`: These are the variable (zero or more) number of arguments for the executable.

Syntactically, you can have more than one `ENTRYPOINT` instruction in a `Dockerfile`. However, the build system will ignore all the `ENTRYPOINT` instructions except the last one. In other words, in the case of multiple `ENTRYPOINT` instructions, only the last `ENTRYPOINT` instruction be effective.

As you may recall from when we covered the `RUN` instruction we installed a service called `supervisord`, we will be using this for the entry point in our image meaning that our `Dockerfile` now looks like the following:

```
# Example Dockerfile showing ENTRYPOINT

FROM alpine:latest
MAINTAINER Russ McKendrick<russ@mckendrick.io>

RUN apk add --update supervisor nginx&&rm -rf /var/cache/apk/*

COPY start.sh /script/
COPY files/default.conf /etc/nginx/conf.d/
COPY files/nginx.conf /etc/nginx/nginx.conf
COPY files/supervisord.conf /etc/supervisord.conf
```

```
ADD webroot.tar /

RUN chown -R nginx:nginx /var/www/html

EXPOSE 80/tcp

ENTRYPOINT ["supervisord"]
```

Now we could leave it here and the image would be functional, however there is one instruction we should pass to our image.

The CMD instruction

The `CMD` instruction can run any command (or application), which is similar to the `RUN` instruction. However, the major difference between those two is the time of execution. The command supplied through the `RUN` instruction is executed during the build time, whereas the command specified through the `CMD` instruction is executed when the container is launched from the newly created image. Thus, the `CMD` instruction provides a default execution for this container. However, it can be overridden by the `docker run` subcommand arguments. When the application terminates, the container will also terminate along with the application and vice versa.

On the face of it the `CMD` instruction is very similar to the `RUN` instruction in that it can run any command passed to it, however there is a major difference between the two instructions.

The command passed to the `RUN` instruction is executed at build time and commands passed using the `CMD` instruction are executed at run time meaning you can define the default execution for the container. This means if no command is passed during the `docker container run` command then the CMD will executed.

The `CMD` instruction has three types of syntax, as shown here:

- The first syntax type is the shell type, as shown here:

  `CMD <command>`

  Wherein, the `<command>` is the shell command, which has to be executed during the launch of the container. If this type of syntax is used, then the command is always executed by using `/bin/sh -c`.

- The second type of syntax is exec or the JSON array, as shown here:

  ```
  CMD ["<exec>", "<arg-1>", ..., "<arg-n>"]
  ```

  Wherein, the code terms mean the following:

  - `<exec>`: This is the executable, which is to be run during the container launch time
  - `<arg-1>, ..., <arg-n>`: These are the variable (zero or more) number of the arguments for the executable

- The third type of syntax is also exec or the JSON array, which is similar to the previous type. However, this type is used for setting the default parameters to the `ENTRYPOINT` instruction, as shown here:

  ```
  CMD ["<arg-1>", ..., "<arg-n>"]
  ```

  Wherein, the code terms mean the following:

  - `<arg-1>, ..., <arg-n>`: These are the variables (zero or more) number of the arguments for the `ENTRYPOINT` instruction, which will be explained in the next section.

Syntactically, you can add more than one `CMD` instruction in `Dockerfile`. However the build system would ignore all the `CMD` instructions except for the last one. In other words, in the case of multiple `CMD` instructions, only the last `CMD` instruction would be effective.

As mentioned in the previous section, our `Dockerfile` could have been run with just the `ENTRYPOINT` instruction defined, however that would give a non-breaking error when `supervisiord` starts up so let's pass a flag which defines where our supervisor configuration file is using the `CMD` instruction:

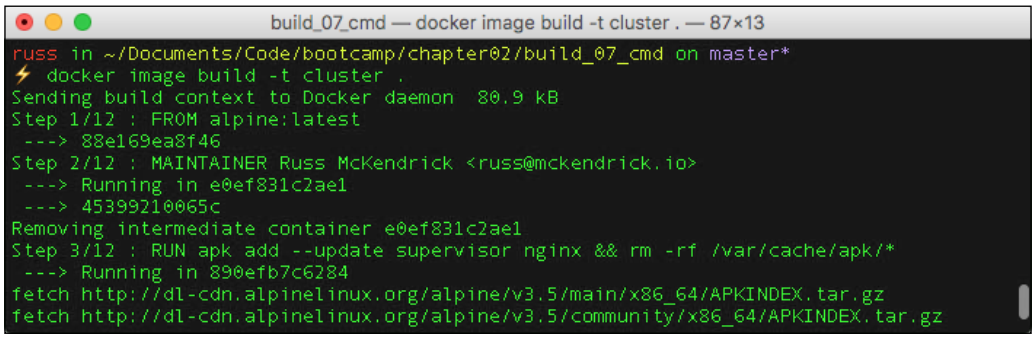
```
# Example Dockerfile showing CMD

FROM alpine:latest
MAINTAINER Russ McKendrick<russ@mckendrick.io>

RUN apk add --update supervisor nginx&&rm -rf /var/cache/apk/*

COPY start.sh /script/
COPY files/default.conf /etc/nginx/conf.d/
```

```
COPY files/nginx.conf /etc/nginx/nginx.conf
COPY files/supervisord.conf /etc/supervisord.conf

ADD webroot.tar /

RUN chown -R nginx:nginx /var/www/html

EXPOSE 80/tcp

ENTRYPOINT ["supervisord"]
CMD ["-c","/etc/supervisord.conf"]
```

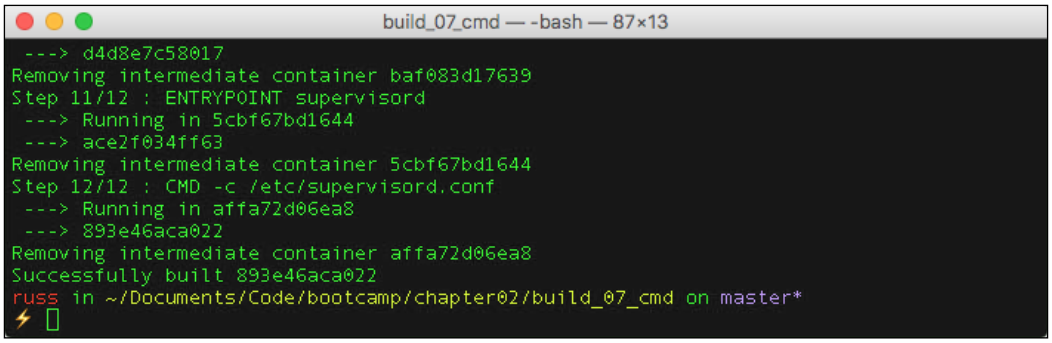
We are now in a position where we can build our image, you can find our completed `Dockerfile` in the `/chapter02/build_07_cmd/` folder in the repo, to build the image simple run the following command:

```
docker image build -t cluster
```

This will kick of the build, as you can see from the following terminal:

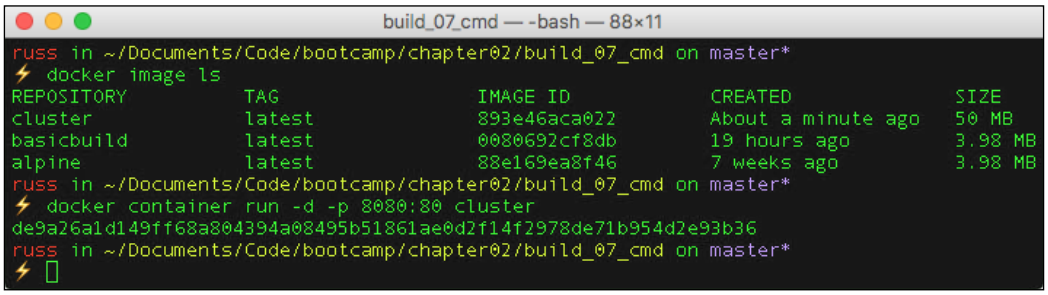


There are 12 steps in the build, it will take a minute or two, but once compete you should see something like the following terminal output:

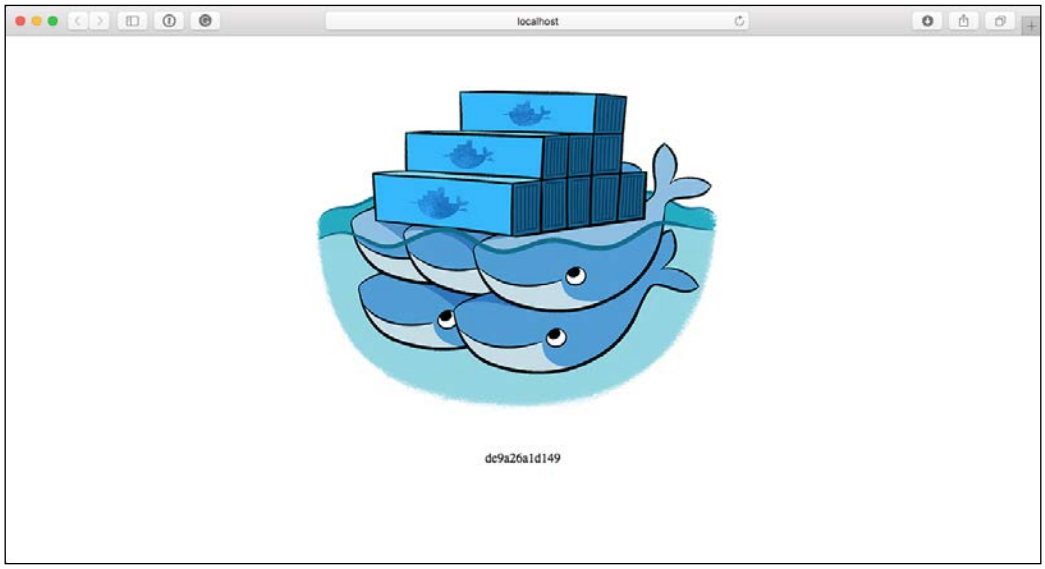


Once you have your image built, you can check and then run it by using the following commands:

```
docker image ls
docker container run -d -p 8080:80 cluster
```



Now the container is running, opening your browser and going `http://localhost:8080/` should show you something like the following page:



There you have it, we have created an image:

- Using the Alpine Linux base (`FROM`)
- Installed `NGINX` and `supervisord` using `apk` (`RUN`)
- Copied the configuration from our Docker host to the image (`COPY`)
- Uploaded and extracting our web root (`ADD`)
- Set the correct ownership of our web root (`RUN`)
- Ensured that port `80` on the container is open (`EXPOSE`)
- Made sure that `supervisord` is the default process (`ENTRYPOINT`)
- Passed the configuration file flag to `supervisord` (`CMD`)

Before moving onto the next section you can stop and remove the container by running the following command making sure you replace the container ID with that of yours:

```
docker container ps
docker container stop de9a26a1d149
docker container rm de9a26a1d149
```

Then remove the image we created by running:

```
docker image rm cluster
```

Next, we are going to go back to our WordPress image and customize it.

Customizing existing images

While the official images should provide you with a fully functioning usable image you may sometimes need to install additional software, in this case we are going to look at installing WordPress CLI using the official WordPress image.

WordPress CLI is a set of command line tools which allow you to manage your WordPress configuration and installation; for more information, see `http://wp-cli.org/`.

You can find a copy of the `Dockerfile` below in the `/chapter02/wordpress-custom/` folder in the repo, as you can see we are just running `RUN` and `COPY`instructions:

```
# Adds wp-cli to the offical WordPress image
FROM wordpress:latest
MAINTAINER Russ McKendrick<russ@mckendrick.io>

# Install the packages we need to run wp-cli
RUN apt-get update &&\
apt-get install -y sudo less mysql-client &&\
curl -o /bin/wp-cli.pharhttps://raw.githubusercontent.com/wp-cli/
builds/gh-pages/phar/wp-cli.phar

# Copy the wrapper for wp-cli and set the correct execute permissions
COPY wp /bin/wp
RUN chmod 755 /bin/wp-cli.phar /bin/wp

# Clean up the installation files
RUN apt-get clean &&rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/
```

You can build the image using the following command:

```
docker image build -t wordpress-custom .
```

Once it has finished building use the following command to check the image:

```
docker image ls
```

However, as we discovered earlier in this chapter it is easier to launch WordPress using Docker Compose, before we do lets remove the image we just built by running:

```
docker image rm wordpress-custom
```

Docker Compose can also trigger builds. Our updated `docker-compose.yml` file can be found in the `/chapter02/wordpress-custom/` folder and below:

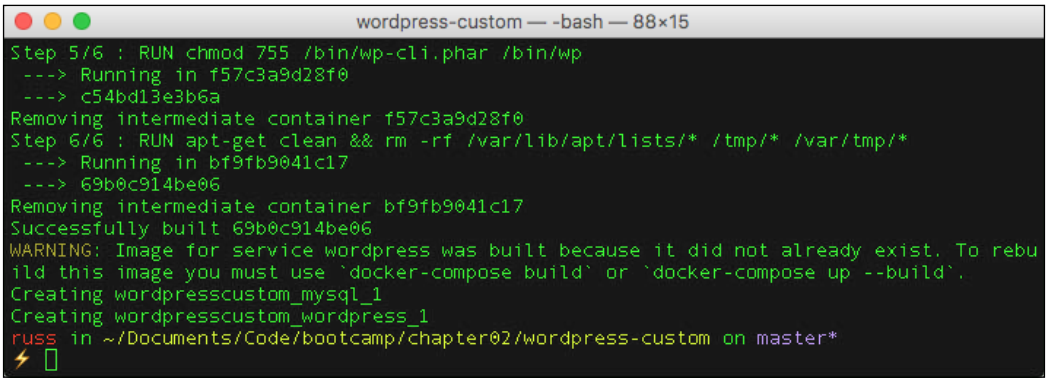
```
version: "3"

services:
mysql:
     image: mysql
     volumes:
       - db_data:/var/lib/mysql
     restart: always
     environment:
       MYSQL_ROOT_PASSWORD: wordpress
       MYSQL_DATABASE: wordpress
wordpress:
depends_on:
       - mysql
     build: ./
     ports:
       - "8080:80"
     restart: always
     environment:
       WORDPRESS_DB_PASSWORD: wordpress

volumes:
db_data:
```

As you can see, it is almost exactly the same as our original `docker-compose.yml` apart from now we have a line that says `build: ./"` rather than `image: wordpress"`.

To launch our WordPress installation, we simply need to run the following command:

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

This will pull and build the container images, once complete you should see something like the following in your terminal:

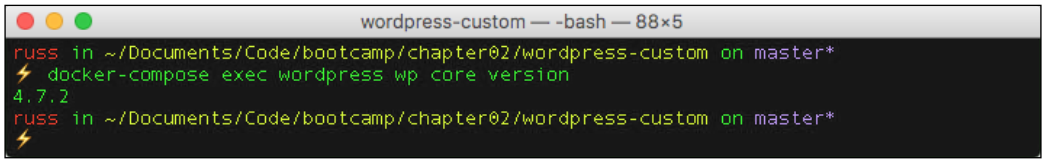


Going to `http://localhost:8080/` should show you the installation screen, however, we are going to typing a few commands to configure WordPress using the WordPress CLI.

First, let's check the version of WordPress we are working with by running:

```
docker-compose exec wordpress wp core version
```

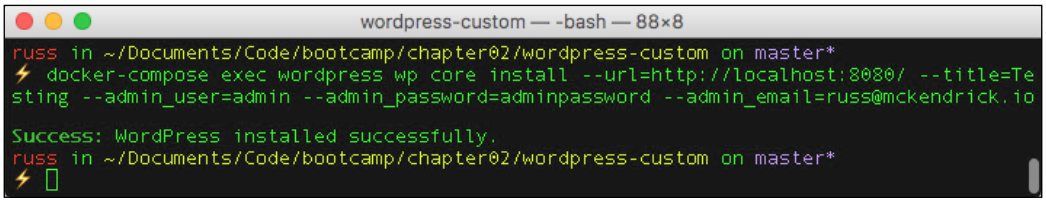
This will connect to the WordPress service and run the `wp core version` command, then return the output:



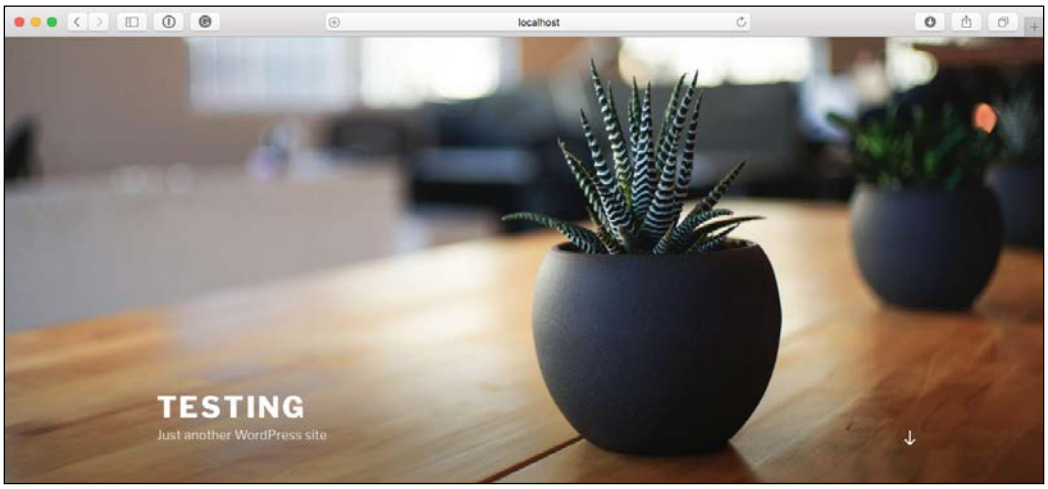
Next, we are going to install WordPress using the `wp core install` command, change the `title`, `admin_user`, `admin_password` and `admin_email` values as you like:

```
docker-compose exec wordpress wp core install --url=http://
localhost:8080/ --title=Testing --admin_user=admin --admin_
password=adminpasswIt ord --admin_email=russ@mckendrick.io
```

Once the command has finished running you should receive a message saying **Success: WordPress installed successfully**:



Going to `http://localhost:8080/` should show you a WordPress site rather than an installation prompt:



Once you have finished with your WordPress installation you can stop and remove it by running:

```
docker-compose stop
docker-compose rm
```

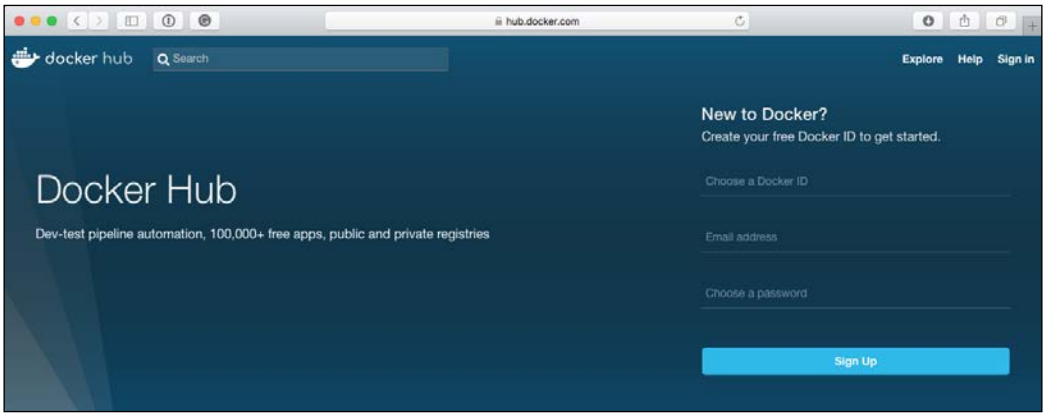
Now we know how to build an image we are going to look at a few different ways to share them.

Sharing your images

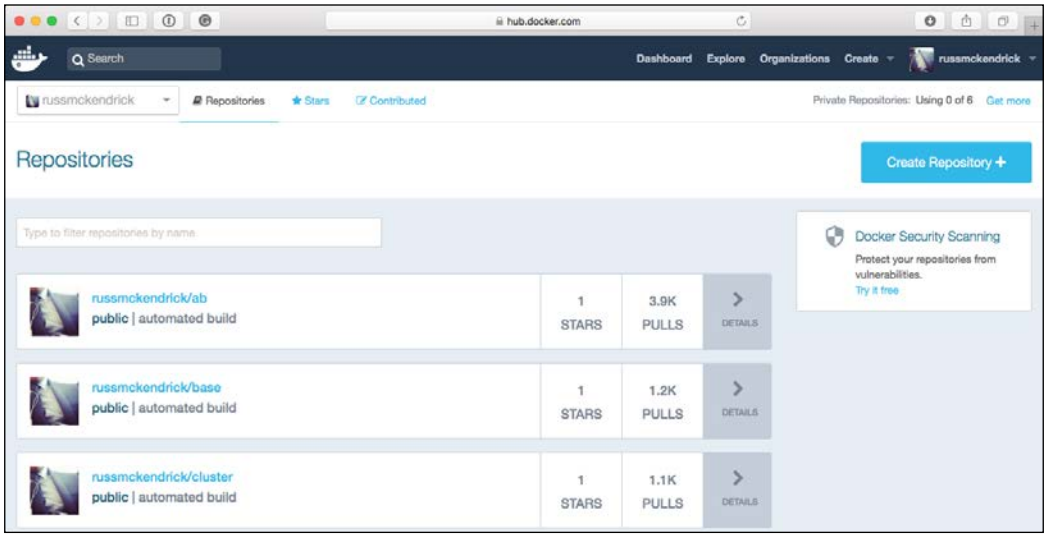
The Docker Hub is a central place used for keeping the Docker images either in a public or private repository.

The Docker Hub provides features, such as a repository for Docker images, user authentications, automated image builds, integration with GitHub or Bitbucket, and managing organizations and groups. The Docker Registry component of the Docker Hub manages the repository.

To work with the Docker Hub, you must register an account using the link at `https://hub.docker.com/`.You can update the **Docker Hub ID**, **Email Address** and **Password** as shown in the following screenshot:



After completing the **Sign Up** process, you need to complete the verification received in an e-mail. After the e-mail verification is completed, you will see something similar to the following screenshot, when you login to the Docker Hub:

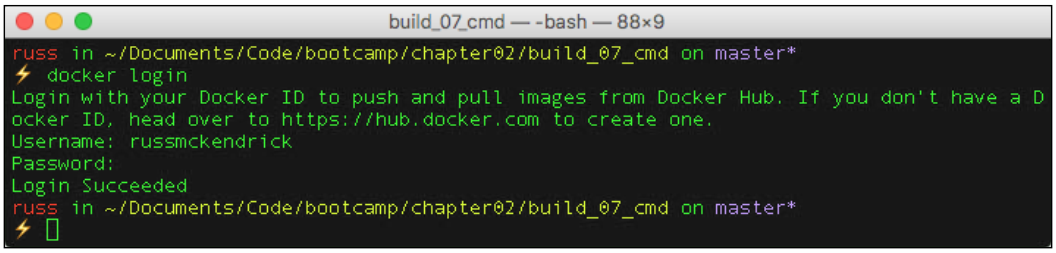


As you can see, I already have a few automated builds configured, we will get to these later on, for now we are going to look at pushing an image from our local Docker host.

First, we need to login to the Docker Hub using the Docker client on the command line, to do this simply use the following command:

```
docker login
```

You should be prompted for your Docker Hub username and password:



Now we are ready to start committing and pushing images to the Docker Hub.

We'll again create an image using the `Dockerfile` we created earlier in the chapter. So, let's create the Docker image using the `Dockerfile` in `/chapter02/build_07_cmd`and push the resulting image to the Docker Hub.

Now we build the image locally using the following command making sure to use your own Docker Hub username in place of mine:

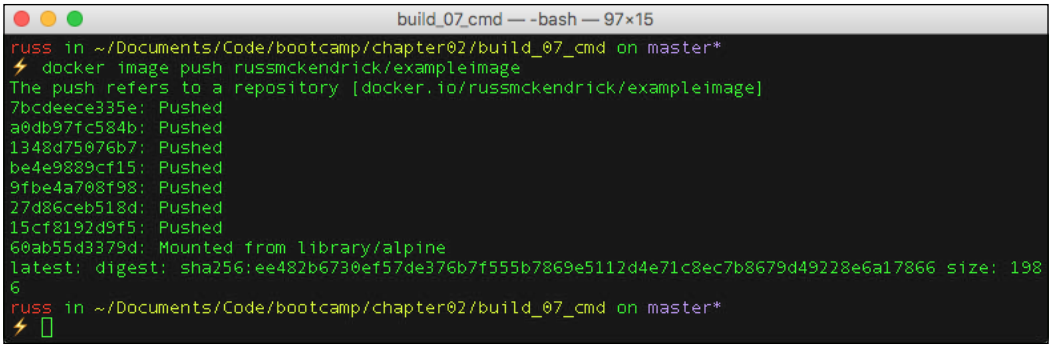
```
docker image build -t russmckendrick/exampleimage .
```

Once built, you can check the image is there by using:

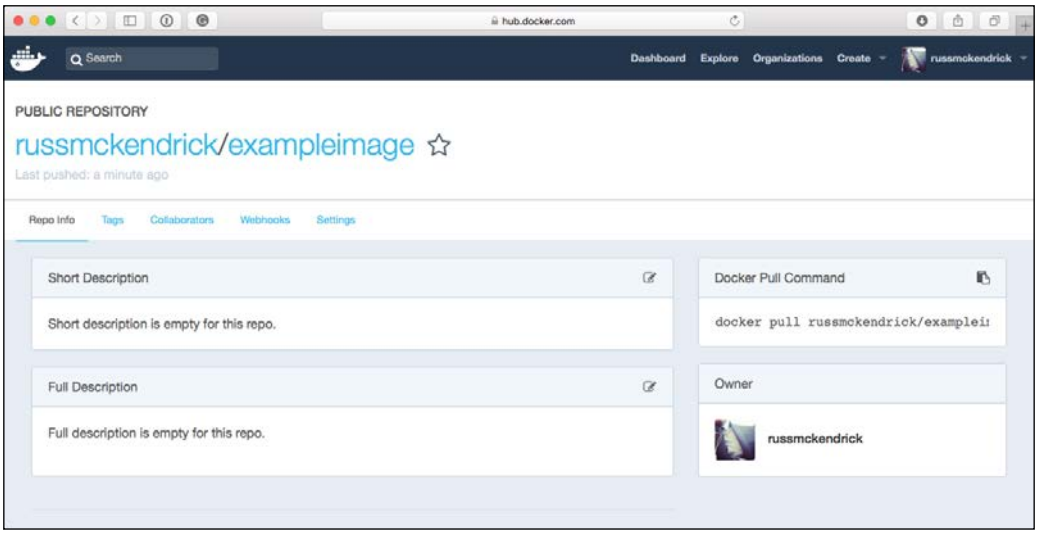
```
docker image ls
```

As we are already logged in all we need to do to push the newly create image is run the following command:

```
docker image push russmckendrick/exampleimage
```



Finally, we can verify the availability of the image on the Docker Hub:



This is where I should probably issue a warning: as you have just experienced it is very easy to publish images to the Docker Hub using the `docker image push` command; however, it is very easy to accidentally push content you maybe wouldn't want to be publicly available. For example, with a simple `COPY` or `ADD` instruction in your `Dockerfile` it is easy to bake sensitive information such as password credentials, certificates keys and non-publicly available code to a publicly accessible Docker Image repository.

It is this reason why I prefer to share a `Dockerfile` or `docker-compose.yml` files with my colleagues using private Git repositories and a good set of instructions . A also, it allows then to check what it is they are going to be running as they are able to review the`Dockerfile` and `docker-compose.yml` files; in fact, they can make changes and share them with me.

Summary

We have covered a lot in this chapter. We have used the Docker command line client to launch and interact with containers. We also used Docker Compose to define multiple container based application, namely WordPress and created and published our own Docker images on the Docker Hub. Finally, we customized the official WordPress Docker image adding additional functionality.

I am sure you will agree that so far using Docker has felt quite intuitive; in our next chapter we will move off our local Docker host and interact with Docker installations on remote hosts.

3
Docker in the Cloud

The third tool, alongside Docker and Docker Compose, which we installed during *Chapter 1*, *Installing Docker Locally* was Docker Machine; this is a command line tool which allows you to manage both local and remote Docker hosts.

In this chapter, we are going to look at the basic usage of three of the public cloud drivers by using Docker Machine to provision Docker hosts in them. We will be launching our Docker hosts in the following:

- Digital Ocean `https://www.digitalocean.com/`
- Amazon Web Services - `https://aws.amazon.com/`
- Microsoft Azure - `https://azure.microsoft.com/`

All using a single command.

Docker Machine

Docker Machine can connect to the following services, provision a Docker host, and configure your local Docker client to be able to communicate with the newly launched remote instance the following:

As well as the three public cloud providers already mentioned, Docker Machine also supports:

- Google Compute Engine - `https://cloud.google.com/compute/`
- Rackspace - `http://www.rackspace.co.uk/cloud/`
- IBM Softlayer `http://www.softlayer.com`
- Exoscale - `https://www.exoscale.ch/`
- VMware vCloud Air - `http://vcloud.vmware.com/`

It also supports the following self-hosted platforms:

- OpenStack - `https://www.openstack.org/`
- Microsoft Hyper-V - `http://www.microsoft.com/virtualization/`
- VMware vSphere - `http://www.vmware.com/uk/products/vsphere/`

Also, it allows you to launch Docker hosts locally using VirtualBox - `https://www.virtualbox.org/`. This is great if your local workstation doesn't meet the minimum specifications for Docker for Mac or Windows.

The Digital Ocean driver

Let us start creating some instances in the cloud. First, let us launch a machine in Digital Ocean.

There are two prerequisites for launching an instance with Docker Machine in Digital Ocean, the first is a Digital Ocean account and the second is an API token.

To sign up for a Digital Ocean account, please visit `https://www.digitalocean.com/` and click **Sign Up**. Once you have or are logged in to your account, you can generate an API token by clicking on the **API** link in the top menu.

To grab your token, click on **Generate New Token** and follow the onscreen instructions.

You only get one chance to make a record of your token; please make sure you store it somewhere safe as it will allow anyone who has it to launch instances into your account.

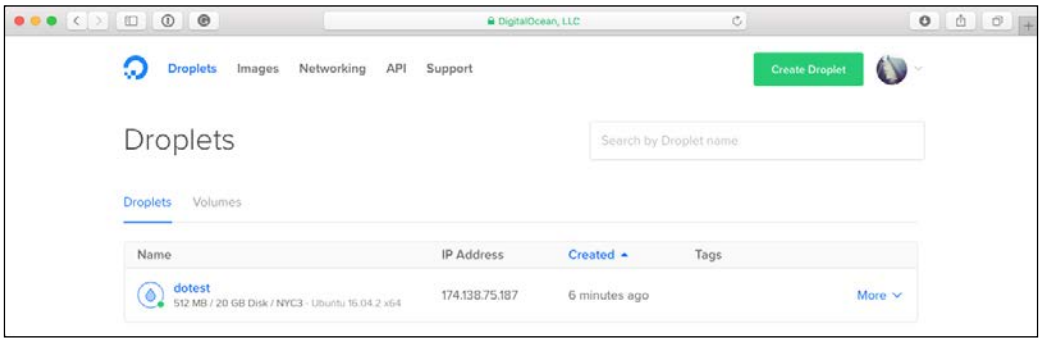
Once you have the token, you can launch your instance using Docker Machine. To do this, run the following command, making sure to replace the example API token with your own:

```
docker-machine create \
    --driver digitalocean \
    --digitalocean-access-token
14760f5bdee403cebb36117c22c83e5ee51188515f493a6c0d281c094c552536 \
    dotest
```

Please note, the tokens used in these examples have been revoked.

This will launch an instance called **dotest** in your Digital Ocean account.

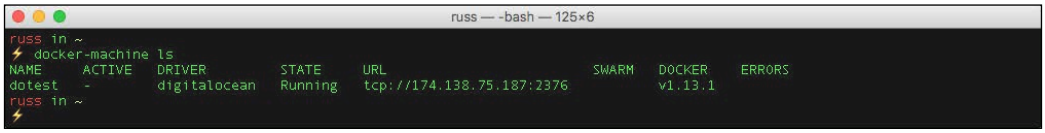
If you check your Digital Ocean control panel, you should now see the instance which was created by Docker Machine listed:



We can also confirm our Digital Ocean Docker host is running by using the following command:

```
docker-machine ls
```

This will return all the machines we have running, confirming their state, IP address, Docker version, and name. There is also a column which lets you know which of the Docker Machine managed Docker hosts your local client is configured to communicate with:



By default, your local Docker client is configured to communicate with our local Docker installation; as we launched our local Docker installation using Docker for Mac or Windows, or you have Docker installed on Linux Docker Machine will not list it.

Let's change it so it interacts with the Digital Ocean instance.

To do this, you have to change some local environment variables; luckily, Docker Machine provides an easy way to find out what these are and change them.

To find out what they all you must do is simply run the following command:

```
docker-machine env dotest
```

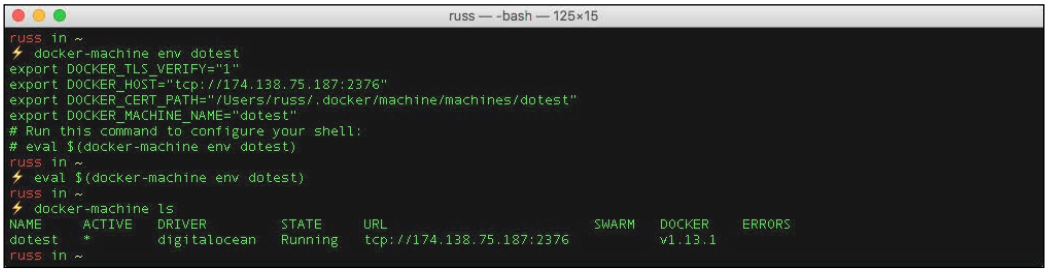
This will tell you exactly what you need to run to change from the `default` machine to `dotest`; the best thing is that the command itself formats the results in such a way which they can execute, so if we run the command again, but this time in a way where the output will be executed:

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

Or if you have launched your instance using PowerShell on Windows then use:

```
docker-machine env dotest | Invoke-Expression
```

And now if you get a listing from Docker Machine, you will notice that the `dotest` environment is now the active one:



Now we have our Digital Ocean instance active, you can run the `docker container run` command on your local machine, and they will have been executed on the Digital Ocean instance; let's test this by running the `hello-world` container.

Run the following command:

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

You should see the image download and then the output of running the `hello-world` container if you then run the following command:

```
docker container ls -a
```

You should see that the `hello-world` container exited a few seconds ago.

You can SSH into the Digital Ocean instance using the following command:

```
docker-machine ssh dotest
```

Once logged in, run the `docker container ls -a` command to demonstrate that the `docker container run` you ran locally was executed on the Digital Ocean instance.

The beauty of this setup is that you shouldn't have to SSH to your instances often.

One thing you may have noticed is that all we told Docker Machine is that we want to use Digital Ocean and our API token; at no point did we tell it which region to launch the instance in, what specification we wanted, or even which SSH key to use.

Docker Machine has some sensible defaults which are as follows:

- digitalocean-image = ubuntu-16-04-x64
- digitalocean-region = nyc3
- digitalocean-size = 512mb

As I am based in the UK, let's look at changing the region and specification of the host launched by Docker Machine.

First, we should remove the `dotest` instance by running the following command:

```
docker-machine rm dotest
```

This will terminate the 512 MB instance running in NYC3.

It is important to terminate instances you are not using as they will will cost you for each hour they are active; remember, one of the key advantages of using Docker Machine is that you can quickly spin up instances both quickly and with as little interaction as possible.

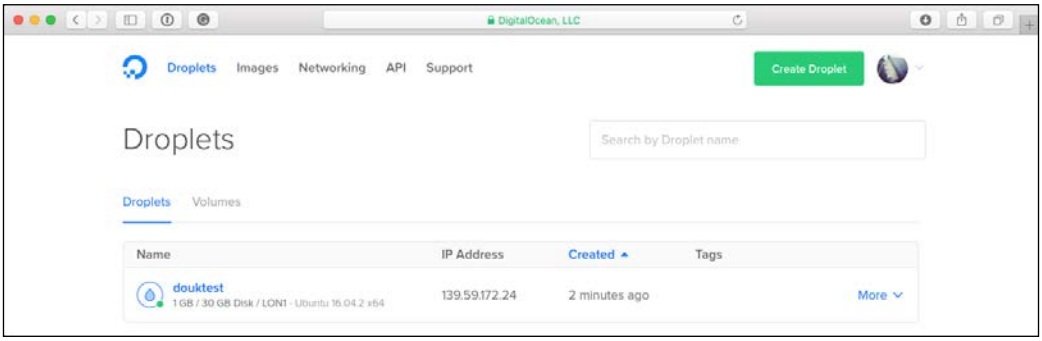
Now we have removed the old instance, let's add some additional flags to our `docker-machine` command to launch the new instance in the desired region and specification, we will be calling our new instance `douktest`. The updated `docker-machine create` command now looks like this (again, remember to replace the example API token with your own):

```
docker-machine create \
    --driver digitalocean \
    --digitalocean-access-token
14760f5bdee403cebb36117c22c83e5ee51188515f493a6c0d281c094c552536\
    --digitalocean-region lon1 \
    --digitalocean-size 1gb \
    douktest
```

You should see similar output from the command as before; once the instance has been deployed, you can make it active by running the following command:

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

When you enter the control panel, you will notice that the instance has launched in the specified region and at the desired specification:



For full details on each of the regions and what machine types are available in each one you can query the Digital Ocean API by running the following command (again, remember to replace the API token):

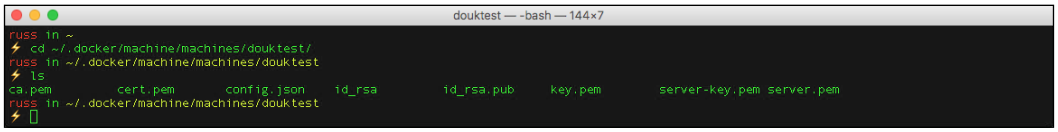
```
curl -X GET -H "Content-Type: application/json" -H "Authorization:
Bearer 14760f5bdee403cebb36117c22c83e5ee51188515f493a6c0d281c094c552536"
"https://api.digitalocean.com/v2/regions" | python -mjson.tool
```

This will output information about each region.

One last thing; we still haven't found out about the SSH key. Each time you run Docker Machine a new SSH key for the instance you are launching is created and uploaded to the provider; each key is stored in the `.docker` folder in your users home directory. For example, the key for `douktest` can be found by running:

```
cd ~/.docker/machine/machines/douktest/
```

Here you will also find the certificates used to authenticate the Docker agent with the Docker installation on the instance and the configuration:



So that covers launching a host in Digital Ocean; how about launching something more exciting than the Hello World container?

No problem, let's use Docker Compose to launch a variation of the WordPress stack we used in *Chapter 2*, *Launching Applications Using Docker*. Start by going to the `/bootcamp/chapter03/wordpress` folder and then run the following command:

```
docker-machine ls
```

To check you have your Docker client configured to use your Digital Ocean Docker host. Once you are sure your client is using the remote host, simply run:

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

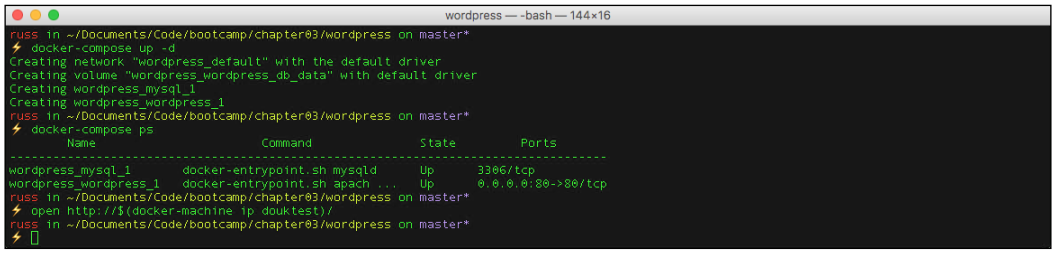
This will download the images we need, then launch two containers. This time you will be able to access the WordPress installation on `port 80` on your Digital Ocean host. To find the IP of your host, you can run the following command:

```
docker-machine ip douktest
```

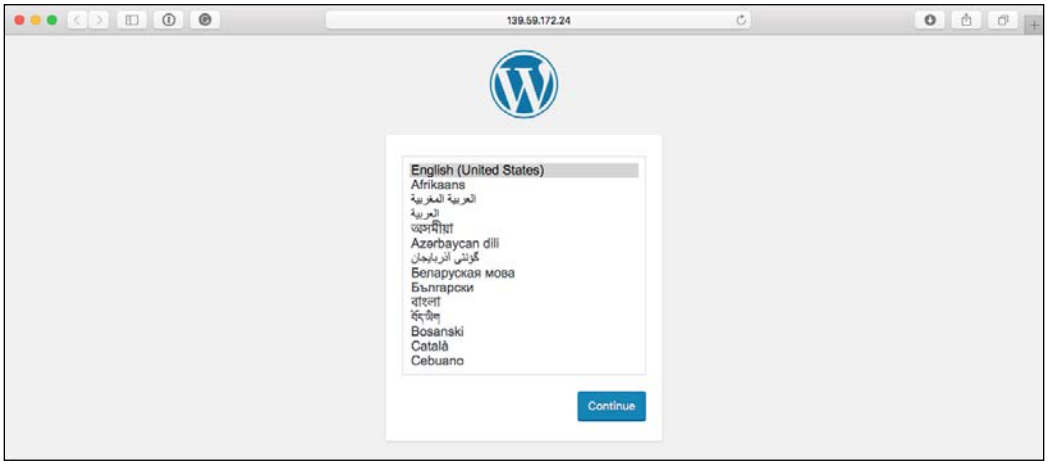
Or on a Mac or Linux machine to open your browser and go to your installation page run the following command:

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

The terminal session below shows the output you can expect to see from the previous commands:



You will then be able to complete your WordPress installation:



I wouldn't recommend leaving your WordPress installation at the installation screen for long as it is possible that someone could complete the installation on your behalf and get up to no good.

Once you have finished your Digital Ocean, host run the following command to terminate it:

```
docker-machine rm douktest
```

Now that we have learned how to launch a Docker host in Digital Ocean let's move on to Amazon Web Services.

The Amazon Web Services driver

If you don't already have an **Amazon Web Services** (**AWS**) account, you should sign up for one at `http://aws.amazon.com/`; if you are new to AWS, then you will be eligible for their free tier `http://aws.amazon.com/free/`.

I would recommend reading through Amazon's getting started guide if you are unfamiliar with AWS before working through this section of the chapter; you can find the guide at `http://docs.aws.amazon.com/gettingstarted/latest/awsgsg-intro/gsg-aws-intro.html`.

The AWS driver is like the Digital Ocean driver in that it has some sensible defaults, Rather than going into too much detail about how to customize the EC2 instance launched by Docker Machine, we will stick with the defaults. For AWS driver, these are as follows:

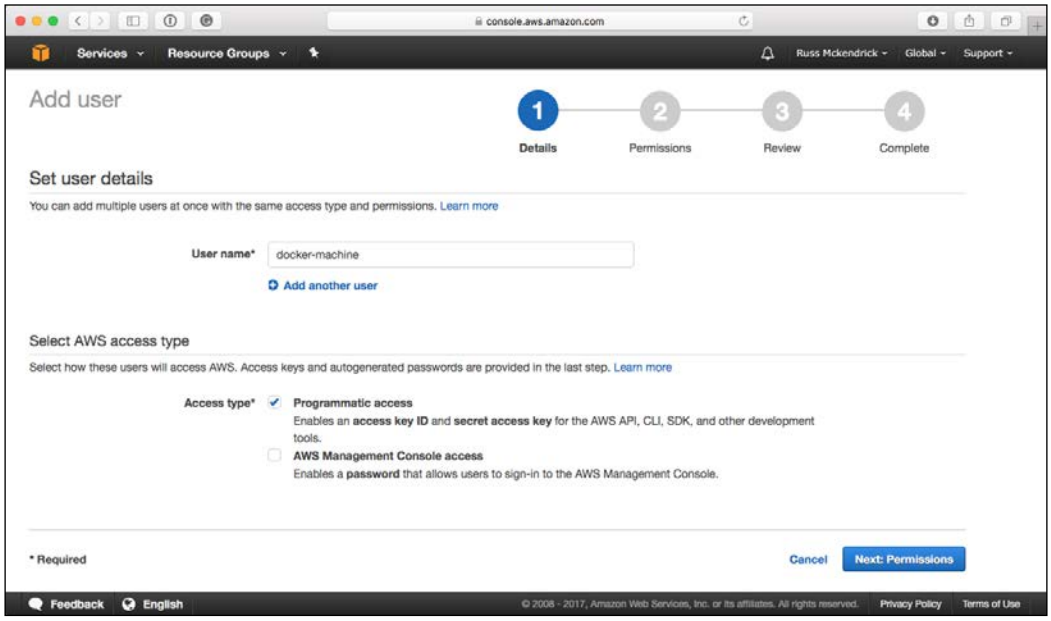
- amazonec2-region = us-east-1 (North Virginia)
- amazonec2-ami = ami-fd6e3bea (Ubuntu 16.04)
- amazonec2-instance-type = t2.micro
- amazonec2-root-size = 16GB
- amazonec2-security-group = docker-machine

Please note, if **amazonec2-security-group** does not exist, it will be created for you by Docker Machine; if it does exist, then Docker Machine will use the pre-existing rules instead.

Before we launch our instance, we will also need to know our **AWS Access** and **AWS Secret** keys and the **VPC ID** we will be launching our instance into; to get these, please log in to the AWS console which can be found at `https://console.aws.amazon.com/`.

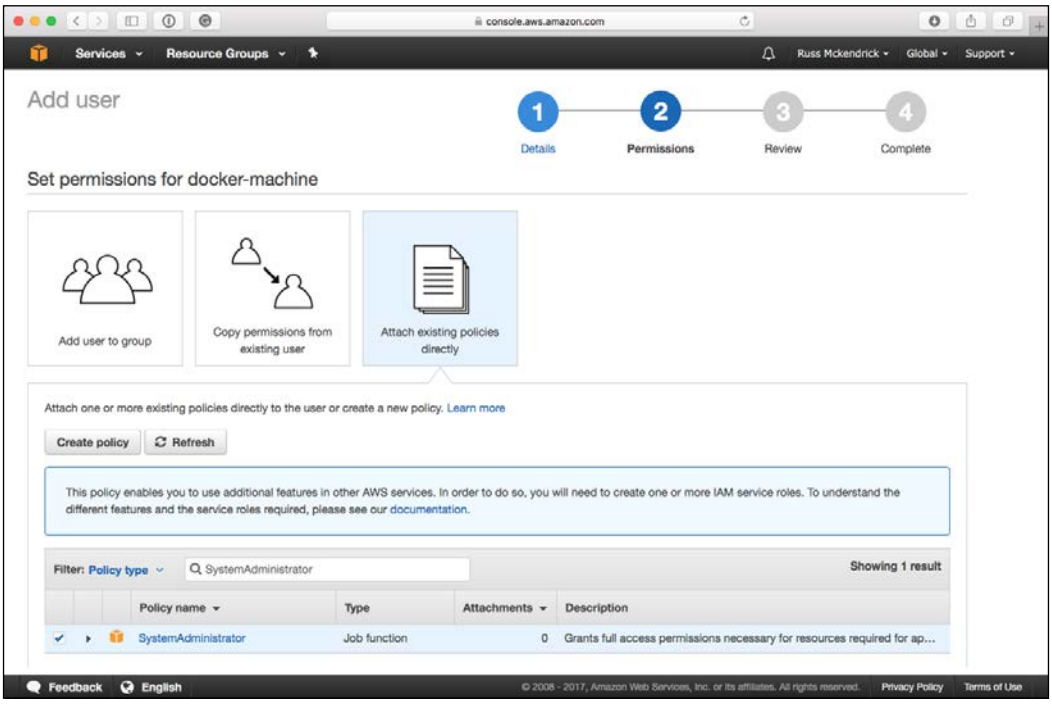
Most of you will be logging with your AWS root account. As your AWS root account shouldn't have any Access and Secret keys associated with it we should add a separate user for Docker Machine by going to **Services** | **IAM** | **Users** and then selecting your user and going to the **Security Credential**s tab.

There you should see a button which says **Add user**, click this and you will be taken to a screen where you can set your user details. Enter the **User name** `docker-machine` and then for the **Access type** tick the **Programmatic access** check box:



When you have entered the details, click on **Next: Permissions** to be taken to the next step. On the permissions page, click on **Attach existing policies directly** and then in the **Policy type** search box, enter `SystemAdministrator` and hit return to filter the policies:

Tick the check box next to `SystemAdministrator` and then click on **Next: Review**:



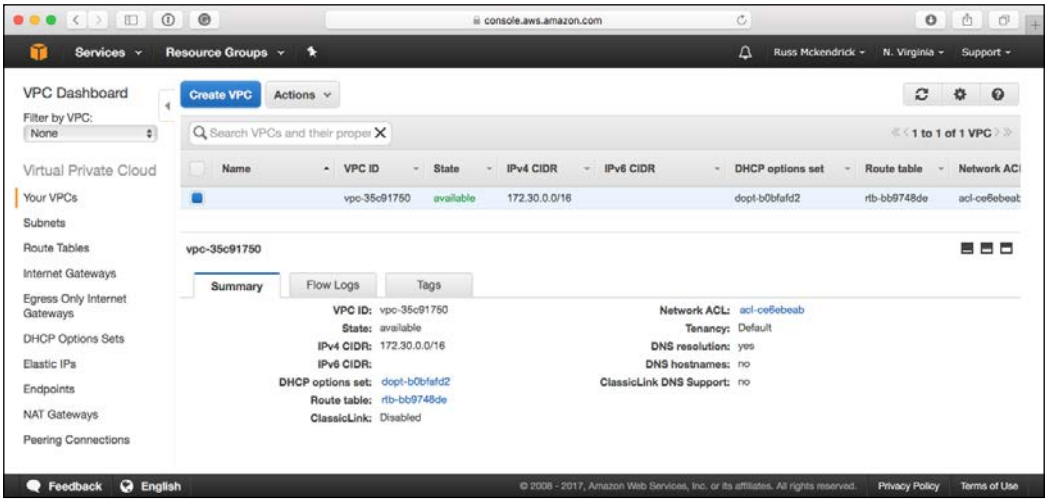
On the review page, click on **Create user** and after a few seconds, you should receive confirmation your user has been successfully created.

Make sure you click on **Download .csv** as you will not be shown the Secret access key again. Now you have your Access key ID and Secret access key.

Before you find your VPC ID, you should make sure you are in the correct region by ensuring that it says, **N. Virginia** in the top-right of your AWS console; if it doesn't, select it from the drop-down list.

Amazon describes Amazon VPC (Amazon Virtual Private Cloud) as letting you provision a logically isolated section of the AWS Cloud where you can launch resources in a virtual network which you define. You have complete control over your virtual networking environment, including the selection of your own IP address range, the creation of subnets, and configuration of route tables and network gateways.

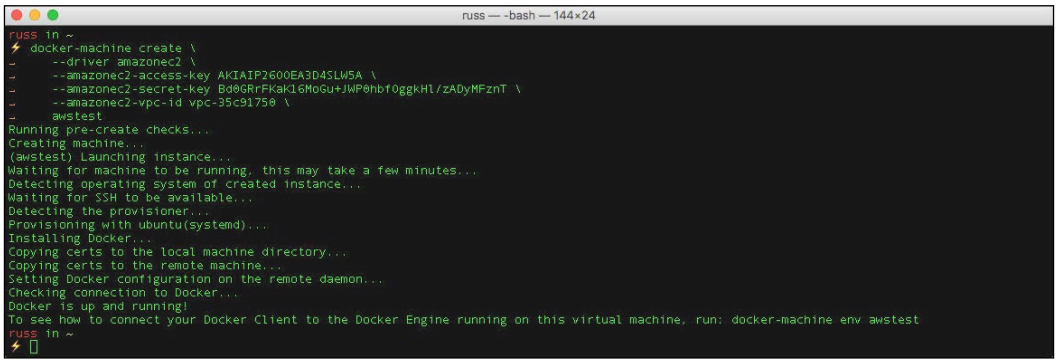
Once you have ensured you are in the correct region, go to **Services** then **VPC** and click on **Your VPCs**; you don't need to worry about creating and configuring a VPC as Amazon provides you with a default VPC in each region. Select the VPC and you should see the something like the following:



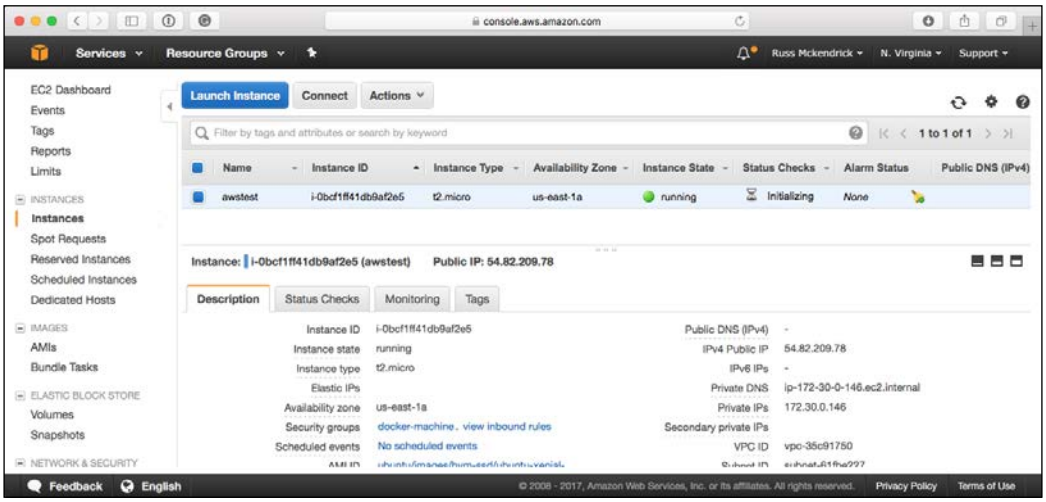
Make a note of the VPC ID; you should now have enough information to launch your instance using Docker Machine. To do this, run the following command:

```
docker-machine create \
    --driver amazonec2 \
    --amazonec2-access-key AKIAIP26OOEA3D4SLW5A \
    --amazonec2-secret-key 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 \
    --amazonec2-vpc-id vpc-35c91750 \
    awstest
```

If all goes well, you should see something like the following output:



You should also be able to see an EC2 instance launched in the AWS Console by clicking on **Services** | **EC2** | **Instances**:



You may have noticed Docker Machine created the security group and assigned an SSH key to the instance without any need for us to get involved, keeping within the principle that you don't need to be an expert in configuring the environments you are launching your Docker instance into.

Before we terminate the instance, let's switch our local Docker client over to use the AWS instance and launch the **Hello World** container by running the following commands:

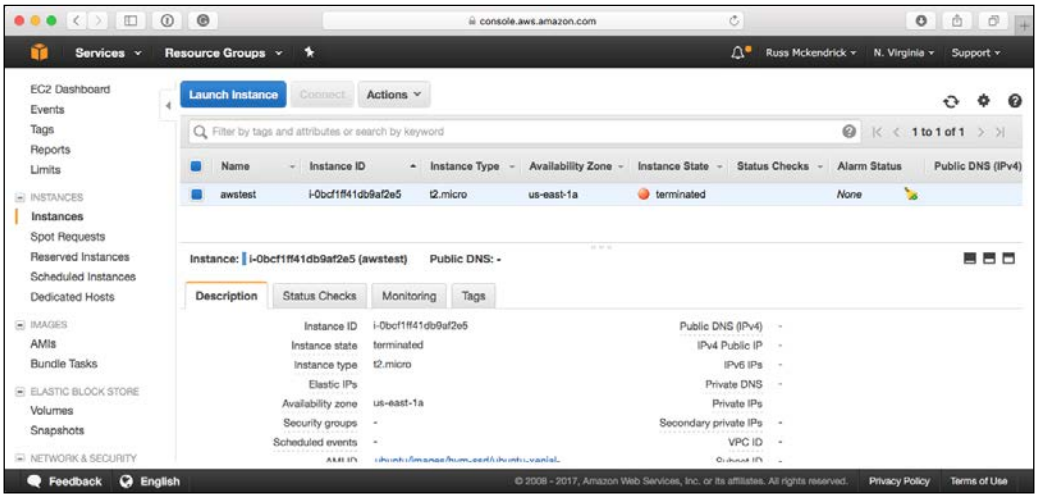
```
eval $(docker-machine env awstest)
docker-machine ls
docker container run hello-world
docker container ls -a
```

As you can see, once you have launched an instance using Docker Machine and switch your local Docker client to it, there is no difference in usage between running Docker locally or on a cloud provider.

Before we start to rack up cost we should terminate our test AWS instance by running the following command:

```
docker-machine rm awstest
```

And then confirm in the AWS console that the instance has terminated correctly:



If you don't do this, the EC2 instance will quite happily sit there costing, you $0.013 per hour until it is terminated.

Please note, this is not Docker for AWS, we will be covering this service in *Chapter 4*, *Docker Swarm*.

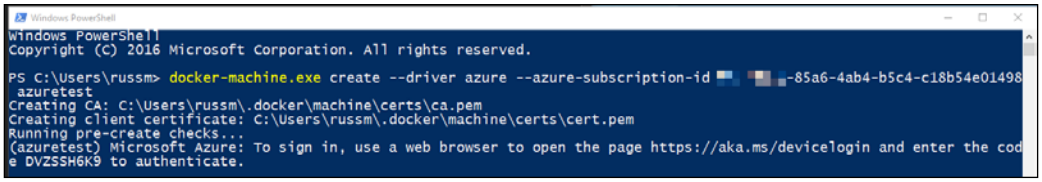
The Microsoft Azure driver

As you may have noticed from the terminal and browser screenshots, so far, we have been using Docker for Mac; let's look at the Microsoft Azure driver using Docker for Windows.

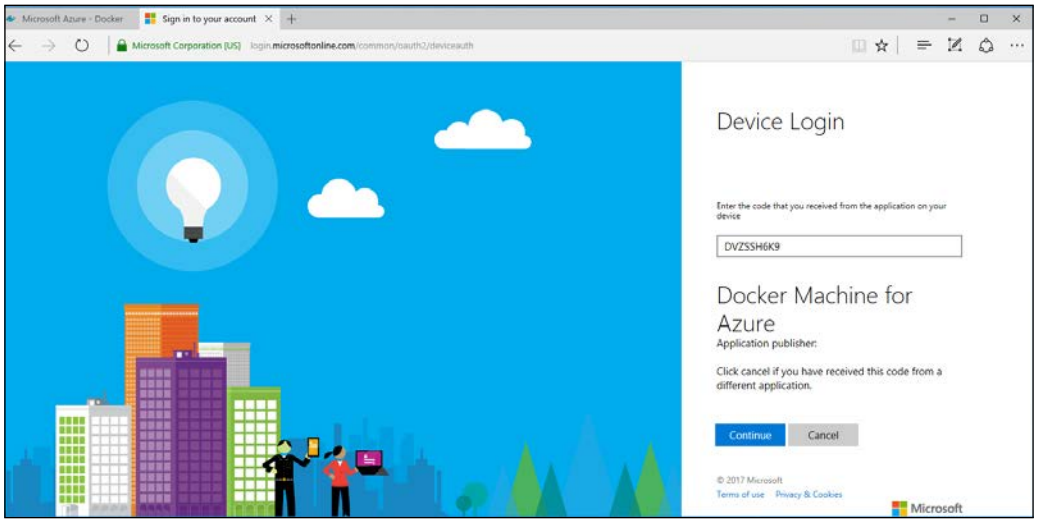
First of all, you will need a Microsoft Azure account; if you don't already have one, you can sign up at `https://azure.microsoft.com/`. Once you have your account, the only piece of information you need to get started is your subscription ID; you can find this in the billing section of the portal.

Once you have your subscription ID, you can authenticate Docker Machine with Azure, to do this enter the following command, making sure to replace the subscription ID with your own:

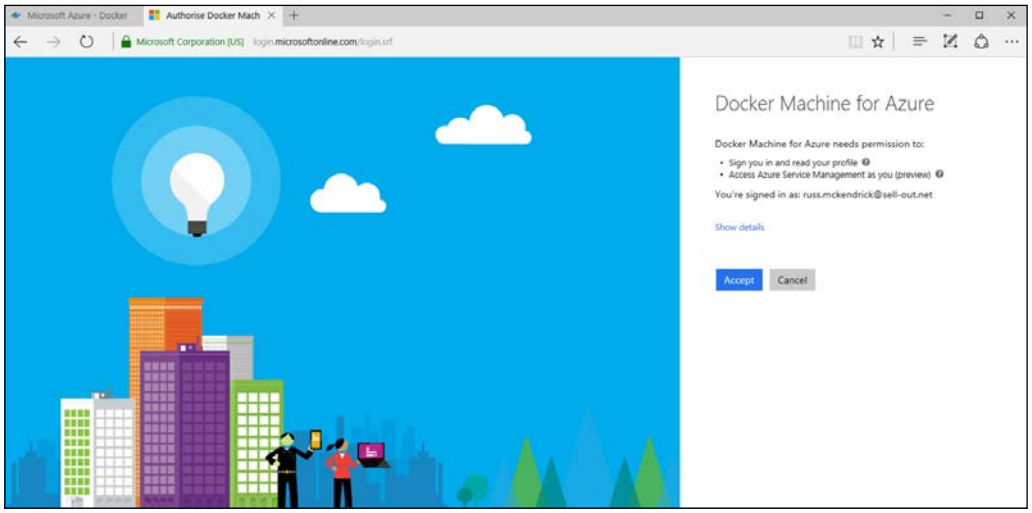
```
docker-machine.exe create --driver azure --azure-subscription-id xxxxxxx-85a6-4ab4-b5c4-c18b54e01498 azuretest
```



You will receive an activation code to authorise Docker Machine; go to `https://aka.ms/devicelogin/` and enter the code you have been given:



Clicking **Continue** will take you a page which then shows you the permissions which Docker Machine is going to grant:



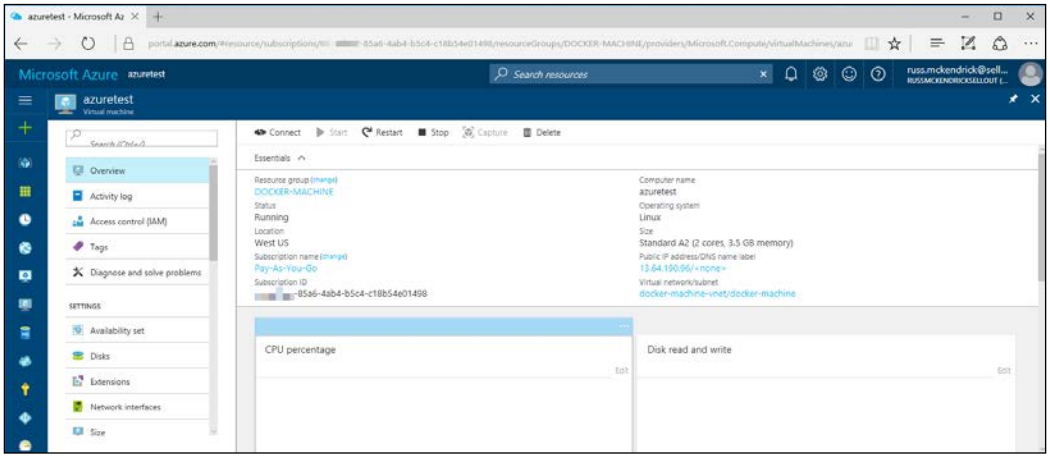
Once you click **Accept,** you should see Docker Machine start bootstrapping the environment. The process will take several minutes; once it completes, you should see something like the following output:

```
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\Users\russm> docker-machine.exe create --driver azure --azure-subscription-id [redacted] -85a6-4ab4-b5c4-c18b54e01498
 azuretest
Creating CA: C:\Users\russm\.docker\machine\certs\ca.pem
Creating client certificate: C:\Users\russm\.docker\machine\certs\cert.pem
Running pre-create checks...
(azuretest) Microsoft Azure: To sign in, use a web browser to open the page https://aka.ms/devicelogin and enter the cod
e DVZSSH6K9 to authenticate.
(azuretest) Registering subscription to resource provider.  ns="Microsoft.Compute" subs=" [redacted] -85a6-4ab4-b5c4-c18b54e
01498"
(azuretest) Registering subscription to resource provider.  ns="Microsoft.Network" subs="[redacted] -85a6-4ab4-b5c4-c18b54e
01498"
(azuretest) Registering subscription to resource provider.  ns="Microsoft.Storage" subs=" [redacted] -85a6-4ab4-b5c4-c18b54e
01498"
(azuretest) Completed machine pre-create checks.
Creating machine...
(azuretest) Querying existing resource group.  name="docker-machine"
(azuretest) Creating resource group.  name="docker-machine" location="westus"
(azuretest) Configuring availability set.  name="docker-machine"
(azuretest) Configuring network security group.  name="azuretest-firewall" location="westus"
(azuretest) Querying if virtual network already exists.  rg="docker-machine" location="westus" name="docker-machine-vnet
"
(azuretest) Creating virtual network.  name="docker-machine-vnet" rg="docker-machine" location="westus"
(azuretest) Configuring subnet.  name="docker-machine" vnet="docker-machine-vnet" cidr="192.168.0.0/16"
(azuretest) Creating public IP address.  name="azuretest-ip" static=false
(azuretest) Creating network interface.  name="azuretest-nic"
(azuretest) Creating storage account.  name="vhdsr5ivfpkvaojdow083rvl" location="westus" sku=Standard_LRS
(azuretest) Creating virtual machine.  name="azuretest" location="westus" size="Standard_A2" username="docker-user" osIm
age="canonical:UbuntuServer:16.04.0-LTS:latest"
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: C:\Program Files\Dock
r\Resources\bin\docker-machine.exe env azuretest
PS C:\Users\russm>
```

As you can see, Docker Machine has done the following:

- Created a resource group
- Configured a network security group
- Configured a network subnet
- Created a virtual network
- Assigned a public IP address
- Created a network interface
- Created a storage account
- Launched a virtual machine

If you go to the resource group within the Azure Portal you should see your virtual machine is launched and ready:



Like the Mac and Linux versions of Docker Machine we need to configure our local Docker client to communicate with the remote host, to do this we need to run the following command:

```
docker-machine.exe env --shell powershell azuretest | Invoke-Expression
```

This is the equivalent of running the following on Mac or Linux:

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

You can check that Azure is now active by running:

```
docker-machine.exe ls
```

Now that we have our client talking to our Azure remote host, we can launch the **Hello World** container by running:

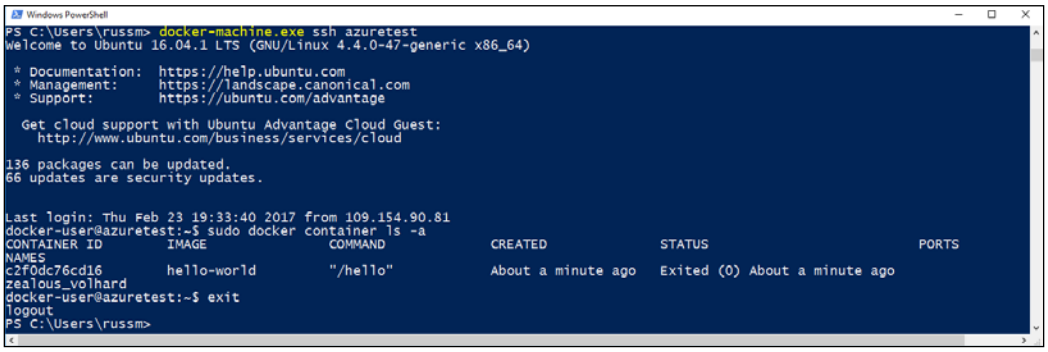
```
docker container run hello-world
```

```
docker container ls -a
```

As with the Mac and Linux version of Docker Machine, you can SSH into your Azure host by running:

```
docker-machine.exe ssh azuretest
```

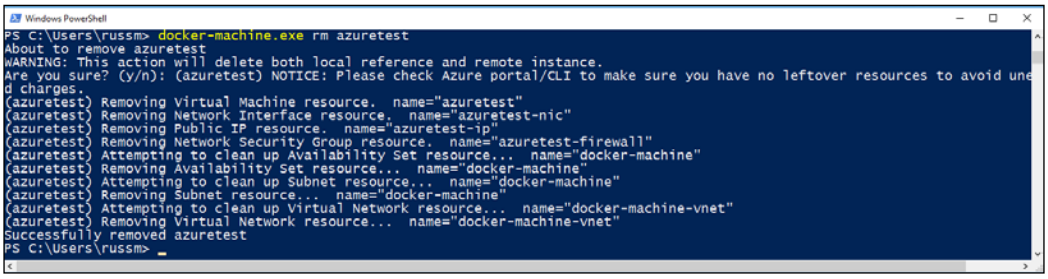
As you can see from the output below, we can see the **Hello World** container:



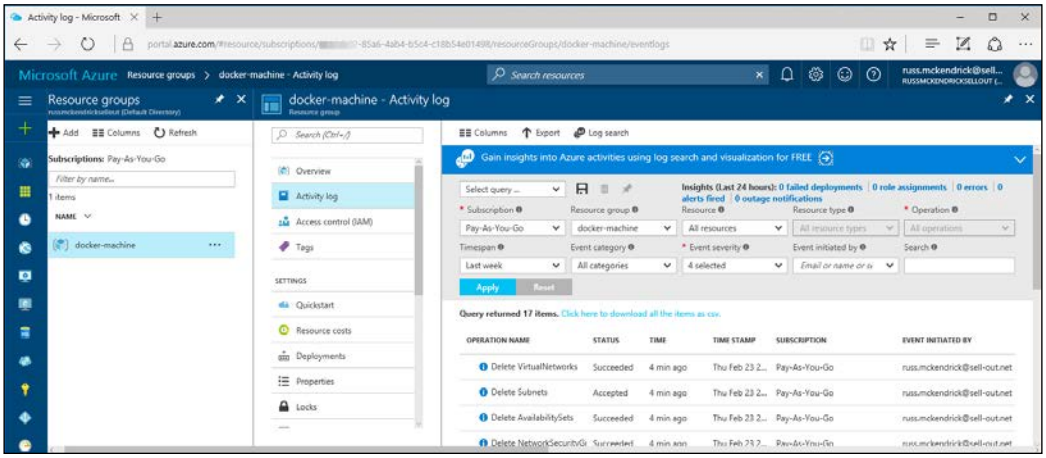
From here you can interact with the Azure host as you would do any other Docker host. Once you are ready to terminate your Azure host, all you need to do is run the following command:

```
docker-machine.exe rm azuretest
```

It will take a short while to remove the host and all the resources associated with it, once complete, you should see something like:



Checking the **Activity log** in the docker-machine **Resource group** using the Azure portal should show you the resources being removed:



As highlighted by the PowerShell output, it is best to check that everything has been properly terminated, the easiest way to do this is to remove the resource group itself, to do this click on the three dots (**...**) on the right-hand side of the **docker-machine** resource.

After you have followed the on-screen prompts, which include typing the name of the resource you are choosing to remove, you should receive confirmation that the resource group has been removed.

While we have used Windows to look at Azure, the process, other than switching the local client to use the remote host, is the same on Mac and Linux.

Please note, this is not Docker for Azure, we will be covering this service in *Chapter 4*, *Docker Swarm*.

References

The examples we have used in this chapter have been launching Ubuntu instances. Docker Machine also supports:

- Debian (8.0+) - `https://www.debian.org/`
- Red Hat Enterprise Linux (7.0+) - `https://www.redhat.com/`
- CentOS (7+) - `https://www.centos.org/`
- Fedora (21+) - `https://getfedora.org/`
- RancherOS (0.3) - `http://rancher.com/rancher-os/`

The other thing to mention about Docker Machine is that by default it operates an opt-in for crash reporting, considering the number of different configuration / environment combinations Docker Machine can be used with it is important that Docker gets notified of any problems to help them make a better product.

If for any reason, you want to opt out, then running the following command will disable crash reporting:

```
mkdir -p ~/.docker/machine && touch ~/.docker/machine/no-error-report
```

For more information on Docker Machine you can see the official documentation:

- Docker Machine - `https://docs.docker.com/machine/`
- Docker Machine Drivers - `https://docs.docker.com/machine/drivers/`
- Docker Machine Command Reference - `https://docs.docker.com/machine/reference/`

Summary

As you can see from examples we have worked through; Docker Machine is a powerful tool as it allows users of all skill levels to be able to launch an instance in a cloud provider without having to roll their sleeves up and get stuck in configuring server instances.

In our next chapter, we are going to look at launching multiple Docker hosts in the same cloud providers and then configuring a Docker Swarm cluster.

4
Docker Swarm

So far we have learned how to launch individual Docker hosts locally using Docker for Mac, Docker for Windows, and Docker Machine for remote hosts, as well as using Docker locally on Linux. Individual Docker hosts are great for local development, or launching a few test instances however as you start moving towards production you need fewer single points of failure.

In this chapter, we are going to get a little more adventurous and create a cluster of Docker hosts. Docker ships a tool called Swarm, when deployed it acts as a scheduler between your Docker client and the Docker host, deciding where to launch containers based on scheduling rules.

We are going to look at the following topics:

- Manually launching a Docker Swarm cluster
- Launching Docker for Amazon Web Services
- Launching Docker for Azure

And also how to launch containers within our cluster.

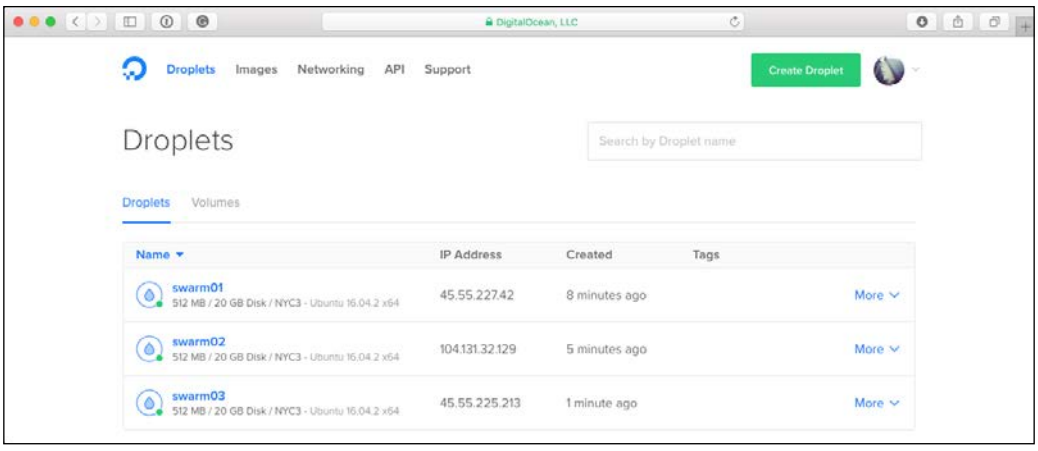
Creating a Swarm manually

At the start of *Chapter 3*, *Docker in the Cloud* we looked at using a Docker Machine to launch a Docker host in Digital Ocean. We are going to start with Digital Ocean again, but this time we are going to launch three hosts and then create a Docker Swarm cluster on them.

To start off with we need to launch the hosts and to do this, run the following commands, remembering to replace the Digital Ocean API access token with your own:

```
docker-machine create \
    --driver digitalocean \
    --digitalocean-access-token
57e4aeaff8d7d1a8a8e46132969c2149117081536d50741191c79d8bc083ae73 \
    swarm01

docker-machine create \
    --driver digitalocean \
    --digitalocean-access-token
57e4aeaff8d7d1a8a8e46132969c2149117081536d50741191c79d8bc083ae73 \
    swarm02

docker-machine create \
    --driver digitalocean \
    --digitalocean-access-token
57e4aeaff8d7d1a8a8e46132969c2149117081536d50741191c79d8bc083ae73 \
    swarm03
```

Once launched, running `docker-machine ls` should show you a list of your images. Also, this should be reflected in your Digital Ocean control panel:



Now we have our Docker hosts and we need to assign a role to each of the nodes within the cluster. Docker Swarm has two node roles:

- **Manager**: A manager is a node which dispatches tasks to the workers, all your interaction with the Swarm cluster will be targeted against a manager node. You can have more than one Manger node, however in this example we will be using just one.
- **Worker**: Worker nodes accept the tasks dispatched by the Manager node(s), these are where all your services are launched. We will go in to services in more detail once we have our cluster configured.

In our cluster, **swarm01** will be the manager node with **swarm02** and **swarm03** being our two worker nodes. We are going to use the `docker-machine ssh` command to execute commands directly on our three nodes, starting with configuring our manager node.

Please note, the commands in the walk through will only work with Mac and Linux, commands to run on Windows will be covered at the end of this section.

Before we initialize the manager node, we need to capture the IP address of `swarm01` as a command-line variable:

```
managerIP=$(docker-machine ip swarm01)
```

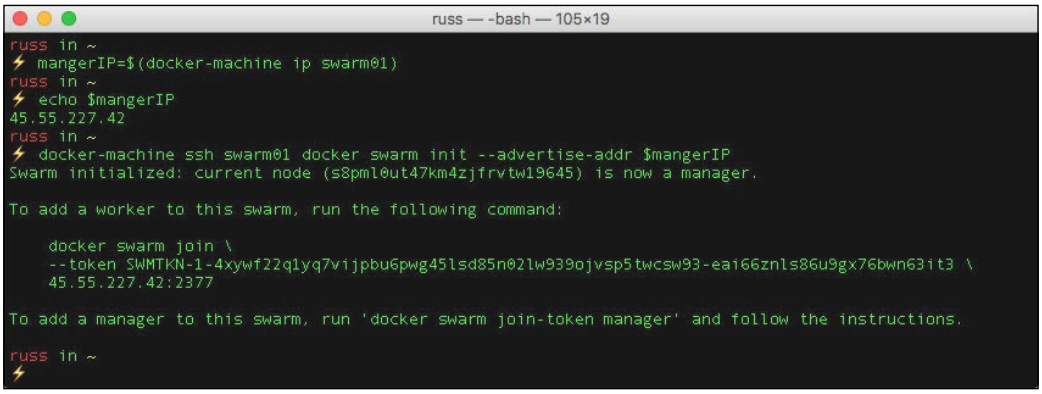
Now that we have the IP address, run the following command to check if it is correct:

```
echo $managerIP
```

And then to configure the manager node, run the following command:

```
docker-machine ssh swarm01 docker swarm init --advertise-addr $managerIP
```

You will then receive confirmation that `swarm01` is now a manager along with instructions on what to run to add a worker to the cluster:



You don't have to a make a note of the instructions as we will be running the command in a slightly different way.

To add our two workers, we need to capture the join token in a similar way we captured the IP address of our manager node using the `$managerIP` variable; to do this, run:

```
joinToken=$(docker-machine ssh swarm01 docker swarm join-token -q worker)
```

Again, you `echo` the variable out to check that it is valid:

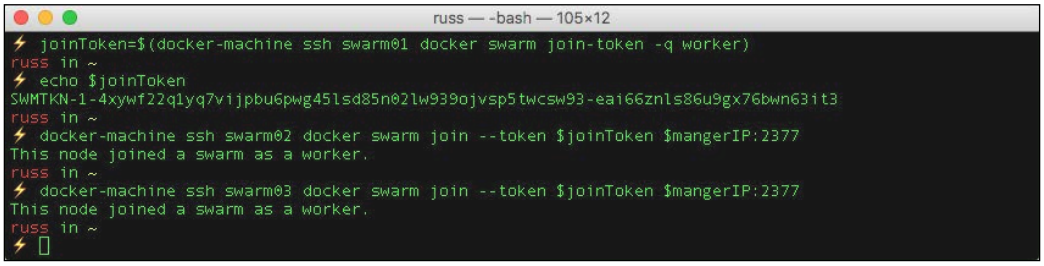
```
echo $joinToken
```

Now it's time to add our two worker nodes into the cluster by running:

```
docker-machine ssh swarm02 docker swarm join --token $joinToken
$managerIP:2377
```

```
docker-machine ssh swarm03 docker swarm join --token $joinToken
$managerIP:2377
```

You should see something like the following terminal output:



Connect your local Docker client to the manager node using the following:

```
eval $(docker-machine env swarm01)
```

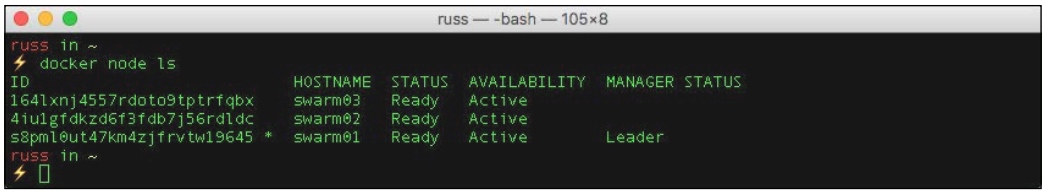
And then running a `docker-machine ls` again shows. As you can see from the list of hosts, `swarm01` is now active but there is nothing in the **SWARM** column; why is that?

Confusingly, there are two different types of Docker Swarm cluster, there is the Legacy Docker Swarm which was managed by Docker Machine, and then there is the new Docker Swarm mode which is managed by the Docker Engine itself.

We have a launched a Docker Swarm Mode cluster. This is now the preferred way of launching Swarm, the legacy Docker Swarm is slowly being retired.

To get a list of the nodes within our Swarm cluster, we need to run the following command:

```
docker node ls
```



For information on each node you can run the following command (the `--pretty` flag renders the JSON output from the Docker API):

```
docker node inspect swarm01 --pretty
```

You are given a wealth of information about the host, including the fact that it is a manager and it has been launched in Digital Ocean. Running the same command; but for a worker node shows similar information:

```
docker node inspect swarm02 --pretty
```

However, as the node is not a manager that section is missing.

Before we look at launching services into our cluster, we should look at how to launch our cluster using Docker Machine on Windows. We will be using PowerShell for this rather than the more traditional Windows CMD prompt, however, even using PowerShell there are a few differences in the commands used due differences between PowerShell and bash.

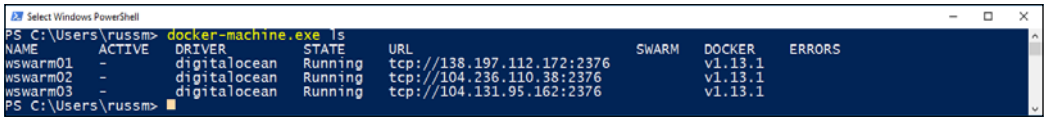
First, we need to launch the three hosts:

```
docker-machine.exe create --driver digitalocean --digitalocean-access-token 57e4aeaff8d7d1a8a8e46132969c2149117081536d50741191c79d8bc083ae73 swarm01
```

```
docker-machine.exe create --driver digitalocean --digitalocean-access-token 57e4aeaff8d7d1a8a8e46132969c2149117081536d50741191c79d8bc083ae73 swarm02
```

```
docker-machine.exe create --driver digitalocean --digitalocean-access-token 57e4aeaff8d7d1a8a8e46132969c2149117081536d50741191c79d8bc083ae73 swarm03
```

Once the three hosts are up and running:

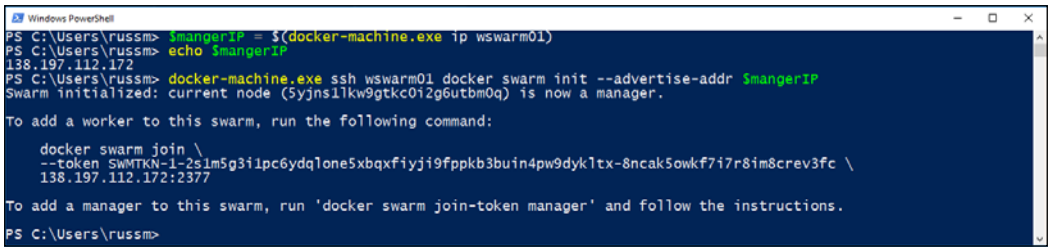


You can create the manager node by running:

```
$managerIP = $(docker-machine.exe ip swarm01)
```

```
echo $managerIP
```

```
docker-machine.exe ssh swarm01 docker swarm init --advertise-addr $managerIP
```



Once you have your manager you can add the two worker nodes:

```
$joinIP = "$(docker-machine.exe ip swarm01):2377"
```

```
echo $joinIP
```

```
$joinToken = $(docker-machine.exe ssh swarm01 docker swarm join-token -q worker)
```

```
echo $joinToken
docker-machine.exe ssh swarm02 docker swarm join --token $joinToken
$joinIP
docker-machine.exe ssh swarm03 docker swarm join --token $joinToken
$joinIP
```

And then configure your local Docker client to use your manager node and check the cluster status:

```
docker-machine.exe env  swarm01 | Invoke-Expression
docker-machine.exe ls
docker node ls
```

At this stage, no matter which operating system you are using, you should have a three node Docker Swarm cluster in Digital Ocean, we can now look at a launching service into our cluster.

Launching a service

Rather than launching containers using the `docker container run` command you need to create a service A service defines a task which the manager then passes to one of the workers and then a container is launched.

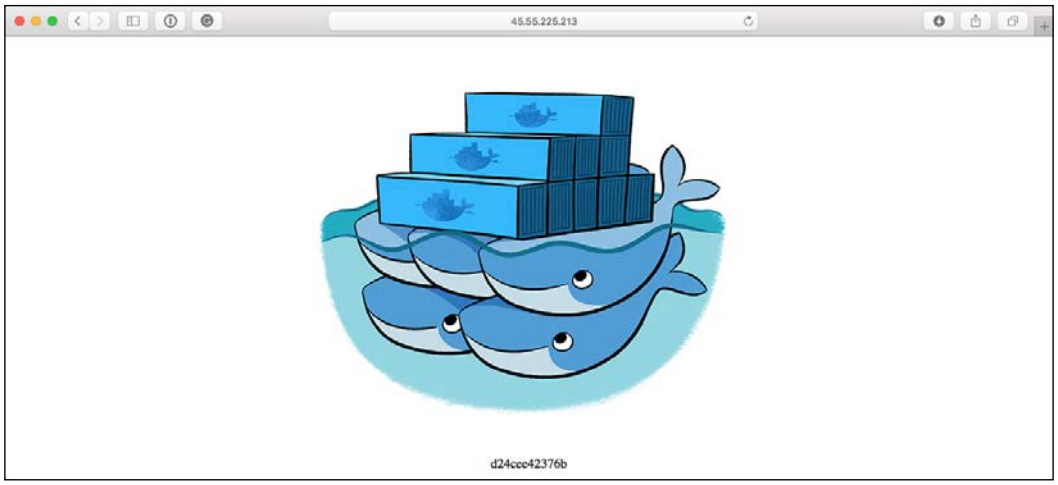
Let's launch a simple service called cluster which uses the image we looked at in *Chapter 2*, *Launching Applications Using Docker*:

```
docker service create --name cluster -p:80:80/tcp russmckendrick/cluster
```

That's it, we should now have a single container running on one of our three nodes. To check that the service is running and get a little more information about the service, run the following commands:

```
docker service ls
docker service inspect cluster --pretty
```

Now that we have confirmed that our service is running, you will be able to open your browser and enter the IP address of one of your three nodes (which you can get by running `docker-machine ls`).One of the features of Docker Swarm is it's routing mesh.

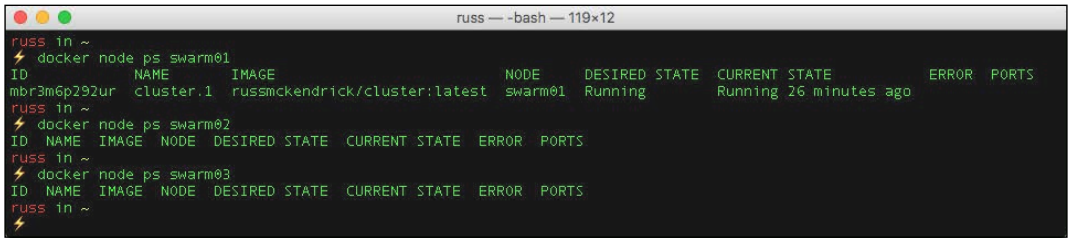


A routing mesh? When we exposed the port using the `-p:80:80/tcp` flag, we did a little more than map port `80` on the host to port `80` on the container, we actually created a Swarm load balancer on `port 80` across all of the hosts within the cluster. The Swarm load balancer then directs requests to containers within our cluster.

Running the commands below, should show you which tasks are running on which nodes, remember tasks are containers which have been launched by the service:

```
docker node ps swarm01
docker node ps swarm02
docker node ps swarm03
```

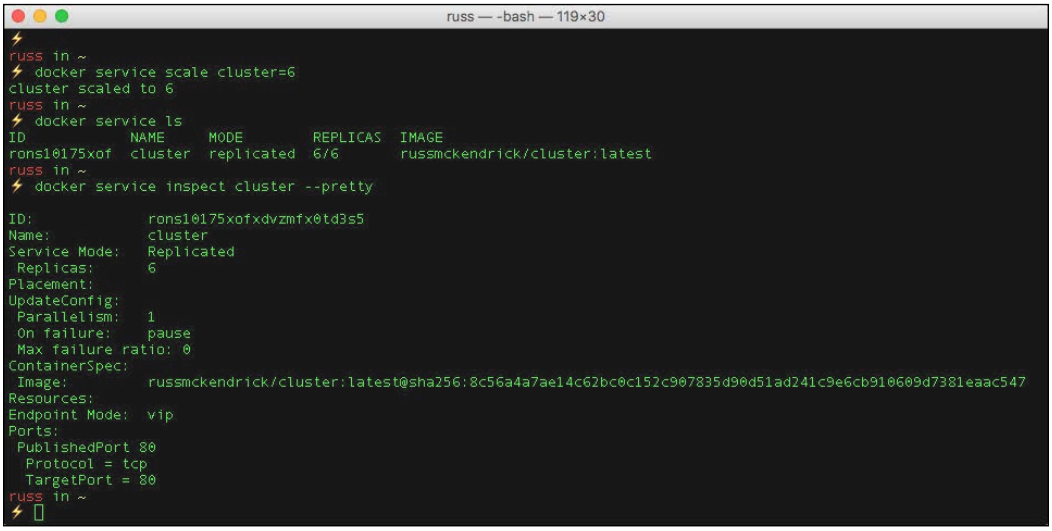
Like me, you probably have your single task running on `swarm01`:



We can make things more interesting by scaling our service to add more tasks, to do this simply run the following commands to scale and check our service:

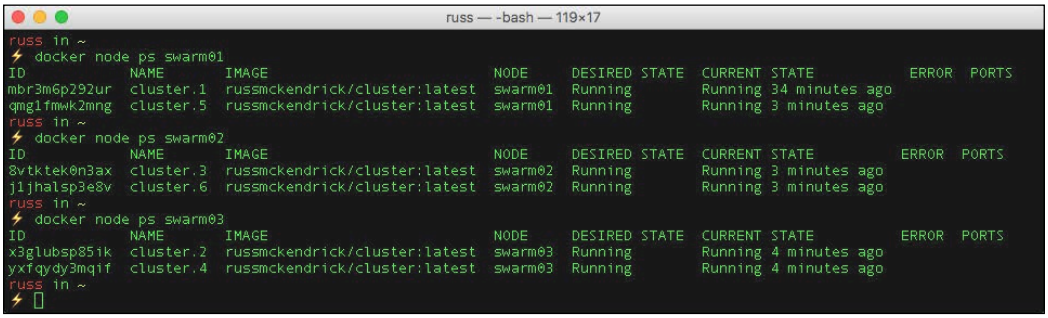
```
docker service scale cluster=6
docker service ls
docker service inspect cluster --pretty
```

As you should see, we now have 6 tasks running within our cluster service:



Checking the nodes should show that the tasks are evenly distributed between our three nodes:

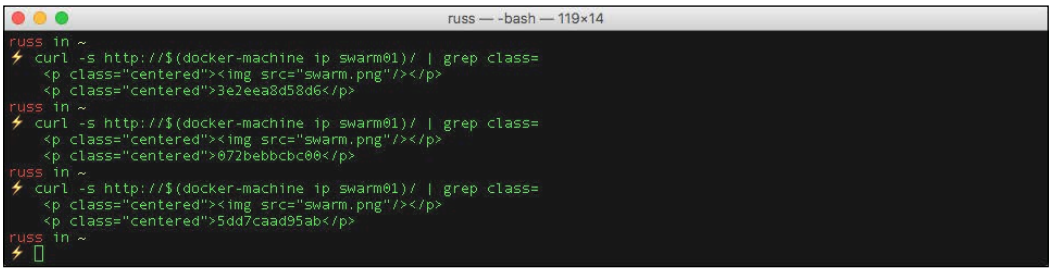
```
docker node ps swarm01
docker node ps swarm02
docker node ps swarm03
```



Hitting refresh in your browser should also update the hostname under the Docker image change, another way of seeing this on Mac and Linux is to run the following command:

```
curl -s http://$(docker-machine ip swarm01)/ | grep class=
```

As you can see from the following terminal output, our requests are being load balanced between the running tasks:



Before we terminate our Docker Swarm cluster let's look at another way we can launch services, before we do we need to remove the currently running service, to do this simply run:

```
docker service rm cluster
```

Now that the service has been removed, we can launch a stack.

Launching a stack

This is where it may get confusing. If a service is the same as running container then a stack is running a collection of services like you would launch multiple containers using Docker Compose. In fact, you can launch a stack using a Docker Compose file, with a few additions.

Let's look at launching our Cluster application again. You can find the Docker Compose file we are going to be using in the repo in the `/bootcamp/chapter04/cluster/` folder, before we go through the contents of the `docker-compose.yml` file, let's launch the stack. To do this run the following command:

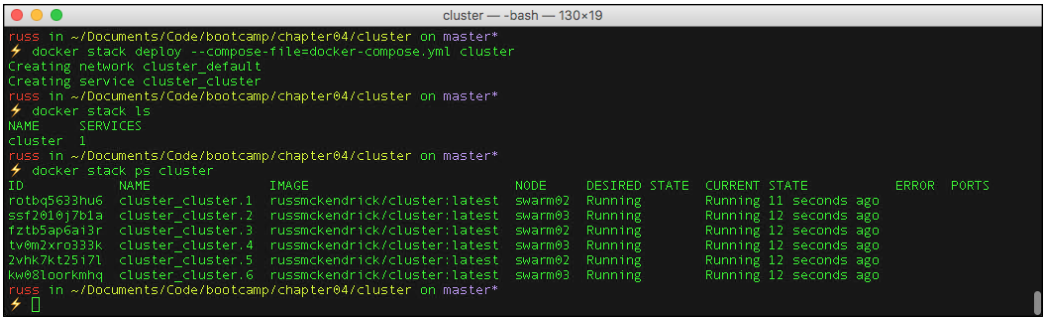
```
docker stack deploy --compose-file=docker-compose.yml cluster
```

You should get confirmation that the network for the stack has been created along with the service. You can list the services launched by the stack by running:

```
docker stack ls
```

And then check on the tasks within the service by running:

```
docker stack ps cluster
```



You may be surprised to see that service has launched its tasks on `swarm02` and `swarm03` only. For an explanation as to why, let's open the `docker-compose.yml` file:

```
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 `docker-compose.yml` file looks like what we covered in *Chapter 2*, *Launching Applications Using Docker*, until we get to the `deploy` section.

You may have already spotted the reason why we only have tasks running on our two worker nodes, as you can see in the `placement` section, we have told Docker to only launch our tasks on nodes with the role of worker.

Next up we have a defined a `restart_policy` this tells the Docker what to do should any of the tasks stop responding, in our case we are telling the Docker to restart them `on-failure`. Finally, we are telling the Docker to launch six `replicas` within our service.

Let's test that restart policy by terminating one of our two worker nodes. There is a graceful way of doing this by draining the node, however, it more fun to just terminate the node, to do this run the following command:

```
docker-machine rm swarm03
```

Running `docker stack ps` cluster immediately after removing the host shows that the Docker hasn't caught up yet.

Running `docker stack ps` a few seconds later will show that we still have six tasks running, but as you can see from the terminal output they are now all running on `swarm02` and the tasks the new ones have replaced are showing as **shutdown**.

Our application should still be available by entering the IP address of `swarm01` or `swarm02` into your browser. Once you have finished with the remain two hosts you can them by running:

```
docker-machine rm swarm01 swarm02
```

So far, we have manually created our Docker Swarm cluster in Digital Ocean, I am sure you agree that so far, the process has been straightforward, especially considering how powerful the clustering technology is, you are already probably starting to think how you can start to deploy services and stacks.

In the next few sections we are going to look at Docker for Amazon Web Services and Docker for Azure, and how Docker can take advantage of the range of supporting features provided by the two public cloud services.

Docker for Amazon Web Services

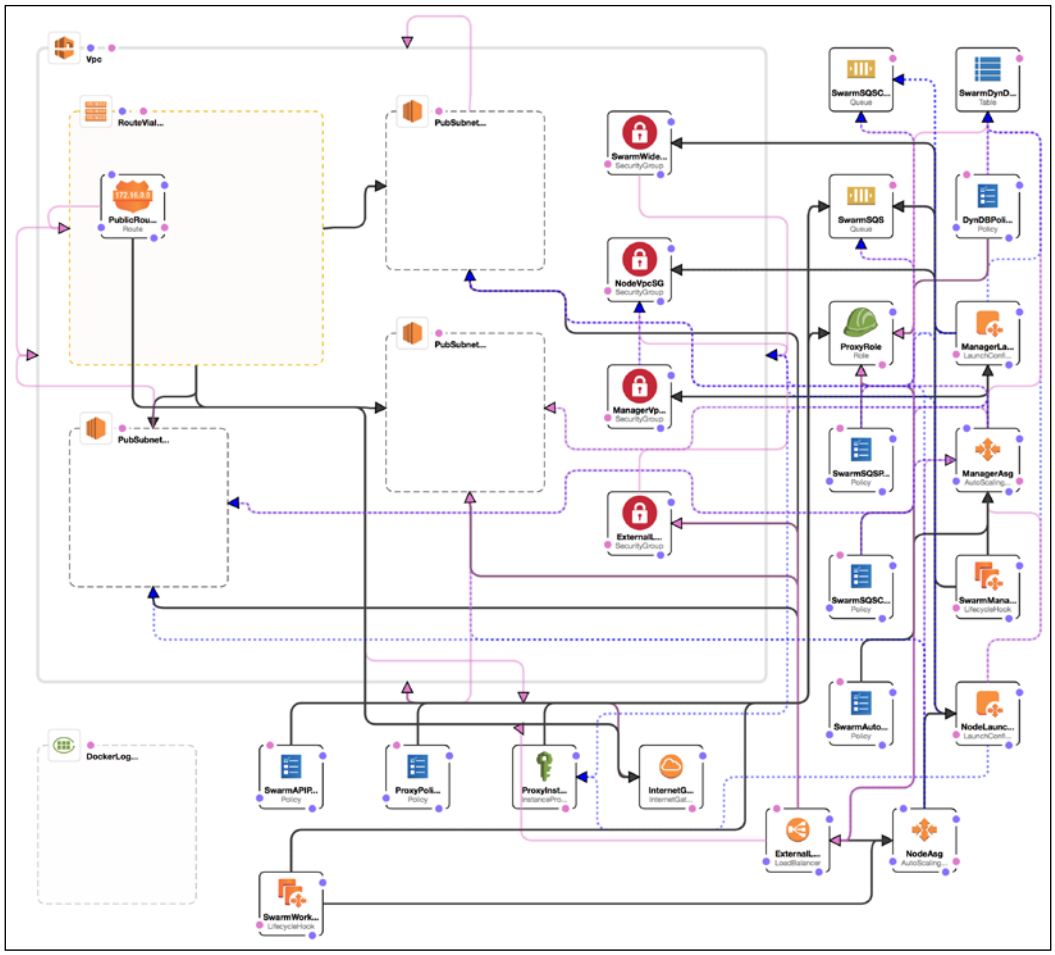
Docker for AWS is a Swarm cluster which has been tuned by Docker to run in Amazon Web Services.

AWS CloudFormation is a templating engine which allows you to define your AWS infrastructure and resources in a controllable and predictable fashion.

The AWS CloudFormation template can be found at:

`https://editions-us-east-1.s3.amazonaws.com/aws/stable/Docker.tmpl`

As you can see there is quite a lot to it, the image below is a visualization of the template above – while you may not be able to see all the content in the image you should get an idea of the complexity of the **CloudFormation** template supplied by Docker.



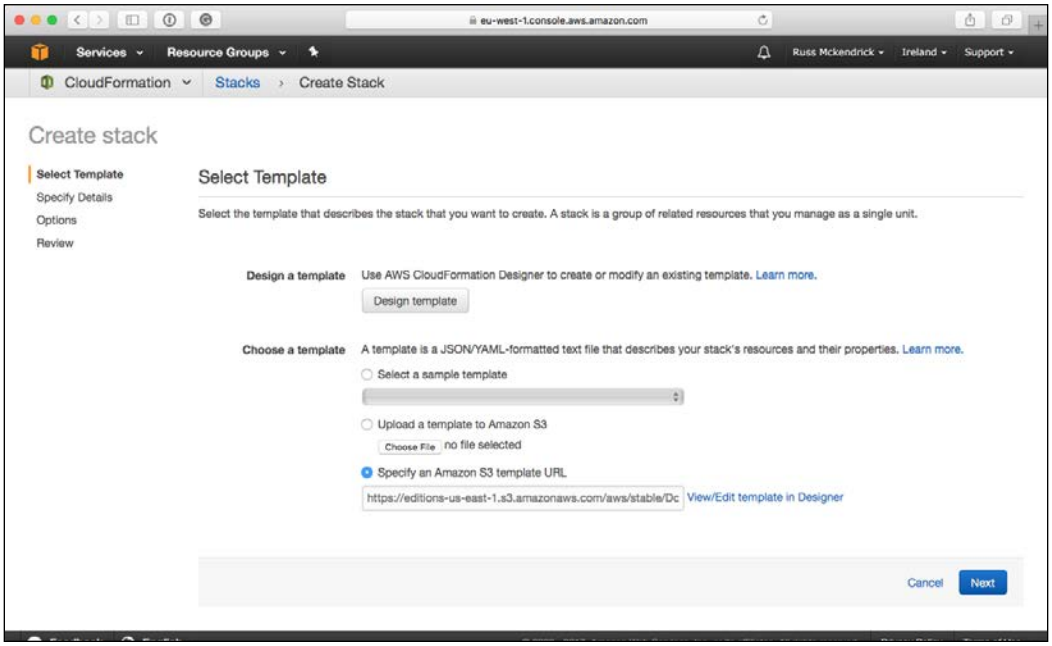
As you can see, the template does all the heavy lifting for you meaning you don't really have to do much apart from one thing, create an SSH key. To do this login to the AWS console at `https://console.aws.amazon.com/`, select **EC2**from the **Services** menu at the top of the screen, once the EC2 dashboard opens click on **Key Pairs** in the left-hand side menu.

Here you will have the option to **Create Key Pair** or **Import Key Pair**. Once you have your SSH key created or imported you can get to launching your Docker for Amazon Web Services cluster, to this, select **CloudFormation** from the **Services** menu.

Clicking **Create New Stack** will take you a page which lets you define your stack, as Docker have already done this for us all you need to do is enter the URL of the stack definition file:

```
https://editions-us-east-1.s3.amazonaws.com/aws/stable/Docker.tmpl
```

In the space below where is says **Specify an Amazon S3 template URL**, making sure that the radio icon above where you entered the URL is selected click on **Next**:



The next page you are taken to is where you define how you would like your stack to look, for this quick demonstration I used the following to roughly match the sizes of the manual Swarm cluster we launched in Digital Ocean:

- **Stack name**: Bootcamp
- **Number of Swarm managers?** 1
- **Number of Swarm worker nodes?** 3
- **Which SSH key to use?**<your own SSH key>
- **Enable daily resource cleanup?** No

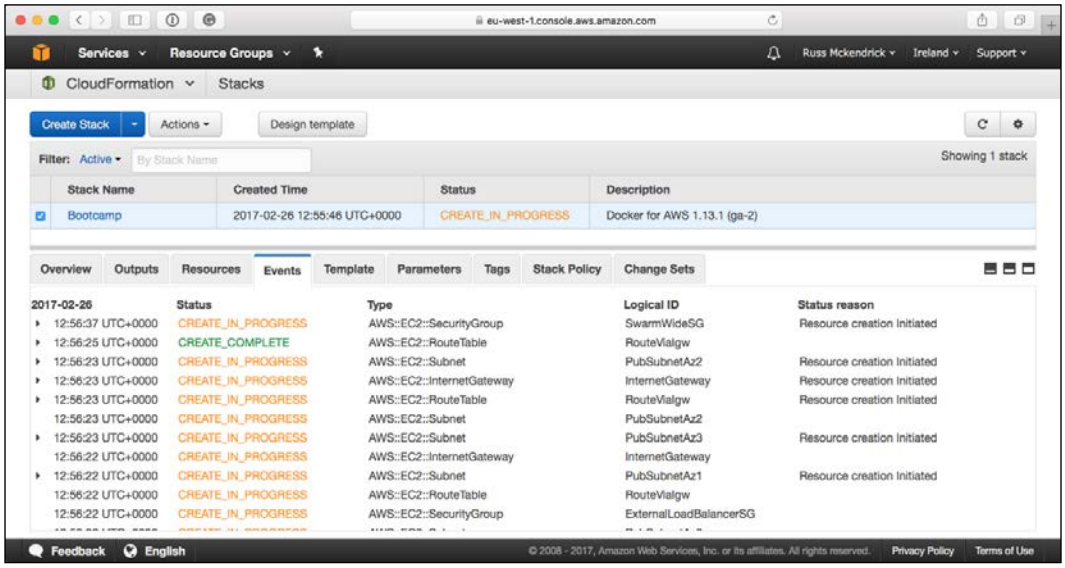
- **Use Cloudwatch for container logging?**Yes
- **Swarm manager instance type?**t2.micro
- **Manager ephemeral storage volume size?** 20
- **Manager ephemeral storage volume type** standard
- **Agent worker instance type?** t2.micro
- **Worker ephemeral storage volume size?** 20
- **Worker ephemeral storage volume type**: standard

Once you have filled in all the details, click on the **Next** button at the bottom of the page. The next screen you are taken to contains additional options such as tagging, we don't need to enter anything here so just click on the **Next** button,

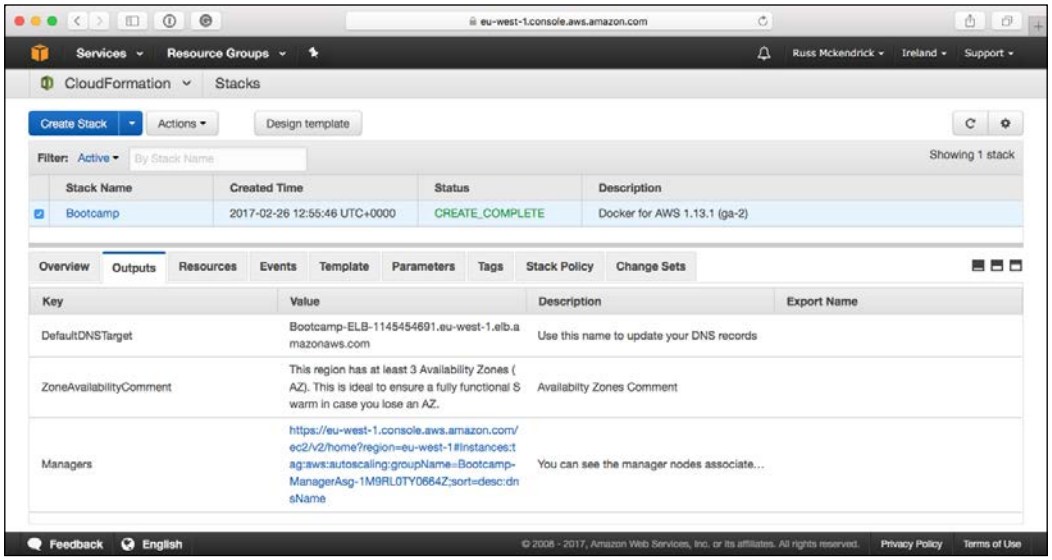
The final page is where we review everything before we comit to launching our stack. If you need to change any of the values you can do so by clicking on **Previous**, once you are happy with how the details you need to tick the box which says, **I acknowledge that AWS CloudFormation might create IAM resources** and then click the **Create** button.

This will immediately start deploying the resources for your Docker for Amazon Web Service cluster, you can check the status of the deployment by having the **Events** tab open.

Clicking the **refresh** button should show you something like the following screen:

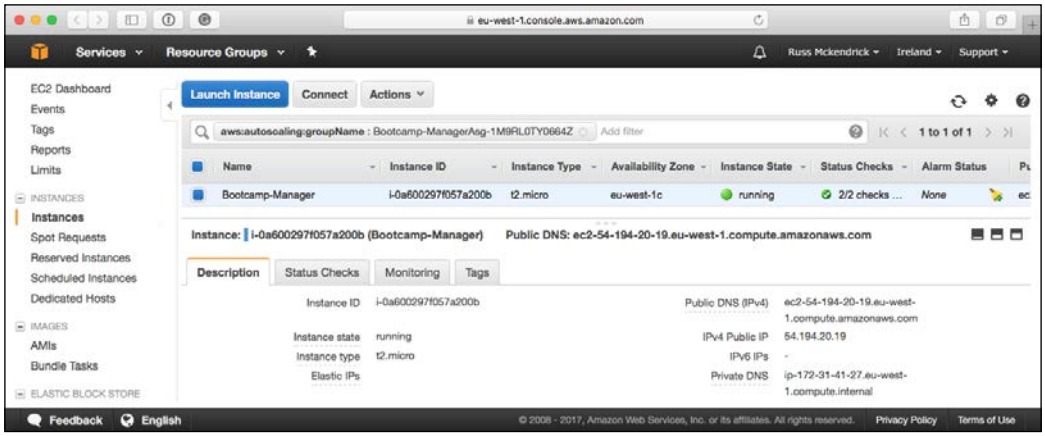


Launching the stack will take several minutes, once it has completed you should see that the **Status** says **CREATE_COMPLETE**. Once you see this, click on the **Outputs** tab:



Here you should see four messages, the first contains the Elastic Load Balancer URL, the second is a message about the availability of your instances and finally you should see a message about **Managers**, this contains a link – click it.

This takes you to the Instances page of the EC2 dashboard, you will also notice that our single manager node has been filtered, selecting it shows information such as the public URL and IP address of the instance:



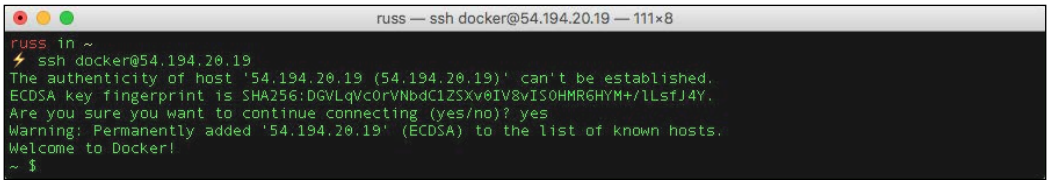
To interact with our cluster, we are going to SSH into the manager node, you need to use the `docker` username. I used the following command:

```
ssh docker@54.194.20.19
```

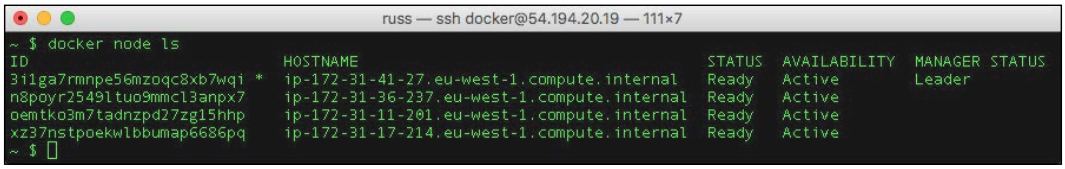
If you downloaded a key pair then you would use something like;

```
ssh docker@54.194.20.19 -I ~/path/to/keypair.pem
```

Once you are logged in you should see something like:



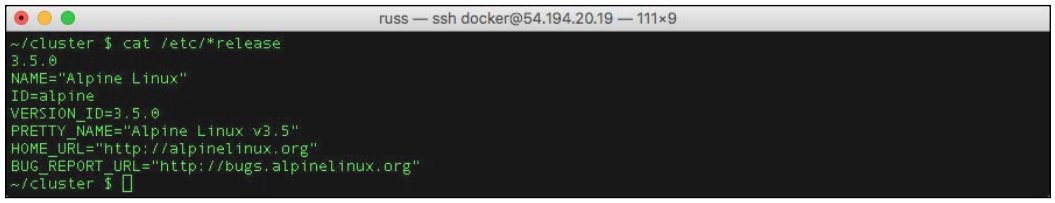
Running `docker node ls` shows that we have three worker nodes and the one manager node we are logged into:



Now let's launch our cluster application, as we are logged into a very basic operating system, in fact as you can from the output of running:

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

We are logged into an Alpine Linux server:



Git is not installed by default so let's install it by switching to the root user and install the Git package using APK:

```
sudo su -
apk update
apk add git
```

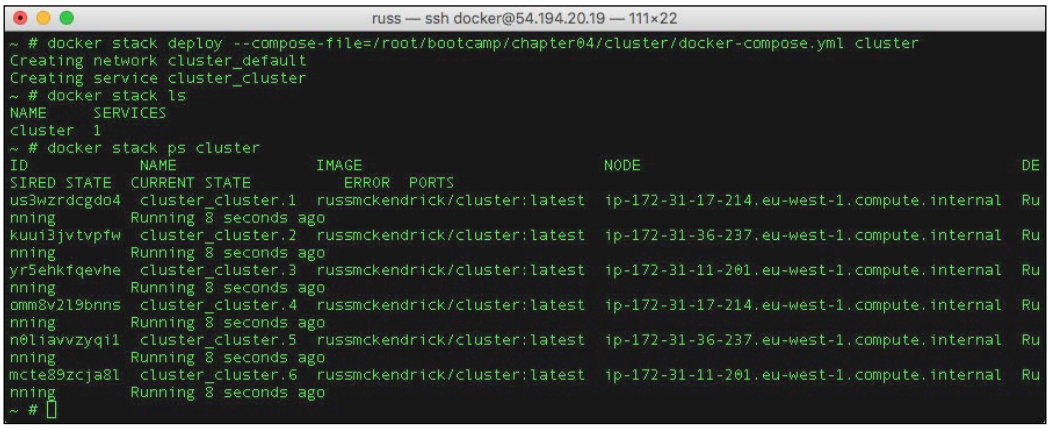
Now that Git is installed we can clone the Bootcamp repo:

```
git clone https://github.com/russmckendrick/bootcamp.git
```

Once Git is installed we can then launch our stack using the following command:

```
docker stack deploy --compose-file=/root/bootcamp/chapter04/cluster/
docker-compose.yml cluster
docker stack ls
docker stack ps cluster
```

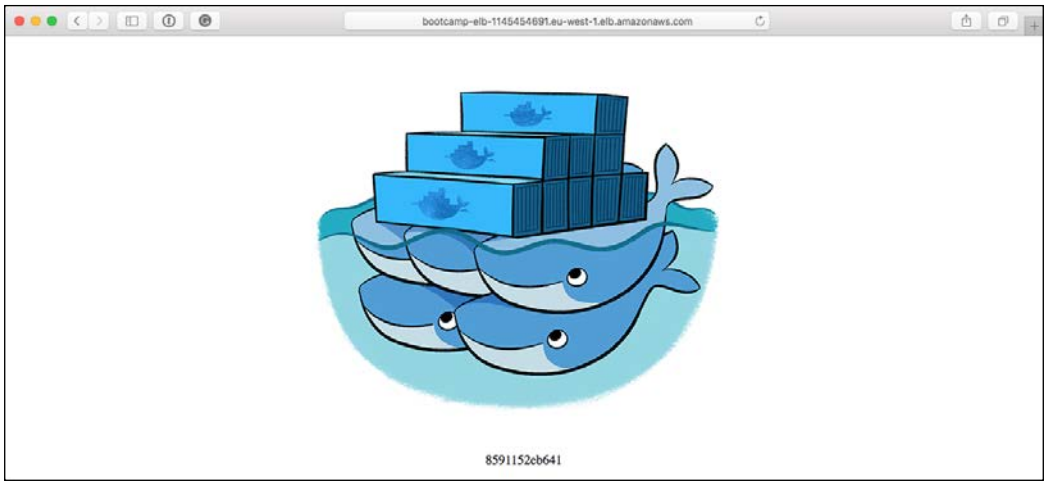
You should see something like the following output:



Now that our stack is launched you can access it using the Elastic Load Balancer URL from the Outputs tab of the CloudFormation stack, in my case the URL was (please note that my URL no longer works):

```
http://bootcamp-elb-1145454691.eu-west-1.elb.amazonaws.com/
```

As you can see from the screen below the page displays as expected with the host name of the container the content is being served from:

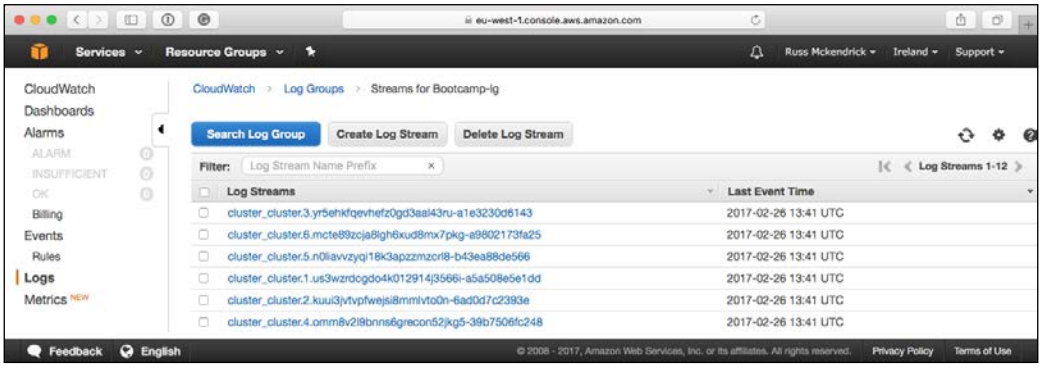


As before, running curl against the Elastic Load Balancer URL shows that hostname of the container is changing (remember to replace the URL with your own):

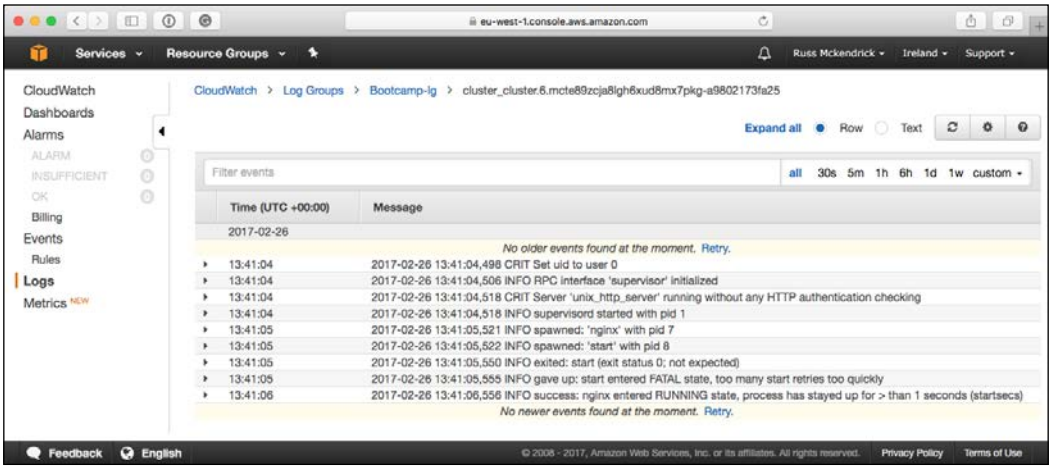
```
curl -s http://bootcamp-elb-1145454691.eu-west-1.elb.amazonaws.com/ |
grep class=
```

Before we teardown our Cluster there is one more to take a quick look at, if you when we launched our Docker for Amazon Web Service stack we said yes to **Use Cloudwatch for container logging**.

This option streams your container logs to Amazons own central logging service, to view return to the AWS console and select **Cloudwatch** from the **Services** menu, once the Cloudwatch dashboard has loaded, click **Logs** in the left-hand side menu and then click on the **Bootcamp-lg** link, here you should list of the containers which were launched by your `docker stack create` command:



Clicking on one of the log streams will show you everything which that container has logged, which in our case should just be a lot of information from the supervisord process:



To tear down our Docker for Amazon Web Services cluster return to the CloudFormation dashboard, select your stack then select **Delete Stack** from the **Actions** menu. This will pop-up a prompt, click the **Yes, Delete** button and deletion of your stack with start immediately.

Removing all the resource will take several minutes, it is important to ensure that all the resources are removed as Amazon operate a pay-as-go model meaning if a resource such as an EC2 instance is running you will be being charged for it so I would recommend you keep the window open and ensure that the deletion is successful.

Speaking of charges, you may have noticed that when we launched our stack there was a link to estimated costs, this takes all the resource defined in the CloudFormation template and runs it through Amazon's Simple Cost Calculator application, our four instance Docker for Amazon Web Services would cost us an estimated $66.98 per month to run.

As you can see, we launched a quite complex configuration without much effort at all, Docker have also applied this same methodology to Microsoft Azure, let's look at that now.

Docker for Azure

Docker for Azure needs a little more work up-front before we can deploy. Luckily, Docker have made this as simple as possible by providing the Azure command line interface as a container.

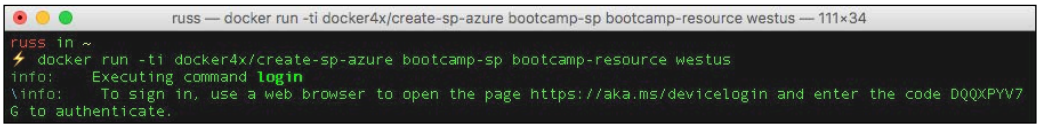
We need to create a service profile and resource group for our deployment to use, to do this simply run the following command:

```
docker run -ti docker4x/create-sp-azure bootcamp-sp bootcamp-resource
westus
```

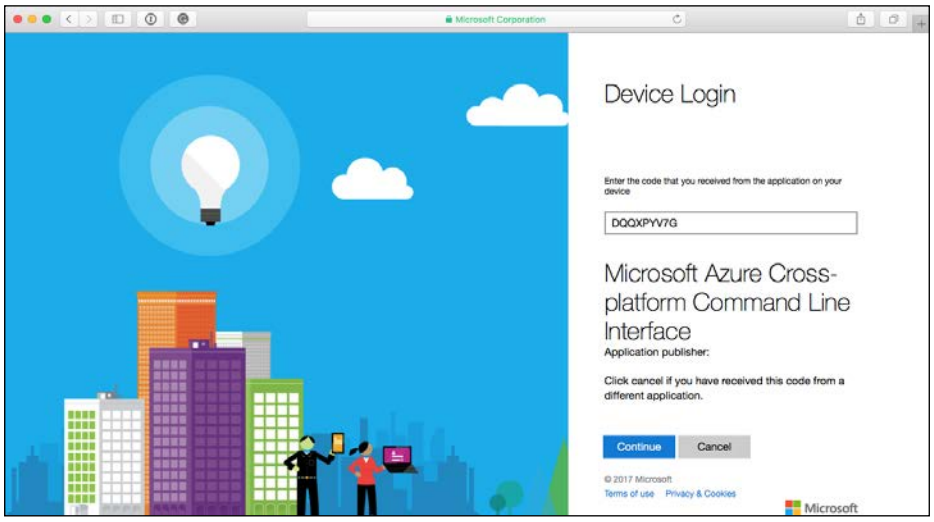
This will download the Azure CLI. The three variables we are passing the command are as follows:

- The name of the service profile
- The name of the resource group
- Which region we would like to launch our cluster in

After a few seconds, you should receive a URL and an authentication code:

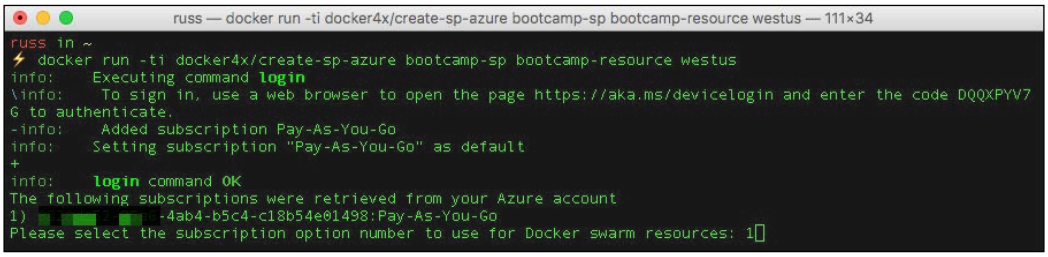


Open `https://aka.ms/devicelogin/` in your browser and enter the code you were given, which in my case was **DQQXPYV7G**:

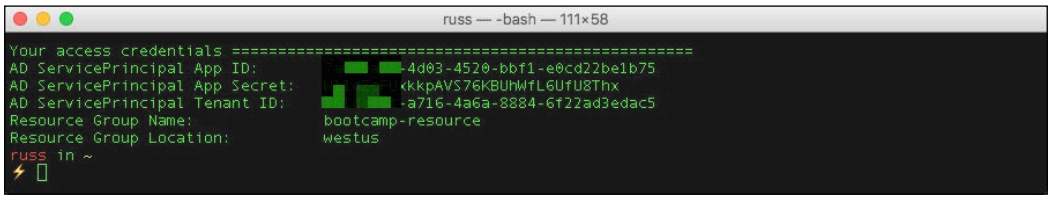


As you can see from the screen above, the application is identifying itself as **Microsoft Azure Cross-platform Command Line Interface** so we know that the request is right; clicking on Continue will ask you to login. Once logged in you will receive confirmation that your request has been authorised and the application has logged in.

After a second or two you should see your command line spring into life, the first thing it will do is ask you which subscription it should use:



Select the right subscription, and then leave the command to finish, it will take around five minutes to complete. At the of the process you should receive your access credentials, make a note of these as you will need them to launch your stack:



Now that we have completed the preparation it is time to launch the Docker for Azure template, you can view the template at the following URL:

```
https://download.docker.com/azure/stable/Docker.tmpl
```

And to launch it simply go to the following URL in your browser:

```
https://portal.azure.com/#create/Microsoft.Template/uri/
https%3A%2F%2Fdownload.docker.com%2Fazure%2Fstable%2FDocker.tmpl
```

You should already be logged in from authorizing the command line interface, if not login and you will be take a to a page which asks for several pieces of information on how you would like your stack to look:

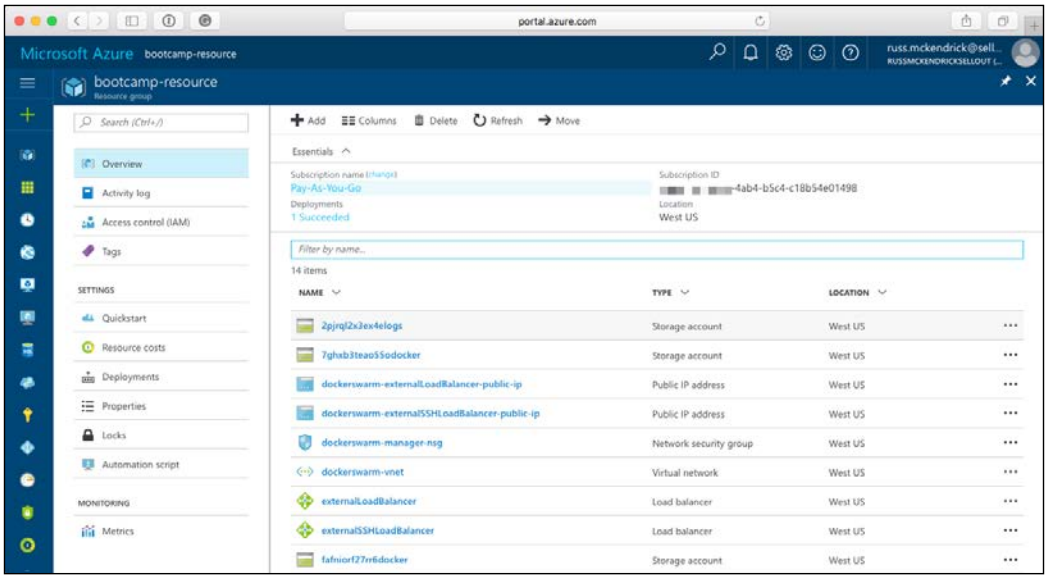
- **Subscription** <Select your subscription>
- **Use existing** <Select your resource group generated in the previous step >
- **Location** <This will be greyed out>
- **Ad Service Principal App ID** <Enter your AD ServicePrincipal App ID generated in the previous step >
- **Ad Service Principal App Secret**: <Enter your AD ServicePrincipal App Secret generated in the previous step >
- **Enable System Prune**: no
- **Manager Count**: 1
- **Manager VM Size**: Standard_D2_v2
- **Ssh Public Key**: <Enter your public key, see below>
- **Swarm Name**: dockerswarm
- **Worker Count**: 3
- **Worker VM Size**: Standard_D2_v2

To quickly copy your public SSH key to your clipboard on a Mac or Linux run the following command (changing the path to your own key if needed):

```
cat ~/.ssh/id_rsa.pub | pbcopy
```

Make sure you tick the box next to **I agree to the terms and conditions stated above**, once you are happy with the contents of the form click on **Purchase**. This will kick off your deployment, the process will take several minutes, once complete your dashboard will have a new resource added to it, depending on your existing resources you may have to scroll to see it or the page may need to be refreshed.

Clicking on **See more** in your resource tile will give you a list of all the resources created by Docker for Azure:



You should be able to see two public IP addresses assigned, one for a **externalLoadBalancer-public-ip** and one for a **externalSSHLoadBalancer-public-ip** make a note of both as we are going to need them, to find out the IP address click on the resource to find more information.

Now that we know the two IP addresses we can SSH into our manager node, SSH is listening on port `50000`, so to SSH to the node run the following command making sure you use the **externalSSHLoadBalancer-public-ip** address:

```
ssh docker@52.160.107.69 -p50000
```

Once logged in, run `docker node ls` and you should see your three worker nodes, if you don't they may still be starting so give it a few minutes more:

```
russ — ssh docker@52.160.107.69 -p50000 — 111×10
russ in ~
⚡ ssh docker@52.160.107.69 -p50000
Welcome to Docker!
swarm-manager000000:~$ docker node ls
ID                            HOSTNAME              STATUS  AVAILABILITY  MANAGER STATUS
0418lj5eygrhpf6a8fv0epz66     swarm-worker000001    Ready   Active
ltxzyppljztylxvbquz98wemz     swarm-worker000000    Ready   Active
qwqmp8vrlq7uch6i6tvrkgioo     swarm-worker000002    Ready   Active
y23eodcyvzg2pgugmt95qnp00 *   swarm-manager000000   Ready   Active        Leader
swarm-manager000000:~$
```

As with Docker for Amazon Web Services, you are SSH'ed into an Alpine Linux host.

Meaning that to install Git we need to change to the root user and using APK to install it:

```
sudo su -
apk update
apk add git
```

Once Git is installed we can check out the Bootcamp repository using;

```
git clone https://github.com/russmckendrick/bootcamp.git
```

And then launch our application stack using the following command:

```
docker stack deploy --compose-file=/root/bootcamp/chapter04/cluster/
docker-compose.yml cluster
```

And make sure everything is running by executing:

```
docker stack ls
docker stack ps cluster
```

Putting the externalLoadBalancer-public-ipaddress into your browser should show you your cluster application. Again, using the CURL command should show us that traffic is being distributed across our containers (remember to use your own Load Balancer IP address):

```
curl -s http://52.160.105.160/ | grep class=
```

There you have it, we have successfully deployed Docker for Azure and our cluster application. The last thing to do is to delete the resources so that we do get any unexpected bills, to do this select **Resource groups** from the left-hand menu and then click on the three dots next to the **bootcamp-resource** entry and select **Delete**.

It will take about 10 minutes to remove all the resources and the group, but it is worth keeping the Azure portal open until the deletion process has completed as you do not want to incur any additional cost.

Depending on how long the resources were live this entire demo would have cost less than $0.10.

Summary

I suspect that by the end of this chapter things were getting very predictable and there were no real surprises, this is by design. As you have experienced, Docker have provided a very powerful clustering solution which once deployed acts in a consistent and predictable way no matter what underlying platform you have launched your cluster on.

There is one important thing which we yet to touch on yet, persistent storage for our containers. This is important, especially in a cluster, as it allows our containers to not only move between hosts but also introduces ways in which we can do rolling updates of our applications.

In the next chapter, we are going to look at both Docker network & volume plugins.

5
Docker Plugins

During DockerCon Europe 2014, there was a round table discussion which took place on the state of the Docker ecosystem, the following problem and possible solution was identified:

> *The problem which Docker currently faces is that by moving to become a platform it is being seen to threaten its own ecosystem. The proposed solution is that Docker ships its own additions to Docker as late-bound, composable, optional extensions and enables other vendors to do likewise. Docker calls this "batteries included but removable".*

During DockerCon 2015 in Seattle, Docker announced the availability of plugins in the experimental branch, the announcement came in the form of a blog post which can be found at `https://blog.docker.com/2015/06/extending-docker-with-plugins/`.

As you can see from the post, Docker provided a solution where third parties can swap out core functionality. Now a user could run the `docker volume` and `docker network` commands along with a driver option to have Docker call external components which add functionality outside of the core Docker Engine while maintaining a high level of compatibility.

In this chapter, we are going to look at two different Docker plugins, the first is a volume plugin called **REX-Ray** and the second is a network plugin called **Weave**.

REX-Ray volume plugin

So far, we have been usingthe local storage which is available on our hosts, as mentioned in *Chapter 4*, *Docker Swarm* that isn't very useful when you potentially have move the storage between multiple hosts either because you are hosting a cluster or because of problems with the host machine itself.

In this example, we are going to be launching a Docker instance in Amazon Web Services, install a volume plugin called REX-Ray, written by EMC, and then launch our WordPress example but this time we will attach AmazonElastic Block Storage volumes to our containers. Once configured, we will move our containers to a second host machine to demonstrate that the data has persisted.

REX-Ray supports several storage types on both public clouds and EMC's own range, as follows:

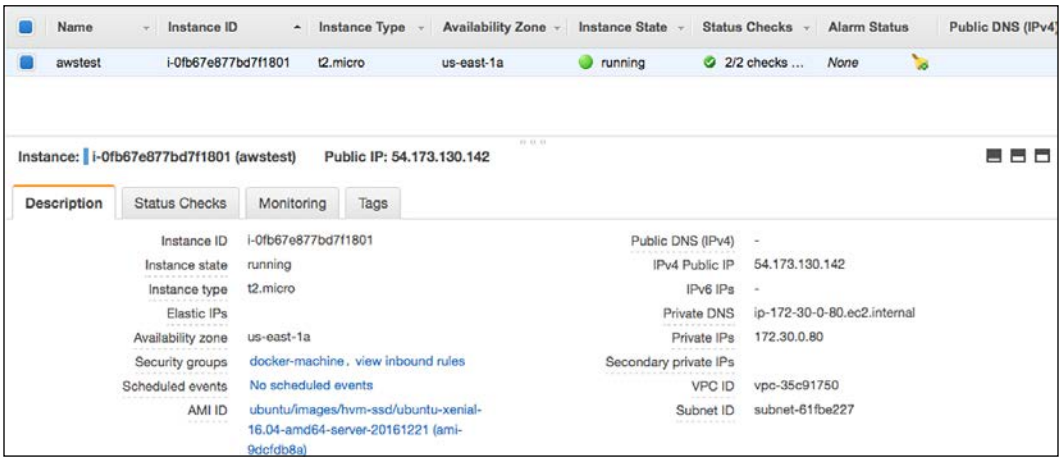
- Amazon Elastic Block Store - `https://aws.amazon.com/ebs/`
- Digital Ocean Block Storage - `https://www.digitalocean.com/products/storage/`
- OpenStack Cinder - `https://wiki.openstack.org/wiki/Cinder/`
- Google Compute Engine - `https://cloud.google.com/compute/docs/disks/`
- EMC Isilon, ScaleIO, VMAX, and XtremIO - `https://www.emc.com/`

The driver is in active development and more types of supported storage are frequently being added, also work is on-going to move the driver over to Dockers new plugin system.

Before we look at installing REX-Ray we need a Docker host in Amazon Web Services, to launch one, use the following command. You can refer to the Amazon Web Services Driver section of *Chapter 2*, *Launching Applications Using Docker*. for details on how to generate your access and secret key and find your VPC ID. Remember to replace the `access-key`, `secret-key` and `vpc-id` with your own:

```
docker-machine create \
    --driver amazonec2 \
    --amazonec2-access-key AKIAJ3GYNKVTEWNMFDHQ \
    --amazonec2-secret-key l2WikM2NIz2GA+1Q2PGKVUCfTNBPBT1Nzgf+jDJC \
    --amazonec2-vpc-id vpc-35c91750 \
    awstest
```

Now that you have your instance launched, you can see it in the AWS Console:



We need to install the REX-Ray plugin. As REX-Ray supports Docker's new plugin format this means we need to run the `docker plugin` command. To start with, we need to configure our local Docker client to connect to our AWS host by running:

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

Now that we are connected to install the plugin, we simply need to run the following command, the `EBS_ACCESSKEY` and `EBS_SECRETKEY` variables are the same we used for Docker Machine, replace them with your own:

```
docker plugin install rexray/ebs \
EBS_ACCESSKEY=AKIAJ3GYNKVTEWNMFDHQ \
EBS_SECRETKEY=l2WikM2NIz2GA+1Q2PGKVUCfTNBPBT1Nzgf+jDJC
```

Before the plugin is installed, you will be asked to confirm that you are OK to grant permissions for the plugin to access various parts of your Docker installation, answer yes (y) to this when prompted and the plugin will be downloaded and installed.

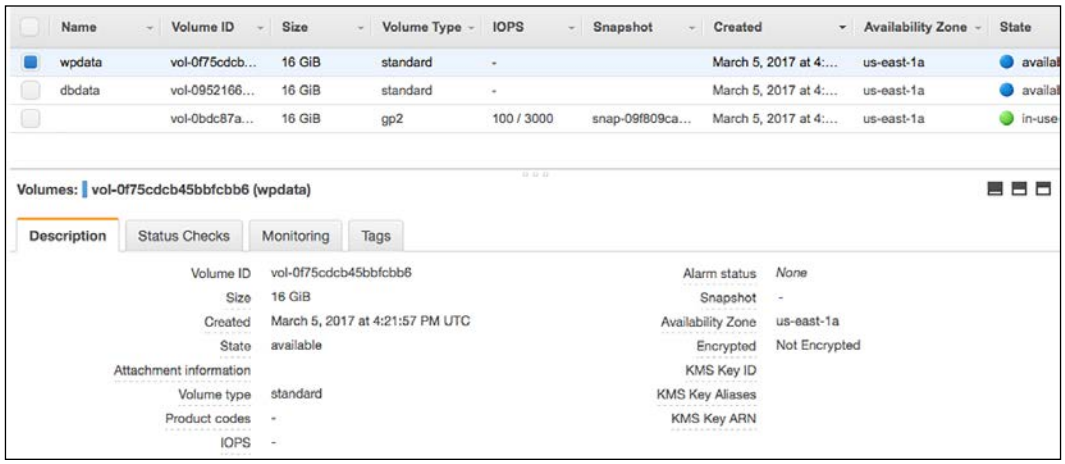
Now that the plugin is installed, we need to create two volumes, one which will hold our WordPress data and the second one will back store our MySQL databases. To create the volumes run the following:

```
docker volume create --driver rexray/ebs --name dbdata
docker volume create --driver rexray/ebs --name wpdata
```

You can see the preceding commands being run in the following terminal:

```
russ in ~/Documents/Code/bootcamp/chapter05/wordpress-rexray on master*
⚡ eval $(docker-machine env awstest)
russ in ~/Documents/Code/bootcamp/chapter05/wordpress-rexray on master*
⚡ docker plugin install rexray/ebs \
    EBS_ACCESSKEY=AKIAJ3GYNKVTEWNMFDHQ \
    EBS_SECRETKEY=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
Plugin "rexray/ebs" is requesting the following privileges:
 - network: [host]
 - mount: [/dev]
 - allow-all-devices: [true]
 - capabilities: [CAP_SYS_ADMIN]
Do you grant the above permissions? [y/N] y
latest: Pulling from rexray/ebs
b855209715aa: Download complete
Digest: sha256:ebe59a30212ae47acc98295af50255d9d102162b8a69c12420886bd75dde5e99
Status: Downloaded newer image for rexray/ebs:latest
Installed plugin rexray/ebs
russ in ~/Documents/Code/bootcamp/chapter05/wordpress-rexray on master*
⚡ docker volume create --driver rexray/ebs --name dbdata
dbdata
russ in ~/Documents/Code/bootcamp/chapter05/wordpress-rexray on master*
⚡ docker volume create --driver rexray/ebs --name wpdata
wpdata
russ in ~/Documents/Code/bootcamp/chapter05/wordpress-rexray on master*
⚡ docker volume ls
DRIVER              VOLUME NAME
rexray/ebs:latest   dbdata
rexray/ebs:latest   wpdata
```

You should also be able to see your two volumes by clicking on **Volumes** in the left-hand side menu of the EC2 section of the AWS Console:



Now we have our two volumes, we need to launch WordPress, to do this we will use the Docker Compose file which can be found in the repo at `/bootcamp/chapter05/wordpress-rexray/`.

As you can see from the `docker-compose.yml` file, we are building a WordPress image with `wp-cli` installed:

```
version: "3"

services:
mysql:
     image: mysql
     volumes:
       - dbdata:/var/lib/mysql
     restart: always
     environment:
       MYSQL_ROOT_PASSWORD: wordpress
       MYSQL_DATABASE: wordpress
wordpress:
depends_on:
       - mysql
     build: ./
     volumes:
       - wpdata:/var/www/html
     ports:
       - "80:80"
     restart: always
     environment:
       WORDPRESS_DB_PASSWORD: wordpress

volumes:
dbdata:
      external:
        name: dbdata
wpdata:
      external:
        name: wpdata
```

As you can also see from the end of the file, we are telling Docker Compose to use the two external volumes we have already created with the `docker volume create` command.

To build our WordPress image and launch the containers run the following command:

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

You can check your containers up by running:

```
docker-compose ps
```

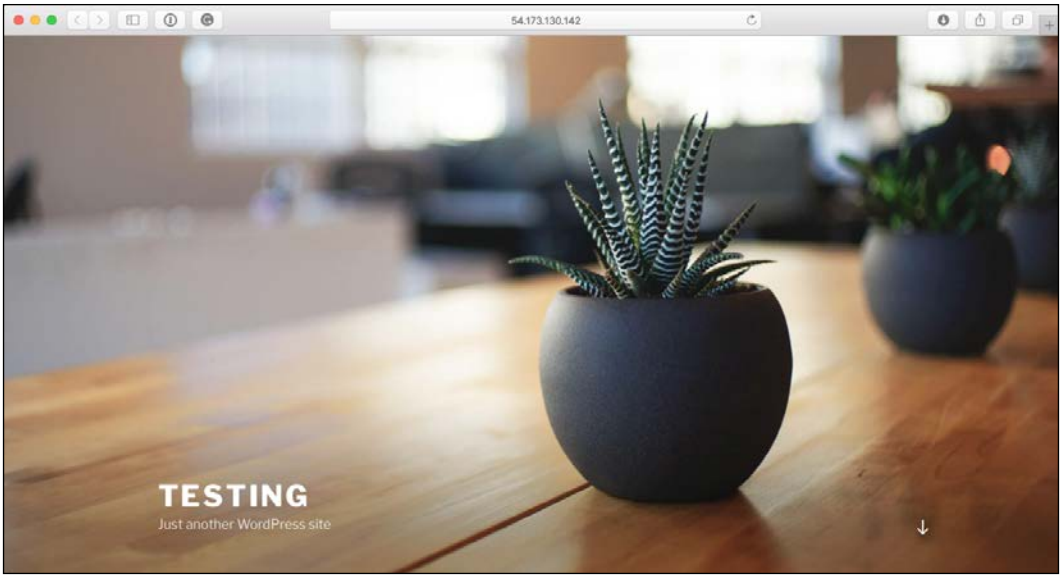
Now that the two containers we make our WordPress application are up and running you can quickly install WordPress by running the following command (update the variables as needed):

```
$awshost = "$(docker-machine ip awstest)"
docker-compose exec wordpress wp core install --url=http://$(awshost)/
--title=Testing --admin_user=admin --admin_password=adminpassword
--admin_email=russ@mckendrick.io
```

Once installed, you should see a message which says **Success: WordPress installed successfully**. This means that you can open your installation in a browser by running:

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

This should present you with the now familiar WordPress site:



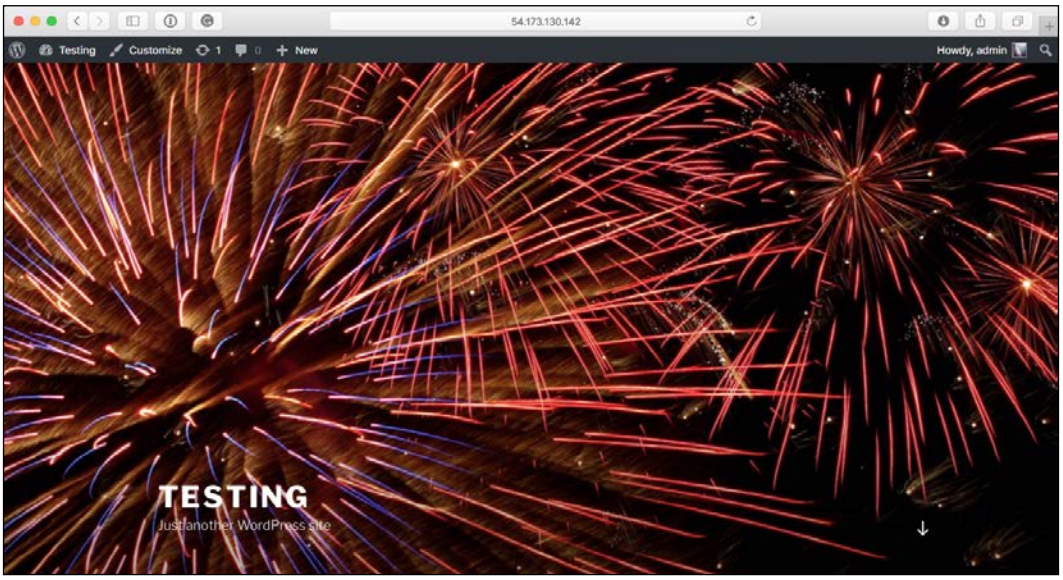
Now let's make a change to our WordPress installation so we can be sure that when we move our application between hosts everything works as expected. We are going to be replacing the image of the plant with fireworks. To do this we need to customize our theme, to get to the theme edit page run the following:

```
open "http://$(docker-machine ip awstest)/wp-admin/customize.
php?return=%2Fwp-admin%2Fthemes.php"
```

You will be prompted to login using the admin username and password which if you followed the installation will be `admin` / `adminpassword` or if you entered your own then use them.

Once you have the page open click on Header Media in the left-hand menu. Scroll down to where it says **Add new image** in the left-hand menu and follow the on-screen prompts to upload, crop and set the new header image, you can find an image called `fireworks.jpg` in the repo or use your own image. Once you have finished click on **Save & Publish**.

Going back to your sites home page should then show your new header image:



Before we remove our Docker host we need to make a note of it's IP address, to do this run the following command:

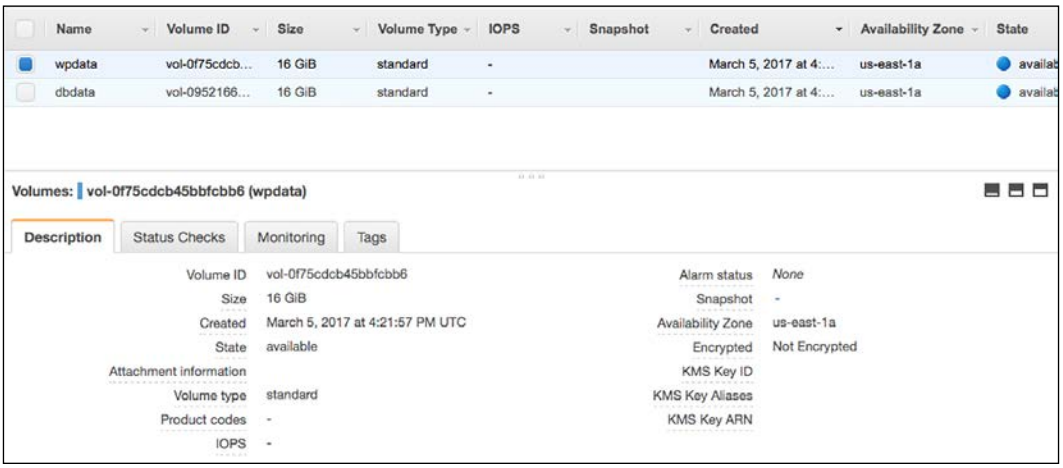
```
echo $(docker-machine ip awstest)
```

And write down the IP address as we are going to need it, in my case the IP address was `54.173.130.142`.

Now let's remove our host using the following command:

```
docker-machine rm awstest
```

Once the host has been removed our two volumes are shown as **available** within the AWS console:



That is our WordPress and database data, to access it on a new Docker host we need to first launch one. To do this run the following command again remembering to replace the credentials and vpc id with your own:

```
docker-machine create \
    --driver amazonec2 \
    --amazonec2-access-key AKIAJ3GYNKVTEWNMFDHQ \
    --amazonec2-secret-key l2WikM2NIz2GA+1Q2PGKVUCfTNBPBT1Nzgf+jDJC \
    --amazonec2-vpc-id vpc-35c91750 \
    awstest2
```

Once the new Docker host is up and running the following command to switch our local client over and install REX-Ray:

```
eval $(docker-machine env awstest2)
docker plugin install rexray/ebs \
  EBS_ACCESSKEY=AKIAJ3GYNKVTEWNMFDHQ \
  EBS_SECRETKEY=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
```

Once REX-Ray is installed, we need to make it aware of our two existing volumes, to do this simply run the following command:

```
docker volume create --driver rexray/ebs --name dbdata
docker volume create --driver rexray/ebs --name wpdata
```

Do not worry, it will not overwrite our existing volumes, it will just make Docker aware that they are there as REX-Ray uses the name you assign to volume rather than a unique ID if it comes across a volume with the name you have told it to use it will assume that is the volume you meant to use, so be careful when naming your volumes as they will be attached to the running container.

You may notice that the commands execute a lot quicker this time, this is because the volumes are already there and do not need re-creating.Running:

```
docker volume ls
```

should show our two volumes are there as before.

Now we need to launch WordPress, to do that just run:

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

If you were to try and access your WordPress site now, you would see a very broken looking site with content, but no styling or images.

This is because the database is still referencing the IP address of the Docker host we terminated, to the database. Run the following the command making sure to replace the IP address in the command to that of your previous Docker host (remember mine was `54.173.130.142`):

```
docker-compose exec wordpress wp search-replace 54.173.130.142 $(docker-
machine ip awstest2)
```

You should see a list of every table within the database along with confirmation of how many instances of the IP address it has replaced with that of the new Docker host.

Going to your new WordPress installation by running:

```
open http://$(docker-machine ip awstest2)
```

Should show your cover image is intact and the WordPress installation is exactly how you left it, apart from the change in IP address.

When you have finished test you can remove your installation by running the following commands:

```
docker-compose stop
docker-compose rm
docker volume rmdbdata
docker volume rmwpdata
docker-machine rm awstest2
```

You may notice that when you run the `docker volume rm` commands you are not prompted to confirm your actions, so be careful.

Checking your AWS console should confirm that your Docker host has been terminated and your two volumes have been removed.

WeaveNetwork Plugin

Weave are one of the original Docker plugins, in-fact they were involved in the round table discussions around Dockers plugin functionality, and Weave was included in the original plugin announcement mentioned at the start of this chapter.

Weave describe their network plugin as:

> *Quickly, easily, and securely network and cluster containers across any environment (on premises, in the cloud, or hybrid) with zero code or configuration.*

Anyone who worked with software defined networks will know that this is quite a bold claim, especially a Weave is creating a mesh network.For a full explanation of what that means, I would recommend reading through Weaves own overview which can be found at `https://www.weave.works/docs/net/latest/how-it-works/`.

Rather than going into any more detail. let's roll our sleeves up and perform an installation. To start with, let's bring up two independent Docker hosts DigitalOcean using Docker Machine.

To make it interesting, we will launch one host in New York Cityand the other in London. As these are going to be acting as individual hosts there is no need to configure Docker Swarm – which is what you would typically need to for multi-host networking with Docker.

To launch the Docker host in New York City run:

```
docker-machine create \
    --driver digitalocean \
    --digitalocean-access-token
57e4aeaff8d7d1a8a8e46132969c2149117081536d50741191c79d8bc083ae73 \
    --digitalocean-region nyc1\
    weave-nyc
```

And then for the Docker host in London run:

```
docker-machine create \
    --driver digitalocean \
    --digitalocean-access-token
57e4aeaff8d7d1a8a8e46132969c2149117081536d50741191c79d8bc083ae73 \
    --digitalocean-region lon1 \
    weave-lon
```

Now that we have our two Digital Ocean hosts we need to get Weave up and running.At the time of writing, Weave has not completed the transition to Dockers native plugin architecture and it is due very soon, so we will be using a control script to configure Weave.

First, we need to download the control scripton our NYC Docker host:

```
docker-machine ssh weave-nyc 'curl -L git.io/weave -o /usr/local/bin/
weave; chmoda+x /usr/local/bin/weave'
```

Once downloaded we can launch Weave using the following command:

```
docker-machine ssh weave-nyc weave launch --password 3UnFh4jhahFC
```

This will download and launch several containers on the Docker host, once downloaded the Weave will be configured and the password set meaning that if you want to add a host to network you will need to provide a valid password.

If you do not define a password then anyone will be able to connect to your Weave network, which is fine if you know that your host machines are running on an isolated closed network, however as we are sending traffic over the public internet we have set a password.

You can check the containers by running:

```
docker $(docker-machine config weave-nyc) container ps
```

Now that we have the three containers we need launched, it is time to install Weave on our London Docker host and then connect it to our NYC Docker host. To do the installation run the following commands:

```
docker-machine ssh weave-lon 'curl -L git.io/weave -o /usr/local/bin/
weave; chmoda+x /usr/local/bin/weave'
```

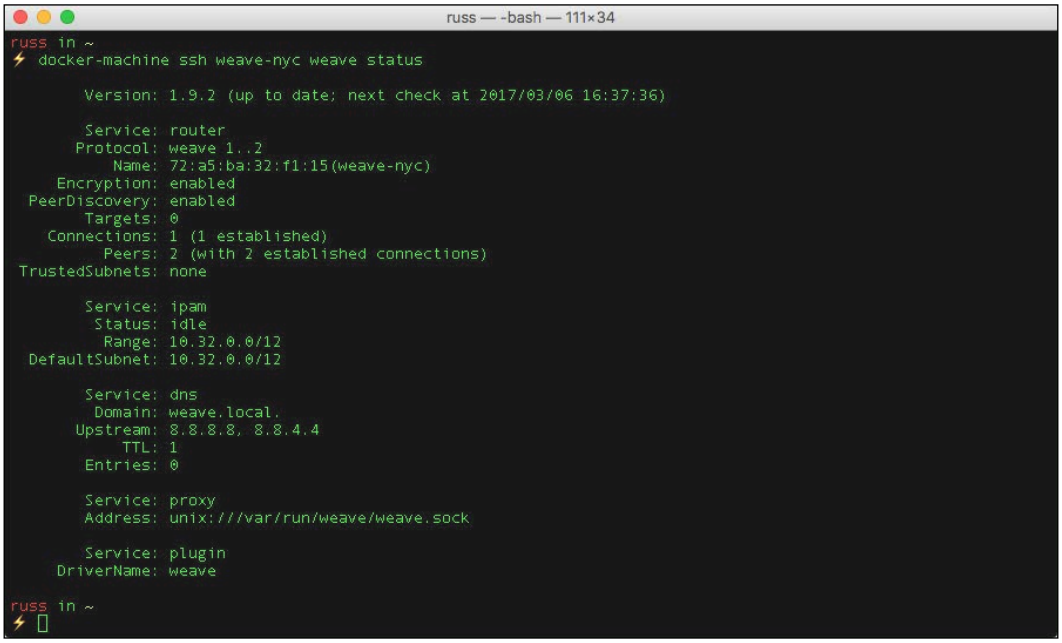
```
docker-machine ssh weave-lon weave launch --password 3UnFh4jhahFC
```

Once the three containers have launched, simply run the following command to connect to our NYC Docker host:

```
docker-machine ssh weave-lon weave connect "$(docker-machine ip weave-
nyc)"
```

Once our second host has been configured you can check the status of the Weave mesh network by running:

```
docker-machine ssh weave-nyc weave status
```



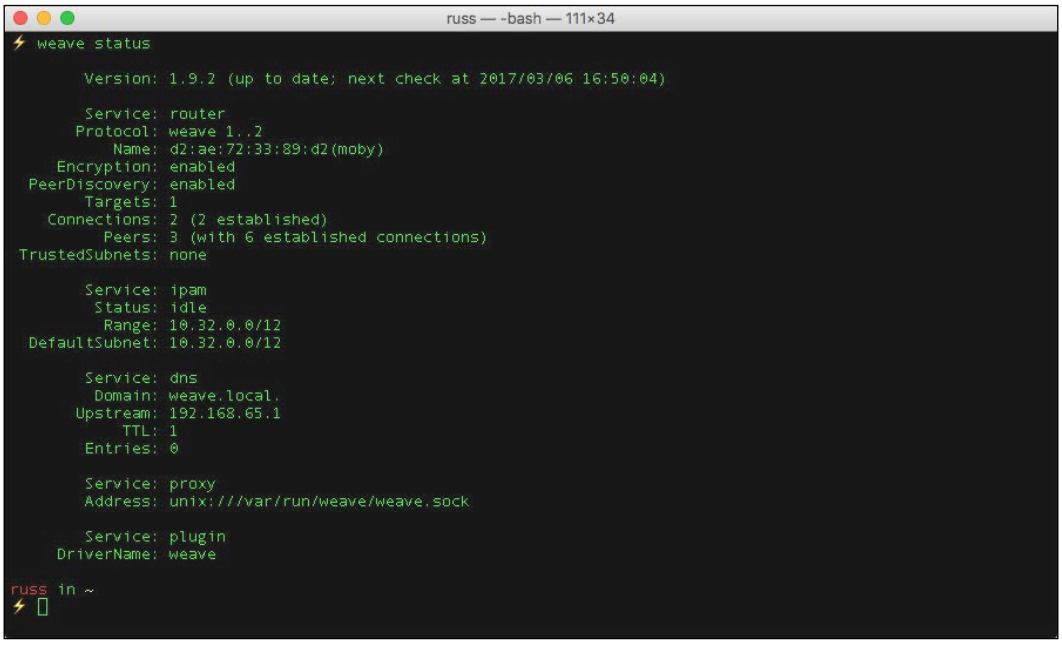
As you can see from the preceding terminal above, we have five services running, and other than providing a password, we didn't have to configure any of them.

As I am running a Mac OS machine, I am also going to install Weave locally, the same instructions will also work on a Linux machine.

The following commands will install the Weave control script which will be used to launch the containers within your Docker for Mac installation and connect to our Weave mesh network:

```
sudo curl -L git.io/weave -o /usr/local/bin/weave; sudochmoda+x /usr/
local/bin/weave
weave launch --password 3UnFh4jhahFC
weave connect "$(docker-machine ip weave-nyc)"
```

Once installed and connected, running `weave status` locally should show you that there are now 3 peers with 6 established connections:



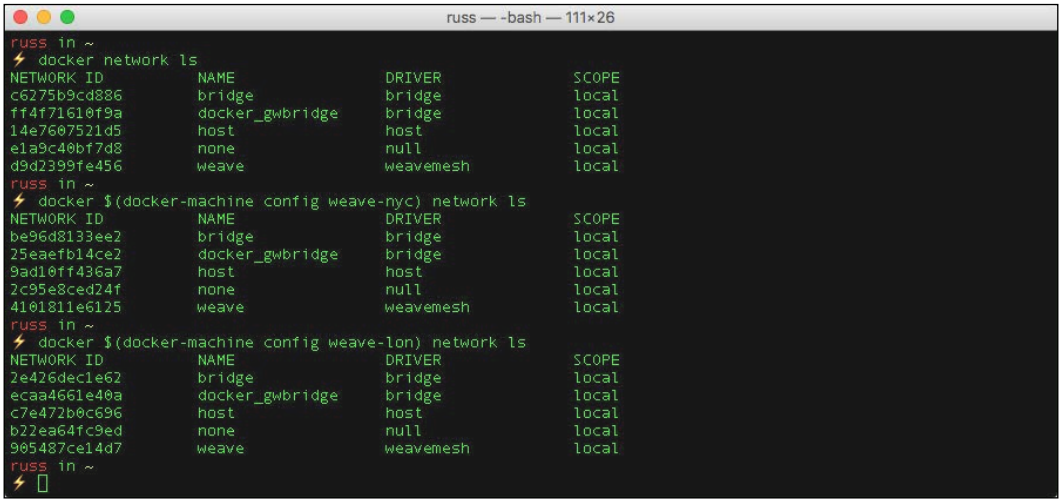
So now we have three Docker hosts:

- One in NYC hosted by Digital Ocean
- One in London hosted by Digital Ocean
- Our local Docker host running on Docker for Mac (or Linux)

All with a network called **weave** using the `weavemesh` driver. You can confirm this by running:

```
docker network ls
docker $(docker-machine config weave-nyc) network ls
docker $(docker-machine config weave-lon) network ls
```

You should see something similar to the following terminal output:



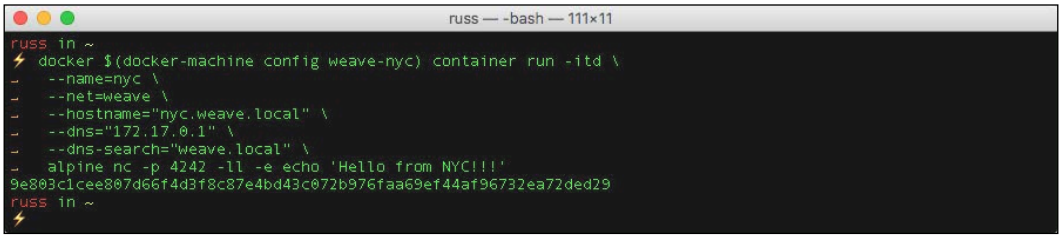
Now we are ready to start launching containers into our Weave network and demonstrate that they can communicate with each other.

Netcat is a service which allows you to be read and write to a network using TCP or UDP.

Let's start by launching a container in NYC running Netcat(nc). Each time a request is sent to port 4242 `nc` will answer with `Hello from NYC!!!`:

```
docker $(docker-machine config weave-nyc) container run -itd \
  --name=nyc \
  --net=weave \
  --hostname="nyc.weave.local" \
  --dns="172.17.0.1" \
  --dns-search="weave.local" \
  alpine nc -p 4242-ll -e echo 'Hello from NYC!!!'
```

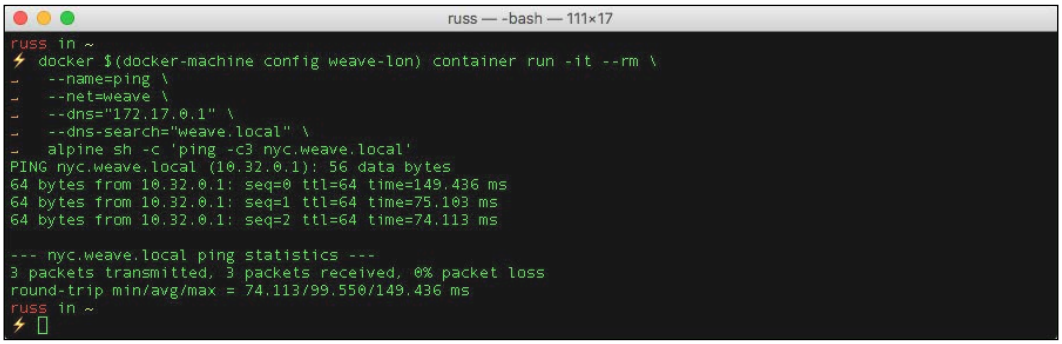
As you can see from the Docker command, we are passing quite a few different options, we are telling the container which network to use, as well configuring the DNS resolver within the container and setting a hostname of `nyc.weave.local`.



Now that we have our NYC container up and running, the first thing to do is to check if we can ping from our London Docker host, to do this run the following:

```
docker $(docker-machine config weave-lon) container run -it --rm \
  --name=ping \
  --net=weave \
  --dns="172.17.0.1" \
  --dns-search="weave.local" \
  alpine sh -c 'ping -c3 nyc.weave.local'
```

This will send three pings to `nyc.weave.local`, all of which should be answered:

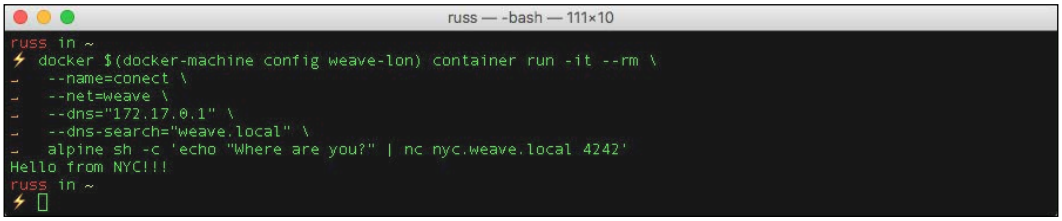


Now that have confirmed that we can Ping the NYC container, we need to connect to port `4242` and check if we get the response we expect:

```
docker $(docker-machine config weave-lon) container run -it --rm \
  --name=conect \
  --net=weave \
```

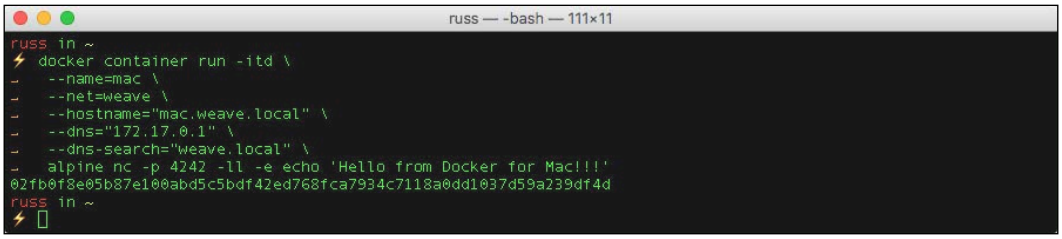
```
  --dns="172.17.0.1" \
  --dns-search="weave.local" \
  alpine sh -c 'echo "Where are you?" | ncnyc.weave.local 4242'
```

You should receive the message **Hello from NYC!!!**:



Now let's launch a container on our local Docker host using the following command:

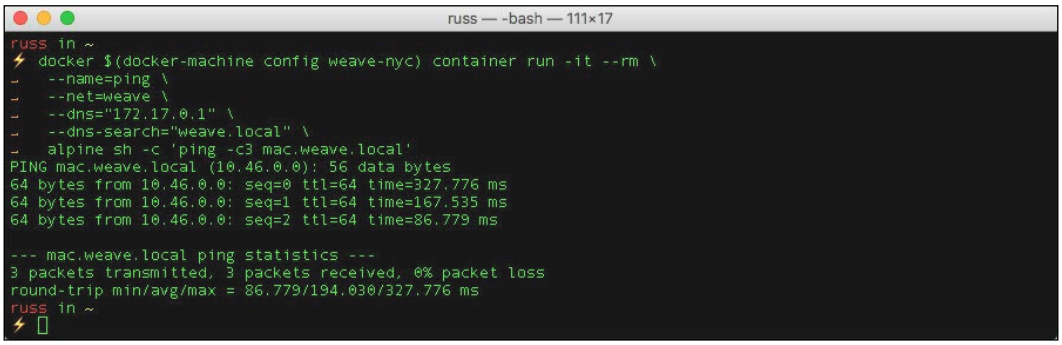
```
docker container run -itd \
  --name=mac \
  --net=weave \
  --hostname="mac.weave.local" \
  --dns="172.17.0.1" \
  --dns-search="weave.local" \
  alpine nc -p 4242 -ll -e echo 'Hello from Docker for Mac!!!'
```



As before, we will do a simple ping test to our local container:

```
docker $(docker-machine config weave-nyc) container run -it --rm \
  --name=ping \
  --net=weave \
  --dns="172.17.0.1" \
  --dns-search="weave.local" \
  alpine sh -c 'ping -c3 mac.weave.local'
```

As expected, we receive a response:



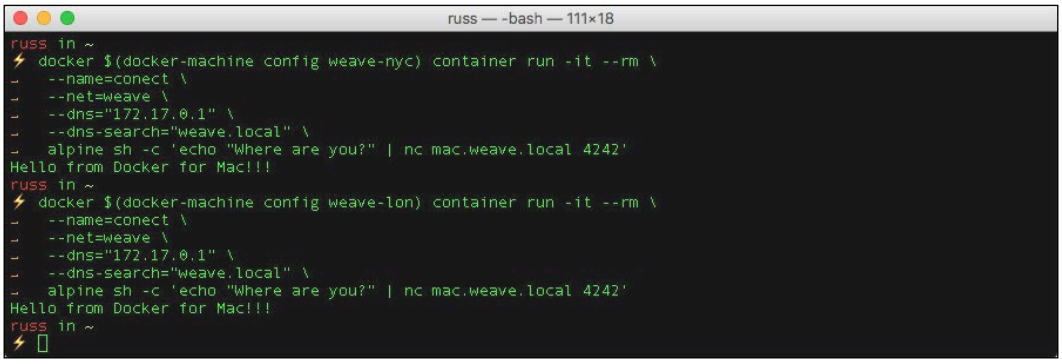
It's a little slow to start with, but it eventually gets better. Now that we know we can ping our local container lets connect to port `4242` and check the response. First, from our NYC Docker host:

```
docker $(docker-machine config weave-nyc) container run -it --rm \
  --name=conect \
  --net=weave \
  --dns="172.17.0.1" \
  --dns-search="weave.local" \
  alpine sh -c 'echo "Where are you?" | ncmac.weave.local 4242'
```

Then from our London Docker host:

```
docker $(docker-machine config weave-lon) container run -it --rm \
  --name=conect \
  --net=weave \
  --dns="172.17.0.1" \
  --dns-search="weave.local" \
  alpine sh -c 'echo "Where are you?" | ncmac.weave.local 4242'
```

As you can see from the following terminal output we got the answer we expected to receive:



To tidy up your local Docker host run the following commands:

```
docker container stop mac
docker container rm mac
weave stop
sudorm -f /usr/local/bin/weave
```

And then to terminate our two Digital Ocean hosts run:

```
docker-machine stop weave-lon weave-nyc
docker-machine rm weave-lon weave-nyc
```

While these tests haven't been as visually interesting as the walkthrough of the REX-Ray Volume plugin, as you have seen, Weave is an incredibly powerful software-defined network, which is very easy to configure.

Speaking fromexperience, this is a difficult combination to pull off, as most SDN solutions are incredibly complex to install, configure, and maintain.

We have only touched on what is possible with Weave. For a full feature list, along with instructions on some most of the advanced use cases, refer to `http://docs.weave.works/weave/latest_release/features.html`.

Summary

Hopefully you are now starting to see use cases for different types of plugins.For example, a developer is fine working with local volumes, however for production traffic you would want to have some sort of either shared or block storage which is accessible to containers across multiple Docker hosts.

With plugins, this is possible without any real changes to your user's workflow as you know exactly how Docker handles volumes created with the `docker volume create` command.

As already mentioned, Docker are in the process of transitioning legacy plugins to a new architecture, a list of legacy plugins can be found at the following URL `https://docs.docker.com/engine/extend/legacy_plugins/` and new plugins which use the new architecture a can be found at `https://store.docker.com/search?q=&type=plugin`.

In the next chapter, we are going to look at how to monitor your containers, and what to do if anything goes wrong.

6
Troubleshooting and Monitoring

In this chapter, we are going to look at commands which will come in useful when troubleshooting your containers, all the commands we will look at are part of the core Docker Engine, we will also look at a way by which you can debug your Dockerfiles.

Once we have finished with the Troubleshooting commands, we will look at how we can monitor our containers using cAdvisor with a Prometheus backend fronted by a Grafana dashboard – don't worry, it is not as complicated as it sounds.

As we are going to be exposing services, some using default credentials I would recommend that you use your local Docker installation for this chapter.

Troubleshooting containers

Computer programs (software) sometimes fail to behave as expected. This is due to faulty code or due to the environmental changes between the development, testing, and deployment systems. Docker container technology eliminates the environmental issues between development, testing, and deployment as much as possible by containerizing all the application dependencies. Nonetheless, there could still be anomalies due to faulty code or variations in the kernel behavior, which needs debugging. Debugging is one of the most complex processes in the software engineering world and it becomes much more complex in the container paradigm because of the isolation techniques. In this section, we are going to learn a few tips and tricks to debug a containerized application using the tools native to Docker, as well as the tools provided by external sources.

Initially, many people in the Docker community individually developed their own debugging tools, but later Docker started supporting native tools, such as `exec`, `top`, `logs`, `events`, and many more. In this section, we will dive deep into the following Docker tools:

- `exec`
- `ps`
- `top`
- `stats`
- `events`
- `logs`
- `attach`

We shall also consider debugging a Dockerfile.

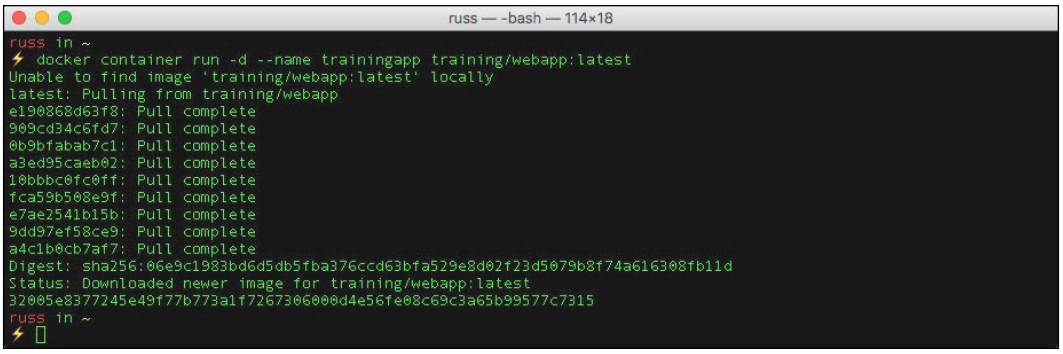
The exec command

The `docker container exec` command provided the much-needed help to users, who are deploying their own web servers or other applications running in the background.

Now, it is not necessary to log in to run the SSH daemon in the container.

First, launch a container:

```
docker container run -d --name trainingapp training/webapp:latest
```



Second, run the `docker container ps` command to get the container ID. Now you have the container ID you can run the `docker container exec` command to log in to the container using either the container ID or as we have named it `trainingapp` you can use that:

```
docker container exec -it 32005e837724 bash
```

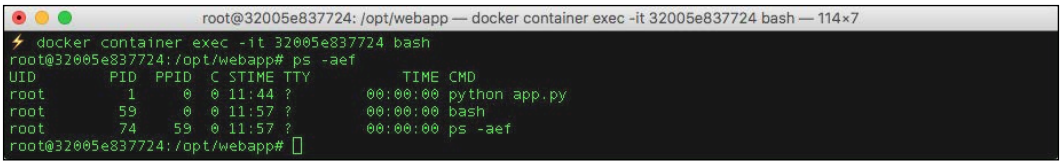
Please note, not every container will have bash installed, some such Alpine Linux don't have bash out of the box but instead uses sh, which bash was based on.

It is important to note that the `docker container exec` command can only access the running containers, so if the container stops functioning then you need to restart the stopped container to proceed. The `docker container exec` command spawns a new process in the target containers namespace using the Docker API and CLI.

A containers name space is what separates the containers from each other, for example you can have several containers all running the same process, but because the processes have been launched within each of the containers namespace they are isolated from one another. A good example of this is are MySQL processes, on a traditional server trying to run more than one MySQL server process will mean that you need to start the process on different ports, use different lock, PID and log files as well as different init scripts.

As Docker is isolating each MySQL server process all you need to worry about is that if you are exposing the MySQL port on the host machine is that you don't assign it on the same port as another container.

So, if you run the `ps -aef` command inside the target container, it looks like this:



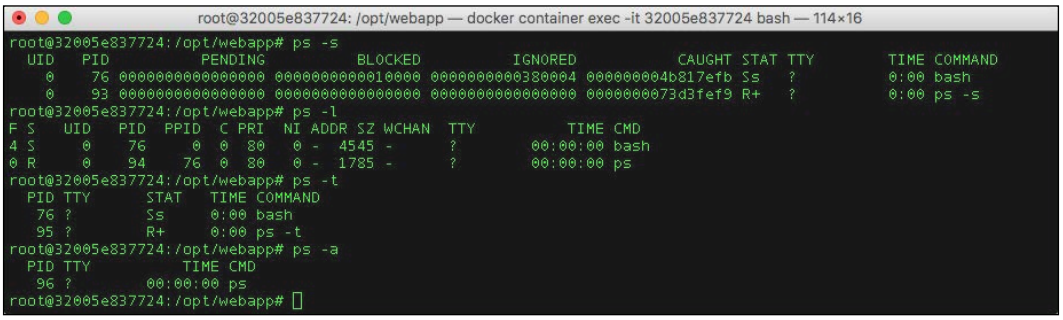
Here, `python app.y` is the application that is already running in the target container, and the `docker container exec` command has added the `bash` process inside the container. If you run `kill -9 59` (replacing the `59` with the PID of your own `bash` process), you will be automatically logged out of the container.

It is recommended that you use the `docker container exec` command only for monitoring and diagnostic purposes, and I personally believe in the concept of one process per container, which is one of the best practices widely accentuated.

The ps command

The `ps` command, which is available inside the container, is used to see the status of the process. This is like the standard `ps` command in the Linux environment and is not a `dockercontainerps` command that we run on the Docker host machine.

This command runs inside the Docker container:



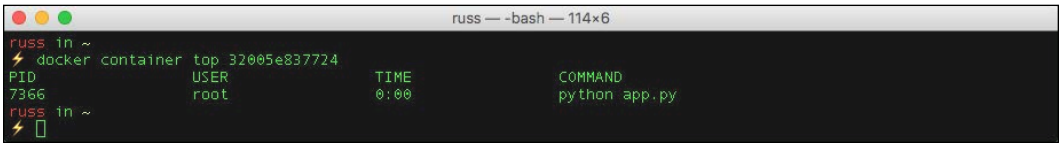
Use `ps --help <simple|list|output|threads|misc|all>` or `ps --help <s|l|o|t|m|a>` for additional help text.

The top command

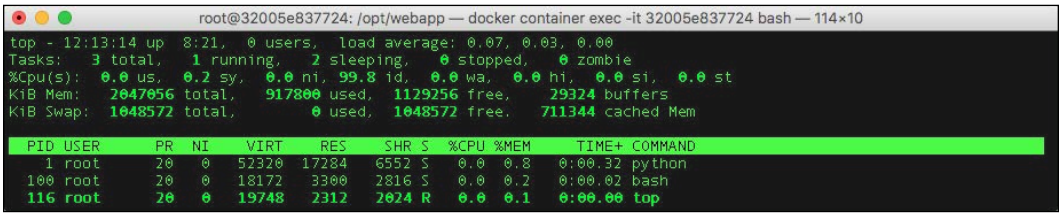
You can run the top command from the Docker host machine using the following command:

```
docker container top CONTAINER [ps OPTIONS]
```

This gives a list of the running processes of a container without logging into the container, as follows:



The within the container the top command provides information about the CPU, memory, and swap usage just like any normal Linux host:



In case you get the error as `error - TERM environment variable not set` while running the `top` command inside the container, perform the following steps to resolve it.

Run `echo$TERM` and if you get the result `dumb`, then, run the following command:

```
export TERM=dumb
```

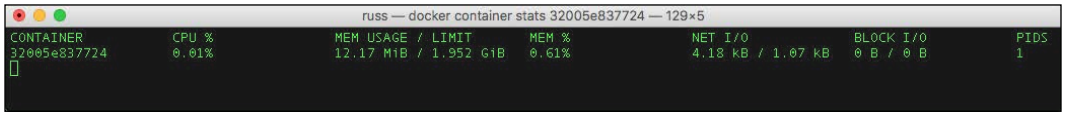
This will resolve your error and you can run the `top` command.

The stats command

The `docker container stats` command provides you with the capability to view the memory, CPU, and the network usage of a container from a Docker host machine, as illustrated here. Running the following command:

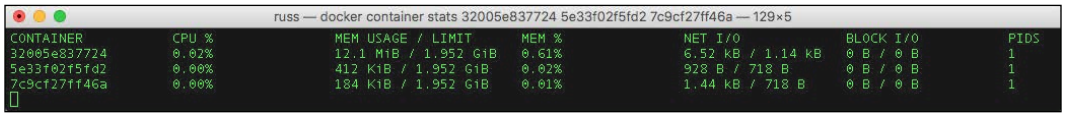
```
docker container stats 32005e837724
```

Gives you the following:



You can run the `stats` command to also view the usage for multiple containers:

```
docker container stats 32005e837724 5e33f02f5fd2 7c9cf27ff46a
```



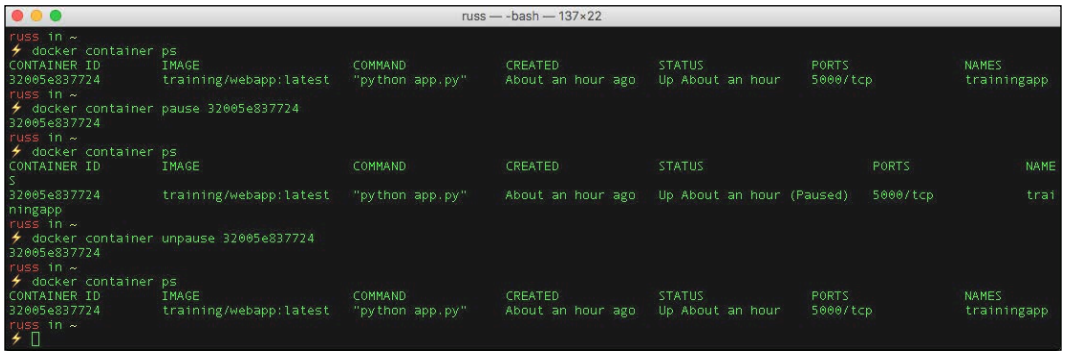
Since Docker 1.5, you have been able to access to container statistics *read only* parameters. This will streamline the CPU, memory, network IO, and block IO of your containers.

This helps you choose the resource limits and in profiling. The Docker stats utility provides you with these resource usage details only for running containers.

You can get detailed information using the endpoint APIs at the following URL `https://docs.docker.com/engine/api/v1.26/`.

The Docker events command

Docker containers will report the following real-time events: `create`, `destroy`, `die`, `export`, `kill`, `omm`, `pause`, `restart`, `start`, `stop`, and `unpause`. Let's `pause` and `unpause` our container:

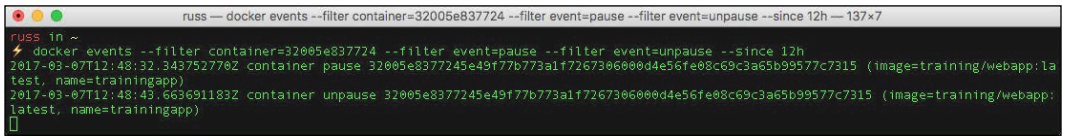


If you specify an image it will also report the `untag` and `delete` events.

Using multiple filters will be handled as an `AND` operation, for example:

```
docker events --filter container=32005e837724 --filter event=pause
--filter event=unpause --since 12h
```

Preceding will display all `pause` and `unpause` events for the container `a245253db38b` for the last 12 hours:



Currently, the supported filters are `container`, `event`, and `image`.

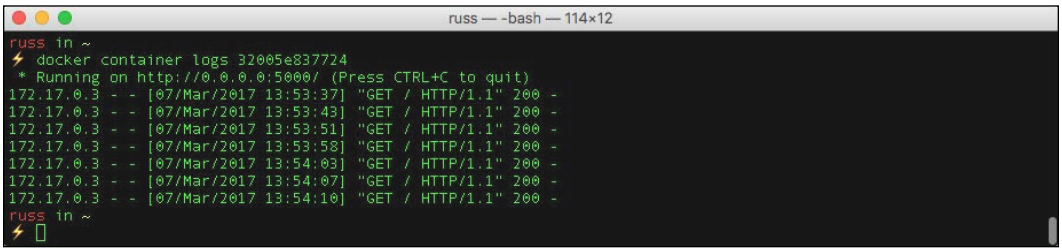
The logs command

This command fetches the log of a container without logging into the container. It batch-retrieves logs present at the time of execution. These logs are the output of STDOUT and STDERR. The general usage is shown in:

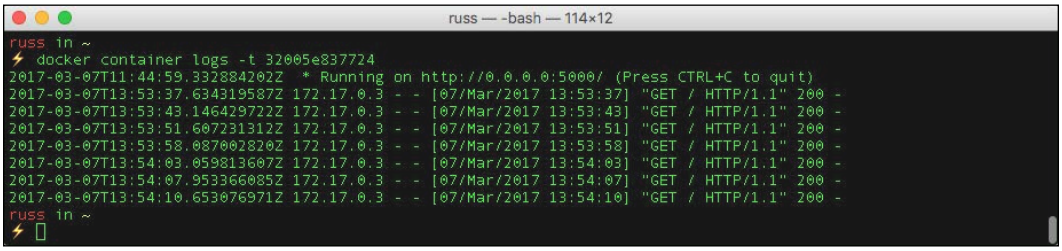
```
docker container logs [OPTIONS] CONTAINERID
```

The `--follow` option will continue to provide the output till the Docker logs command is terminated printing any new log entries to the screen in real time,`-t` will provide the timestamp, and `--tail=<number of lines>` will show the number of lines of the log messages of your container:

```
docker container logs 32005e837724
```




```
docker container logs -t 32005e837724
```



We also used the `docker container logs` command in previous chapters to view the logs of our database containers.

The attach command

This command attaches the running container and it is very helpful when you want to see what is written in `stdout` in real time, let's launch new test container which outputs something to `stdout`:

```
docker container run -d --name=newtest alpine /bin/sh -c "while true; do
sleep 2; df -h; done"
```

Now we can attach to the container using the following command to see the output;

```
docker container attach newtest
```

By default, this command attaches `stdin` and proxies signals to the remote process. Options are available to control both behaviors. To detach from the process, use the default *Ctrl* + *Q* sequence.

Debugging a Dockerfile

Every instruction we set in the Dockerfile is going to be built as a separate, temporary image for the other instruction to build itself on top of the previous instruction.

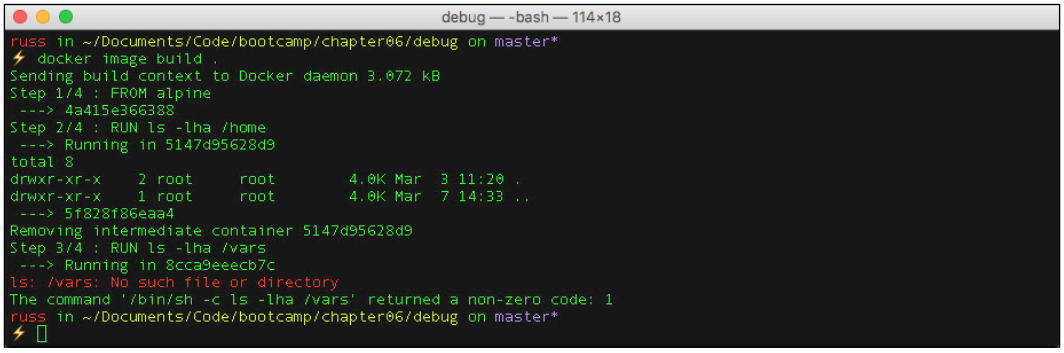
There is a Dockerfile in the repo at `/bootcamp/chapter06/debug`:

```
FROM alpine
RUN ls -lha /home
RUN ls -lha /vars
CMD echo Hello world
```

Building the image using the following command:

```
docker image build
```

Gives you the following output:



So, there is an error in our Docker file. You may notice there is a line in the output which says `--->5f828f86eaa4`this is actually an image file which was built following the successful execution of the `RUN ls -lha /home` line.

```
debug — -bash — 114×7
russ in ~/Documents/Code/bootcamp/chapter06/debug on master*
⚡ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
<none>              <none>              5f828f86eaa4        11 minutes ago      3.99 MB
alpine              latest              4a415e366388        3 days ago          3.99 MB
russ in ~/Documents/Code/bootcamp/chapter06/debug on master*
⚡
```

This means that we can launch a container using this image:

```
docker container run -it --name=debug 5f828f86eaa4 /bin/sh
```

Notice that as we are using Alpine Linux as our base we are using `/bin/sh` rather than `/bin/bash`

We can then debug our application, which in this case is simple:

```
debug — -bash — 114×21
russ in ~/Documents/Code/bootcamp/chapter06/debug on master*
⚡ docker container run -it --name=debug 5f828f86eaa4 /bin/sh
/ # ls -lha /vars
ls: /vars: No such file or directory
/ # ls -lha /var
total 48
drwxr-xr-x   12 root     root        4.0K Mar  3 11:20 .
drwxr-xr-x    1 root     root        4.0K Mar  7 14:49 ..
drwxr-xr-x    4 root     root        4.0K Mar  3 11:20 cache
dr-xr-xr-x    2 root     root        4.0K Mar  3 11:20 empty
drwxr-xr-x    5 root     root        4.0K Mar  3 11:20 lib
drwxr-xr-x    2 root     root        4.0K Mar  3 11:20 local
drwxr-xr-x    3 root     root        4.0K Mar  3 11:20 lock
drwxr-xr-x    2 root     root        4.0K Mar  3 11:20 log
drwxr-xr-x    2 root     root        4.0K Mar  3 11:20 opt
drwxr-xr-x    2 root     root        4.0K Mar  3 11:20 run
drwxr-xr-x    3 root     root        4.0K Mar  3 11:20 spool
drwxrwxrwt    2 root     root        4.0K Mar  3 11:20 tmp
/ # exit
russ in ~/Documents/Code/bootcamp/chapter06/debug on master*
⚡
```

Debugging is a process of analyzing what's going on and it's different for every situation, but usually the way we start debugging is by trying to manually make the instruction that fail work manually and understand the error. When I get the instruction to work, I usually exit the container, update my Dockerfile and repeat the process until I have something working.

Notice that when the line which is causing the error is corrected (by supplying the correct line `RUN ls -lha /var`) and we try the build again that Docker doesn't create a new image for the one step which was successful:

```
debug-working — -bash — 114×32
russ in ~/Documents/Code/bootcamp/chapter06/debug-working on master*
⚡ docker build .
Sending build context to Docker daemon 2.048 kB
Step 1/4 : FROM alpine
 ---> 4a415e366388
Step 2/4 : RUN ls -lha /home
 ---> Using cache
 ---> 5f828f86eaa4
Step 3/4 : RUN ls -lha /var
 ---> Running in ef432bd58499
total 48
drwxr-xr-x   12 root     root        4.0K Mar  3 11:20 .
drwxr-xr-x    1 root     root        4.0K Mar  7 14:53 ..
drwxr-xr-x    4 root     root        4.0K Mar  3 11:20 cache
dr-xr-xr-x    2 root     root        4.0K Mar  3 11:20 empty
drwxr-xr-x    5 root     root        4.0K Mar  3 11:20 lib
drwxr-xr-x    2 root     root        4.0K Mar  3 11:20 local
drwxr-xr-x    3 root     root        4.0K Mar  3 11:20 lock
drwxr-xr-x    2 root     root        4.0K Mar  3 11:20 log
drwxr-xr-x    2 root     root        4.0K Mar  3 11:20 opt
drwxr-xr-x    2 root     root        4.0K Mar  3 11:20 run
drwxr-xr-x    3 root     root        4.0K Mar  3 11:20 spool
drwxrwxrwt    2 root     root        4.0K Mar  3 11:20 tmp
 ---> 5c16ef803390
Removing intermediate container ef432bd58499
Step 4/4 : CMD echo Hello world
 ---> Running in bf3a29014dfd
 ---> e2040343dfd9
Removing intermediate container bf3a29014dfd
Successfully built e2040343dfd9
russ in ~/Documents/Code/bootcamp/chapter06/debug-working on master*
⚡
```

Once it has built the temporary image is removed and we are left with our final image:

```
debug-working — -bash — 114×7
russ in ~/Documents/Code/bootcamp/chapter06/debug-working on master*
⚡ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
<none>              <none>              e2040343dfd9        About a minute ago   3.99 MB
alpine              latest              4a415e366388        3 days ago          3.99 MB
russ in ~/Documents/Code/bootcamp/chapter06/debug-working on master*
⚡
```

That was quite a simple example, but it should give you an idea of how to debug a more complex Dockerfile.

Monitoring containers

In the last section, we looked at how you can use the API built into Docker to gain an insight to what resources your containers are running by running the `docker container stats` and `docker container top` commands. Now, we are to see how we can take it to the next level by using **cAdvisor** from Google.

Google describes cAdvisor as follows:

> *cAdvisor (Container Advisor) provides container users an understanding of the resource usage and performance characteristics of their running containers. It is a running daemon that collects, aggregates, processes, and exports information about running containers. Specifically, for each container, it keeps resource isolation parameters, historical resource usage, histograms of complete historical resource usage, and network statistics. This data is exported by a container and is machine-wide.*

The project started off life as an internal tool at Google for gaining an insight into containers that had been launched using their own container stack.

Google's own container stack was called "Let Me Contain That For You" or lmctfy for short. The work on lmctfy has been installed as a Google port functionality over to libcontainer that is part of the Open Container Initiative. Further details on lmctfy can be found at `https://github.com/google/lmctfy/`

cAdvisor is written in Go (`https://golang.org`); you can either compile your own binary or you can use the pre-compiled binary that are supplied via a container, which is available from Google's own Docker Hub account. You can find this at `http://hub.docker.com/u/google/`.

Once installed, cAdvisor will sit in the background and capture metrics that are like that of the `dockercontainer stats` command. We will go through these stats and understand what they mean later in this chapter.

cAdvisor takes these metrics along with those for the host machine and exposes them via a simple and easy-to-use built-in web interface.

There are several ways to install cAdvisor; the easiest way to get started is to download and run the container image that contains a copy of a precompiled cAdvisor binary:

```
docker network create monitoring
docker container run -d \
  --volume=/:/rootfs:ro \
  --volume=/var/run:/var/run:rw \
  --volume=/sys:/sys:ro \
  --volume=/var/lib/docker/:/var/lib/docker:ro \
```

```
  --publish=8080:8080 \
  --name=cadvisor \
  google/cadvisor:latest
```

You should now have a cAdvisor container up and running on your host machine.

Before we start looking at stats, let's look at cAdvisor in more detail by discussing why we have passed all the options to the container.

The cAdvisor binary is designed to run on the host machine alongside the Docker binary, so by launching cAdvisor in a container, we are isolating the binary in its own environment. To give cAdvisor access to the resources it requires on the host machine, we have to mount several partitions and also give the container privileged access to let the cAdvisor binary think it is being executed on the host machine.

So now, we have cAdvisor running; what do we need to do to configure the service in order to start collecting metrics?

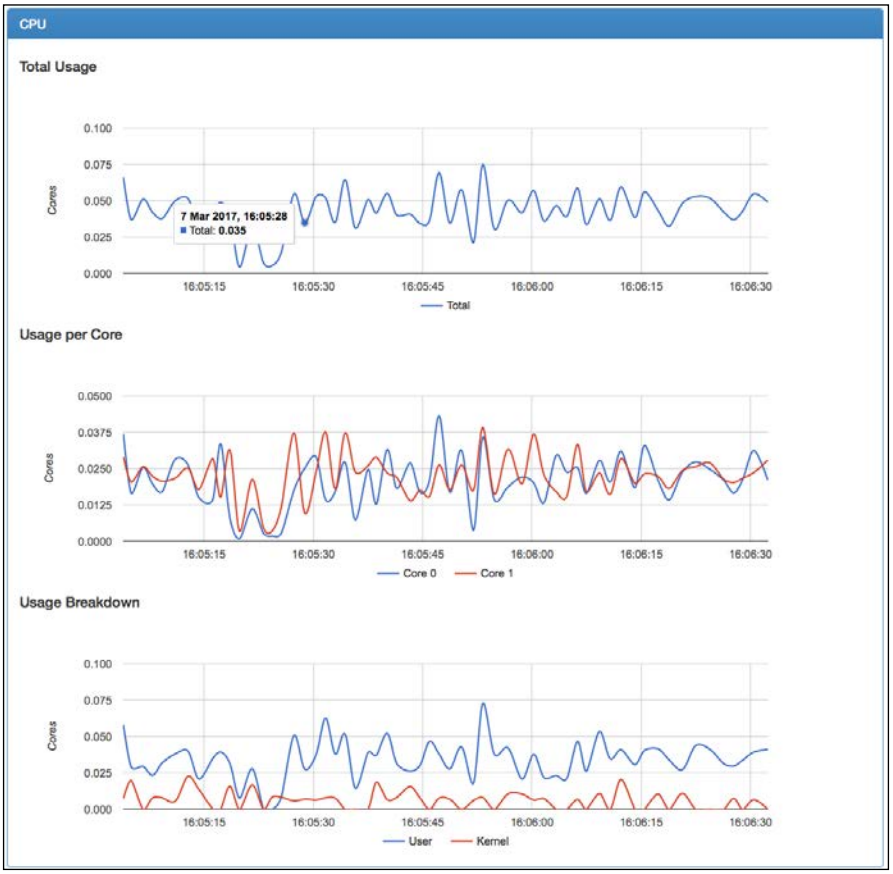
The short answer is, nothing at all. When you started the cAdvisor process, it instantly started polling your host machine to find out what containers are running and gathered information on both the running containers and your host machine.

cAdvisor should be running on the 8080 port; if you open `http://localhost:8080/`, you should be greeted with the cAdvisor logo and an overview of your host machine:

This initial page streams live stats about the host machine, though each section is repeated when you start to drill down and view the containers. To start with, let's look at each section using the host information.

The overview section gives you a bird's-eye view of your system; it uses gauges so you can quickly get an idea of which resources are reaching their limits. In the following screenshot, there is very little in the way of CPU utilization and the file system usage is relatively low; however, we are using 66% of the available RAM:

Next up is the graph which shows the CPU utilization over the last minute:



Here is what each term means:

- **Total Usage**: This shows an aggregate usage across all cores
- **Usage per Core**: This graph breaks down the usage per core
- **Usage Breakdown**: This shows aggregate usage across all cores, but breaks it down to what is being used by the kernel and what is being used by the user-owned processes

The Memory section is split into two parts. The graph tells you the total amount of memory used by all the processes for the host or container; this is the total of the hot and cold memory. The Hot memory is the current working set; pages that have been touched by the kernel recently. The Cold memory is the page that hasn't been touched for a while and could be reclaimed if needed.

The Usage Breakdown gives a visual representation of the total memory in the host machine, or allowance in the container, alongside the total and hot usage.

The network section shows the incoming and outgoing traffic over the last minute. You can change the interface using the drop-down box on the top-left.

There is also a graph that shows any networking errors. Typically, this graph should be flat. If it isn't, then you will be seeing performance issues with your host machine or container.

The final section, filesystem, gives a breakdown of the filesystem usage. In the following screenshot, `/dev/vda1` is the boot partition, overlay is the main filesystem running your running containers.

Now we can look at our containers. At the top of the page, there is a link of your running containers, it says **Docker Containers**; you can either click on the link or go directly to `http://localhost:8080/docker/`.

Once the page loads, you should see a list of all your running containers, and also a detailed overview of your Docker process, and finally a list of the images you have downloaded.

Subcontainers shows a list of your containers; each entry is a clickable link that will take you to a page that will give you the following details:

- **Isolation**:
    - **CPU**: This shows you the CPU allowances of the container; if you have not set any resource limits, you will see the host's CPU information

    - **Memory**: This shows you the memory allowances of the container; if you have not set any resource limits, your container will show an unlimited allowance

- **Usage**:
    - **Overview**: This shows gauges so you can quickly see how close to any resource limits you are
    - **Processes**: This shows the processes for just your selected container
    - **CPU**: This shows the CPU utilization graphs isolated to just your container
    - Memory: This shows the memory utilization of your container

The Driver status section gives the basic stats on your main Docker process, along with the information on the host machine's kernel, host name, and also the underlying operating system.

It also gives information on the total number of containers and images. You may notice that the total number of images is a much larger figure than you expected to see; this is because it is counting each file system as an individual image.

Finally, you get a list of the Docker images which are available on the host machine. It lists the Repository, Tag, Size, and when the image was created, along with the images' unique ID. This lets you know where the image originated from (Repository), which version of the image you have downloaded (Tag) and how big the image is (Size).

This is all great, what's the catch?

So, you are maybe thinking to yourself that all this information available in your browser is really useful; being able to see real-time performance metrics in an easily readable format is a real plus.

The biggest drawback of using the web interface for cAdvisor, as you may have noticed, is that it only shows you one minute's worth of metrics; you can quite literally see the information disappearing in real time.

As a pane of glass gives a real-time view into your containers, cAdvisor is a brilliant tool; if you want to review any metrics that are older than one minute, you are out of luck.

That is, unless you configure somewhere to store all your data; this is where Prometheus comes in.So what's Prometheus? Its developers describe it as follows:

> *Prometheus is an open-source system's monitoring and alerting toolkit built at SoundCloud. Since its inception in 2012, it has become the standard for instrumenting new services at SoundCloud and is seeing growing external usage and contributions.*

OK, but what does that have to do with cAdvisor? Well, Prometheus has quite a powerful database backend that stores the data it imports as a time series of events.

One of the things cAdvisor does, by default, is expose all the metrics it is capturing on a single page at `/metrics`; you can see this at `http://localhost:8080/metricson` our cAdvisor installation. The metrics are updated each time the page is loaded, you should see something like:

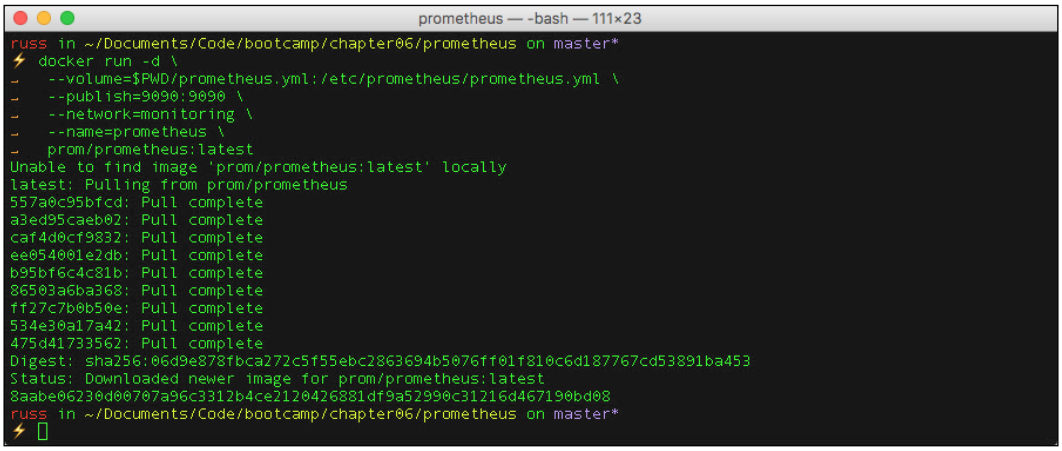
```
# HELP cadvisor_version_info A metric with a constant '1' value labeled by kernel version, OS version
# TYPE cadvisor_version_info gauge
cadvisor_version_info{cadvisorRevision="ae6934c",cadvisorVersion="v0.24.1",dockerVersion="17.03.0-ce"
# HELP container_cpu_system_seconds_total Cumulative system cpu time consumed in seconds.
# TYPE container_cpu_system_seconds_total counter
container_cpu_system_seconds_total{id="/"} 14.11
container_cpu_system_seconds_total{id="/docker"} 5.47
container_cpu_system_seconds_total{id="/rngd"} 0.06
container_cpu_system_seconds_total{id="/docker/3ad31d092a0ecd11c41109d733a149382630e48a38bc962be34ae3
container_cpu_system_seconds_total{id="/docker/9f159f5bc96fca5e4b5972bf6756ab247e8c3669b1605a39fe31c6
5.44
container_cpu_system_seconds_total{id="/docker/ce2774584d5bda66550316f73c0e4b5249576cab2e361e64c48953
# HELP container_cpu_usage_seconds_total Cumulative cpu time consumed per cpu in seconds.
# TYPE container_cpu_usage_seconds_total counter
container_cpu_usage_seconds_total{cpu="cpu00",id="/"} 29.728217281
container_cpu_usage_seconds_total{cpu="cpu00",id="/docker"} 20.751768636
container_cpu_usage_seconds_total{cpu="cpu00",id="/rngd"} 0.023016997
container_cpu_usage_seconds_total{cpu="cpu01",id="/"} 33.694437473
container_cpu_usage_seconds_total{cpu="cpu01",id="/docker"} 23.66852384
container_cpu_usage_seconds_total{cpu="cpu01",id="/rngd"} 0.083115237
container_cpu_usage_seconds_total{cpu="cpu00",id="/docker/3ad31d092a0ecd11c41109d733a149382630e48a38b
```

As you can see in the preceding screenshot, this is just a single long page of raw text. The way Prometheus works is that you configure it to scrape the `/metrics` URL at a user-defined interval, let's say every five seconds; the text is in a format that Prometheus understands and it is ingested into the Prometheus's time series database.

What this means is that, using Prometheus's powerful built-in query language, you can start to drill down into your data. Let's look at getting Prometheus up and running.

First of all, there is a work configuration file in the repo at `/bootcamp/chapter06/prometheus/`you will need to make sure you are in this folder as we are going to mounting the configuration file from within there:

```
docker container run -d \
  --volume=$PWD/prometheus.yml:/etc/prometheus/prometheus.yml \
  --publish=9090:9090 \
  --network=monitoring \
  --name=prometheus \
  prom/prometheus:latest
```



The configuration file we have launched Prometheus with looks like the following:

```
global:
scrape_interval: 15s # By default, scrape targets every 15 seconds.
external_labels:
    monitor: 'Docker Bootcamp'
scrape_configs:
  - job_name: 'cadvisor'
scrape_interval: 5s
static_configs:
      - targets: ['cadvisor:8080']
```

As we have launched our Prometheus container within the monitoring network our installation will be able scrape the metrics from `http://cadvisor:8080/`, also note that we haven't added `/metrics` to the URL as this added automatically by Prometheus.

Opening `http://localhost:9090/targets` in your browser should show you something like the following:

Also, the status menu has links to the following information pages:

- **Runtime information&Build information**: This displays how long Prometheus has been up and polling data, if you have configured an end point and details of the version of Prometheus that you have been running
- **Command-Line Flags**: This shows all the runtime variables and their values
- **Configuration**: This is a copy of the configuration file we injected into the container when it was launched
- **Rules**: This is a copy of any rules we injected; these will be used for alerting

As we only have a few containers up and running at the moment, let's launch one that runs Redis so we can start to look at the query language built into Prometheus.

We will use the official Redis image for this and as we are only going to use this as an example we won't need to pass it any user variables:

```
docker container run -d --name my-redis-server redis
```

We now have a container called my-redis-server running. cAdvisor should already be exposing metrics about the container to Prometheus; let's go ahead and see.

In the Prometheus web interface, go to the **Graph** link in the menu at the top of the page. Here, you will be presented with a text box into which you can enter your query. To start with, let's look at the CPU usage of the Redis container.

In the box, enter the following:

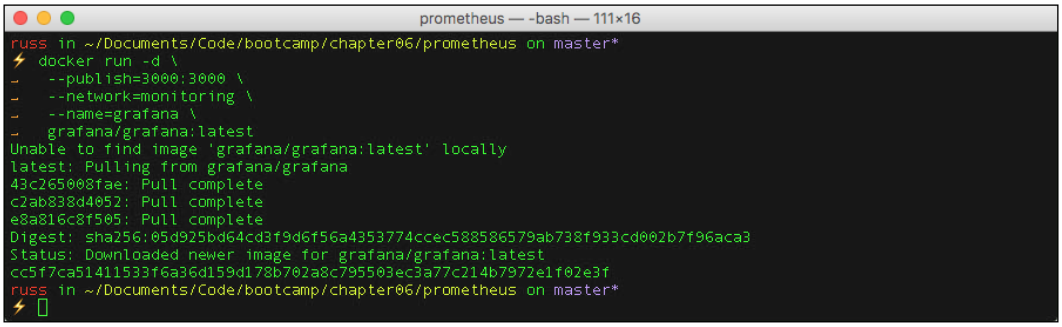
```
container_cpu_usage_seconds_total{job="cadvisor",name="my-redis-
server"}
```

Then, after clicking on Execute, you should have two results returned, listed in the Console tab of the page. If you remember, cAdvisor records the CPU usage of each of the CPU cores that the container has access to, which is why we have two values returned, one for `cpu00` and one for `cpu01`. Clicking on the **Graph** link will show you results over a period of time:

As you can see in the preceding screenshot, we now have access to the usage graphs for the last 5 minutes, which is about how long ago I launched the Redis instance before generating the graph.

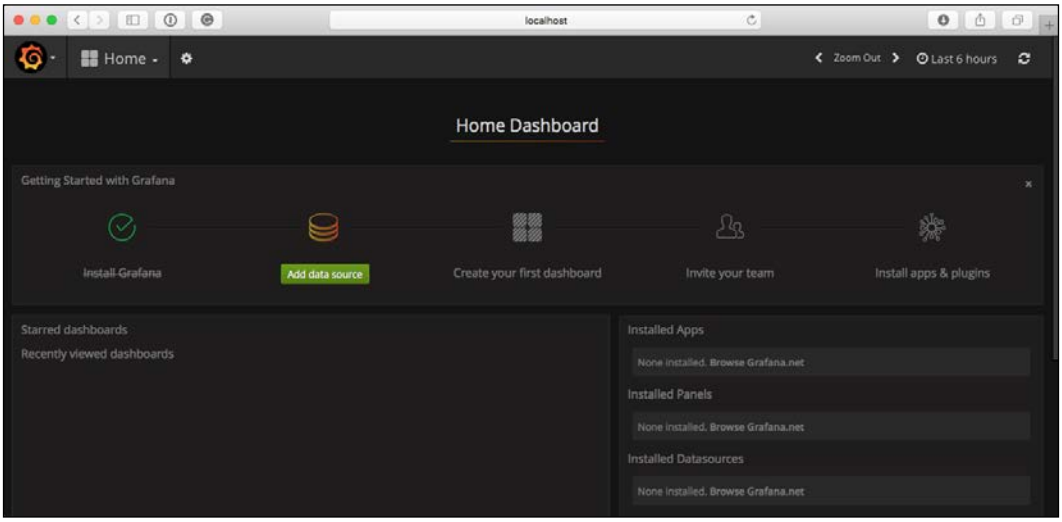
Graphing, as you may have noticed, isn't Prometheus's strong point. Luckily Grafana has been able to use Prometheus as a data source for a while, let's now launch a Grafana container:

```
docker container run -d \
  --publish=3000:3000 \
  --network=monitoring \
  --name=grafana \
grafana/grafana:latest
```



Once the container has launched, go to `http://localhost:3000/` in your browser and you will be prompted to login, the default username and password is `admin` / `admin`.

Now you are logged in you should see something like the following page:



As you may have guessed, we need to click **Add data source**and the add then enter the following information:

- **Name**: prometheus
- **Type**: <Select Prometheus from the drop down list>
- **Url**: `http://prometheus:9090`
- **Access**: <Select proxy from the drop down list>

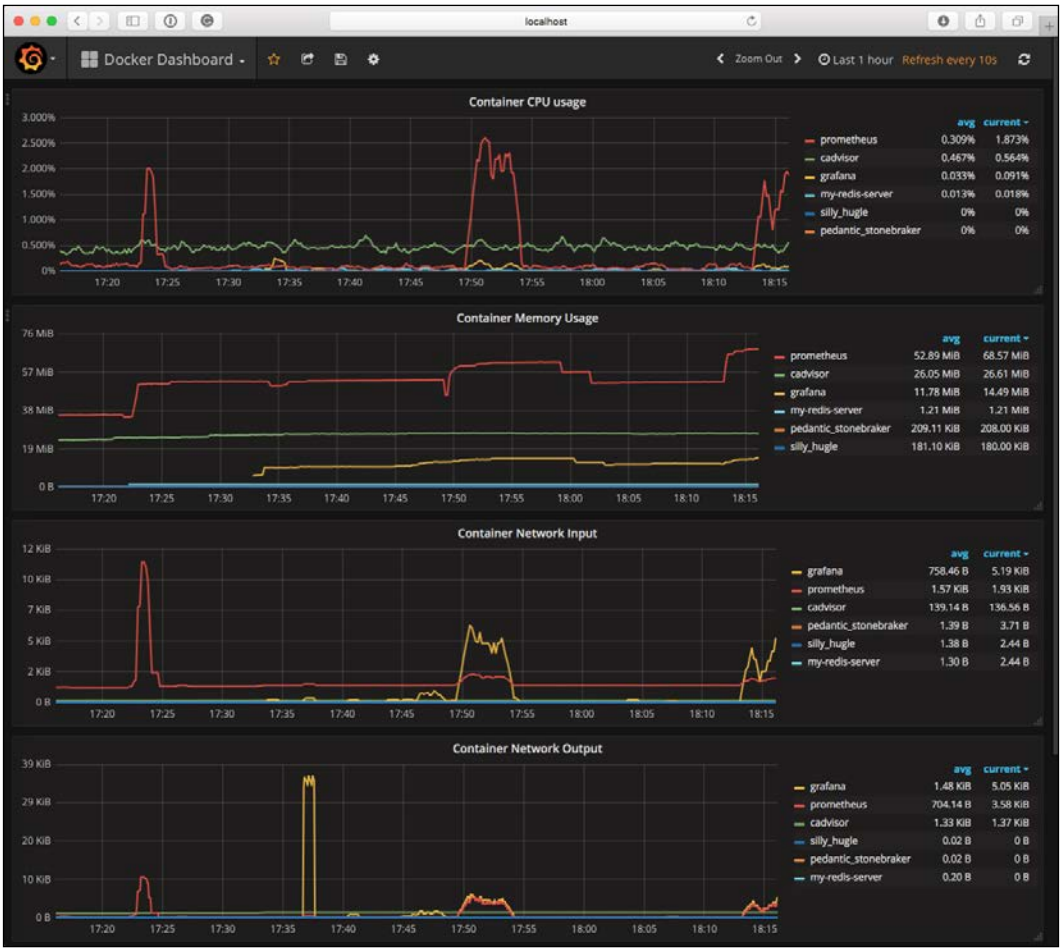
Leave everything else as is and then click on **Add**, after a second or two your data source should have successfully been added and the connection test passed.

Now that we have our data source added we can add a dashboard. There are plenty of dashboards available, we are going to use the one published by Brian Christner which can be found at `https://grafana.net/dashboards/179/`.

To import the dashboard,click on the **Grafana logo** in the top left, in the menu which opens go to **Dashboards** and then select **Import**. In the pop-up dialog which opens enter the URL for the dashboard, which is `https://grafana.net/dashboards/179/`, into the **Grafana.net Dashboard** box and then click on the **Load** button.

That will load the dashboard configuration, on the next page you given two options, the **Name** is already filled in so just select **prometheus** from the dropdown **Prometheus** list and click the **Import** button.

Once imported you should be greeted by a dashboard which looks similar (I have tweaked it for the screenshot) to the following:



You may notice from the screen above that we now have over an hour's worth of data from cAdvisor stored in Prometheus.

It's worth pointing that the current experimental build of Docker has a built-in Prometheus endpoint much in the same way that cAdvisor has. Once this hits the stable release I expect to see this be a great out of the box solution for monitoring your Docker hosts.

However, this is just one way you monitor your containers as there are numerous other tools both of which are self-hosted or run as a software as a service in the cloud such as:

- Sysdig - `http://www.sysdig.org/`
- Sysdig Cloud - `https://sysdig.com/`
- Datadog - `http://docs.datadoghq.com/integrations/docker/`
- New Relic - `https://newrelic.com/partner/docker`
- Coscale - `http://www.coscale.com/docker-monitoring`
- Elastic Metric Beat - `https://www.elastic.co/products/beats/metricbeat`

Summary

Hopefully now you should have an idea of where to start when it comes to looking into problems with containers, be it building them, checking logs, attaching to a container to further into issues or gathering performance metrics.

In the next chapter, which is also our last, we will look at some of the different scenarios and use cases for both Docker and techniques we have covered in this and previous five chapters.

7
Putting It All Together

In this, our final chapter, we will look at how we put everything we have learned in the previous chapters together along with how it could fit with your development and deployment workflows.

Also, we will talk about how to best describe Docker to others, typically you will find that people will assume that Docker containers are just like virtual machines. We will also look at what the benefits are along with some use cases.

Workflows

The first five chapters of this book work through a typical workflow for working with Docker containers through development all the way through to production:

- Local development & packaging (*Chapter 1*, *Installing Docker Locally* and *Chapter 2*, *Launching Applications Using Docker*)
- Staging and remote testing (*Chapter 3*, *Docker in the Cloud*)
- Production (*Chapter 4*, *Docker Swarm* and *Chapter 5*, *Docker Plugins*)
- On-going support (*Chapter 6*, *Troubleshooting and Monitoring*)

In our first few chapters we learned how to install and interact with Docker locally, typically when developing an application or software stack a developer or system administrator will test locally first.

Once the application / stack has been fully developed you can share it using the Docker Hub as both a public or private image, or if your image contains things you do now want to distribute via a third party you can host your own Docker Registry.

Once you have your packaged image, you may need other people to test it. As your image is available in a registry your colleagues or friends can pull your image and run it as you intended locally on their own machine without the worry of having to install and configure either your application or software stack.

If you need people to test remotely then you can spin up a Docker host on a public cloud provider and quickly deploy your application or software stack there.

Once everyone is happy you can deploy your application / software stack a servicein a multi-host cluster running Docker Swarm, this means that your service will be running in both a highly available and easy maintain environment. Deploying as a service will also allow you to easily roll out updates for your application or software stack using Swarms in-built service update features.

If you need share or persist storage between your containers or hosts then you can install one of the many volume plugins, likewise if you need to something more advanced than the multi-host networking provided by Swarm, no problem, replace it with a network plugin, remember "batteries included, but replaceable".

Finally, if you need to debug your images or running container you can use the commands and tools discussed in *Chapter 6*, *Troubleshooting and Monitoring*.

Describing containers

Compartmentalization that comprises both virtualization and containerization is the new normal for IT agility. Virtualization has been the enigmatic foundation for the enormous success of cloud computing. Now with the containerization idea becoming ubiquitous and usable, there is a renewed focus on using containers for faster application building, deployment, and delivery. Containers are distinctively fitted with a few game-changing capabilities and hence there is a rush in embracing and evolving the containerization technologies and tools.

Essentially a container is lightweight, virtualized, portable, and the software-defined environment in which software can run in isolation of other software running on the same physical host. The software that runs inside a container is typically a single-purpose application. Containers bring forth the much-coveted modularity, portability, and simplicity for IT environments. Developers love containers because they speed up the software engineering whereas operation team loves because they can just focus on runtime tasks such as logging, monitoring, lifecycle management and resource utilization rather than deployment and dependency management.

Describing Docker

Linux containers are hugely complicated and not user-friendly. Having realized the fact that several complexities are coming in the way of massively producing and fluently using containers, an open-source project got initiated with the goal of deriving a sophisticated and modular platform comprising an enabling engine for simplifying and streamlining various containers' lifecycle phases. That is, the Docker platform is built to automate the crafting, packaging, shipping, deployment and delivery of any software application embedded inside a lightweight, extensible, and self-sufficient container.

Docker is being positioned as the most flexible and futuristic containerization technology in realizing highly competent and enterprise-class distributed applications. This is to make deft and decisive impacts as the brewing trend in the IT industry is that instead of large monolithic applications distributed on a single physical or virtual server, companies are building smaller, self-defined and sustainable, easily manageable and discrete ones. In short, services are becoming microservices these days to give the fillip to the containerization movement.

The Docker platform enables artistically assembling applications from disparate and distributed components and eliminates any kind of deficiencies and deviations that could come when shipping code. Docker through a host of scripts and tools simplifies the isolation of software applications and makes them self-sustainable by running them in transient containers. Docker brings the required separation for each of the applications from one another as well as from the underlying host. We have been hugely accustomed to virtual machines that are formed through an additional layer of indirection in order to bring the necessary isolation.

This additional layer and overhead consumes a lot of precious resources and hence it is an unwanted cause for the slowdown of the system. On the other hand, Docker containers share all the resources (compute, storage and networking) to the optimal level and hence can run much faster. Docker images, being derived in a standard form, can be widely shared and stocked easily for producing bigger and better application containers. In short, the Docker platform lays a stimulating and scintillating foundation for optimal consumption, management, and maneuverability of various IT infrastructures

The Docker platform is an open-source containerization solution that smartly and swiftly automates the bundling of any software applications and services into containers and accelerates the deployment of containerized applications in any IT environments (local or remote systems, virtualized or bare metal machines, generalized or embedded devices, etc.). The container lifecycle management tasks are fully taken care of by the Docker platform. The whole process starts with the formation of a standardized and optimized image for the identified software and its dependencies. Now the Docker platform takes the readied image to form the containerized software. There are image repositories made available publicly as well as in private locations. Developers and operations teams can leverage them to speed up software deployment in an automated manner.

The Docker ecosystem is rapidly growing with a number of third-party product and tool developers in order to make Docker an enterprise-scale containerization platform. It helps to skip the setup and maintenance of development environments and language-specific tooling. Instead, it focuses on creating and adding new features, fixing issues and shipping software. Build once and run everywhere is the endemic mantra of the Docker-enabled containerization. Concisely speaking, the Docker platform brings in the following competencies.

- **Agility**: Developers have freedom to define environments and the ability to create applications. IT Operation team can deploy applications faster allowing the business to outpace competition.
- **Controllability**: Developers own all the code from infrastructure to application.
- **Manageability**: IT operation team members have the manageability to standardize, secure, and scale the operating environment while reducing overall costs to the organization.

Distinguishing Docker containers

Precisely speaking, Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run: source code, runtime, system tools, and system libraries (anything that can be installed on a server). This guarantees that the software will always run the same, regardless of its operating environment:

Containers running on a single machine share the same operating system kernel. They start instantly and use less RAM. Container images are constructed from layered filesystems and share common files, making disk usage and image downloads much more efficient.

- Docker containers are based on open standards. This standardization enables containers to run on all major Linux distributions and other operating systems such as Windows and macOS.

There are several benefits being associated with Docker containers as enlisted below.

- **Efficiency**: Containers running on a single machine all leverage a common kernel so they are lightweight, start instantly and make more efficient use of RAM.
    - **Resource sharing** among workloads allows greater efficiency compared to the use of dedicated and single-purpose equipment. This sharing enhances the utilization rate of resources
    - **Resource partitioning** ensures that resources are appropriately segmented to meet up the system requirements of each workload. Another objective for this partitioning is to prevent any kind of untoward interactions among workloads.
    - **Resource as a Service** (**RaaS**): Various resources can be individually and collectively chosen, provisioned and given to applications directly or to users to run applications.
- **Native Performance**: Containers guarantee higher performance due to its lightweight nature and less wastage
- **Portability**: Applications, dependencies, and configurations are all bundled together in a complete filesystem, ensuring applications work seamlessly in any environment (virtual machines, bare metal servers, local or remote, generalized or specialized machines, etc.). The main advantage of this portability is it is possible to change the runtime dependencies (even programming language) between deployments. Couple this with Volume plugins and your containers are truly portable.

- **Real-time Scalability**: Any number of fresh containers can be provisioned in a few seconds in order to meet up the user and data loads. On the reverse side, additionally provisioned containers can be knocked down when the demand goes down. This ensures higher throughput and capacity on demand. Tools like:
    - Docker Swarm
    - Kubernetes (`https://kubernetes.io/`)
    - Apache Mesos(`http://mesos.apache.org/`)
    - DC/OS (`https://dcos.io/`)

  To name but a few of the clustering solutions which further simplify elastic scaling
- **High Availability**: By running with multiple containers, redundancy can be built into the application. If one container fails, then the surviving peers – which are providing the same capability – continue to provide service. With orchestration, failed containers can be automatically recreated (rescheduled) either on the same or a different host, restoring full capacity and redundancy.
- **Maneuverability**: Applications running in Docker containers can be easily modified, updated or extended without impacting other containers in the host.
- **Flexibility**: Developers are free to use whichever programming languages and development tools they prefer.
- **Clusterability**: Containers can be clustered for specific purposes on demand and there are integrated management platforms for cluster-enablement and management.
- **Composability**: Software services hosted in containers can be discovered, matched for, and linked to form business-critical, process-aware and composite services.
- **Security**: Containers isolate applications from one another and the underlying infrastructure by providing an additional layer of protection for the application
- **Predictability**: With immutable images, the image always exhibits the same behavior everywhere because the code is contained in the image. That means a lot in terms of deployment and in the management of the application lifecycle.

- **Repeatability**: With Docker, one can build an image, test that image and then use that same image in production.
- **Replicability**: With containers, it is easy to instantiate identical copies of full application stack and configuration. These can then be used by new hires, partners, support teams, and others to safely experiment in isolation.

Virtual Machines versus containers

Containers quite drastically vary from the highly visible and viable **virtual machines** (**VMs**). Virtual machines represent hardware virtualization whereas containers facilitate operating system-level virtualization. Some literature points out that virtual machines are system or OS containers whereas containers typically stand for application containers.

On the functional side, containers are like VMs, but there are dissimilar in many other ways. Like virtual machines, containers too share the various system resources such as processing, memory, storage, etc. The key difference is that all containers in a host machine share the same OS kernel of the host operating system.

Though there is heavy sharing, containers intrinsically maintain a high isolation by keeping applications, runtimes, and other associated services separated from each other using the recently incorporated kernel features such as namespaces and cgroups.

On the resource provisioning front, application containers can be realized in a few seconds, whereas virtual machines often take a few minutes. Containers also allow direct access to device drivers through the kernel and this makes I/O operations faster.

Workload migration to nearby or faraway cloud environments can be accelerated with the containerization capability. The tools and APIs provided by the Docker container technology are very powerful and more developer-friendly than those available with VMs.These APIs allow the management of containers to be integrated into a variety of automated systems for accelerated software engineering.

The Docker use cases

Containerization is emerging as the way forward for the software industry as it brings forth a newer and richer way of building and bundling any kind of software, shipping and running them everywhere. That is the fast-evolving aspect of containerization promises and provides software portability, which has been a constant nuisance for IT developers and administrators for long decades now. The Docker idea is flourishing here because of a number of enabling factors and facets. This section is specially prepared for telling the key use cases of the Docker idea.

Integrating containers into workflows

Workflows are a widely accepted and used abstraction for unambiguously representing the right details of any complicated and large-scale business and scientific applications and executing them on distributed compute systems such as clusters, clouds, and grids. However, workflow management systems have been largely evasive on conveying the relevant information of the underlying environment on which the tasks inscribed in the workflow are to run. That is, the workflow tasks can run perfectly on the environment for which they were designed. The real challenge is to run the tasks across multiple IT environments without tweaking and twisting the source codes of the ordained tasks. Increasingly the IT environments are heterogeneous with the leverage of disparate **operating systems** (**OSes**), middleware, programming languages and frameworks, databases, etc. Typically workflow systems focus on data interchange between tasks and environment-specific. The same workflow, which is working fine in one environment, starts to crumble when it is being migrated and deployed on different IT environments. All kinds of known and unknown dependencies and incompatibilities spring up to denigrate the workflows delaying the whole job of IT setup, application installation and configuration, deployment, and delivery. Containers are the best bet for resolving this imbroglio once for all.

*Chao Zheng and Douglas Thain (Integrating Containers into Workflows: A Case Study Using Makeflow, Work Queue, and Docker)*has done a good job of analyzing several methods in order to experimentally prove the unique contributions of containers in empowering workflow / process management systems. They have explored the performance of a large bioinformatics workload on a Docker-enabled cluster and observed the best configuration to be locally managed containers that are shared between multiple tasks.

Docker for High-Performance Computing (HPC) and Technical Computing (TC) applications

(Douglas M. Jacobsen and Richard Shane Canon) – Currently containers are being overwhelmingly used for the web, enterprise, mobile and cloud applications. However, there are questions being asked and doubts being raised on whether containers can be a viable runtime for hosting technical and scientific computing applications. Especially there are many high-performance computing applications yearning for perfect a deployment and execution environment. The authors of this research paper have realized that Docker containers can be a perfect answer for HPC workloads.

In many cases, users desire to have the ability to easily execute their scientific applications and workflows in the same environment used for development or adopted by their community. Some researchers have tried out the cloud option but the challenges there are many. The users need to solve how they handle workload management, file systems, and basic provisioning. Containers promise to offer the flexibility of cloud-type systems coupled with the performance of bare-metal systems. Furthermore, containers have the potential to be more easily integrated into traditional HPC environments which mean that users can obtain the benefits of flexibility without the added burden of managing other layers of the system (i.e. batch systems, file systems, etc.).

Minh Thanh Chung and the team have analyzed the performance of virtual machines and containers for high-performance applications and benchmarked the results that clearly show containers are the next-generation runtime for HPC applications. In short, Docker offers many attractive benefits in an HPC environment. To test these, IBM Platform LSF and Docker have been integrated outside the core of Platform LSF and the integration leverages the rich Platform LSF plugin framework.

We all know that the aspect of compartmentalization is for resource partitioning and provisioning. That is, physical machines are subdivided into multiple logical machines (virtual machines and containers). Now on the reverse side, such kinds of logical systems carved out of multiple physical machines can be linked together to buildavirtual supercomputer to solve certain complicated problems. *Hsi-En Yu* and *Weicheng Huang* have described how they built a virtual HPC cluster in the research paper "*Building a Virtual HPC Cluster with Auto Scaling by the Docker*". They have integrated the auto-scaling feature of service discovery with the lightweight virtualization paradigm (Docker) and embarked on the realization of a virtual cluster on top of physical cluster hardware.

Containers for telecom applications

Csaba Rotter and the team has explored and published a survey article on the title "*Using Linux Containers in Telecom Applications*". Telecom applications exhibit strong performance and high availability requirements, therefore running them in containers requires additional investigations. A telecom application is a single or multiple node application responsible for a well-defined task. Telecom applications use standardized interfaces to connect to other network elements and implements standardized functions. On top of the standardized functions, a telecom application can have vendor-specific functionality. There is a set of **quality of service** (**QoS**) and **quality of experience** (**QoE**) attributes such as high availability, capacity, performance / throughput, etc. The paper has clearly laid out the reasons for the unique contributions of containers in having next-generation telecom applications.

Efficient Prototyping of Fault Tolerant Map-Reduce Applications with Docker-Hadoop (Javier Rey and the team) – Distributed computing is the way forward for compute and data-intensive workloads. There are two major trends. Data becomes big and there are realizations that big data leads to big insights through the leverage of pioneering algorithms, script and parallel languages such as Scala, integrated platforms, new-generation databases, and dynamic IT infrastructures. MapReduce is a parallel programming paradigm currently used to perform computations on massive amounts of data. Docker-Hadoop1, a virtualization testbed conceived to allow the rapid deployment of a Hadoop cluster. With Docker-Hadoop, it is possible to control the nodes characteristics and run scalability and performance tests that otherwise would require a large computing environment. Docker-Hadoop facilitates simulation and reproduction of different failure scenarios for the validation of an application.

Interactive Social Media Applications - AlinCalinciuc and the team has come out with a research publication titled as *OpenStack and Docker: building a high-performance IaaS platform for interactive social media applications*. It is a well-known truth that interactive social media applications face the challenge of efficiently provisioning new resources in order to meet the demands of the growing number of application users. The authors have given the necessary description on how Docker can run as a hypervisor, and how the authors could manage to enable for the fast provisioning of computing resources inside of an OpenStack IaaS using the nova-docker plug-in that they had developed.

Summary

At this point of time, Docker is nothing short of an epidemic and every enterprising business across the globe is literally obsessed with the containerization mania for their extreme automation, transformation, and disruption.

With the blossoming of hybrid IT, the role of Docker-enabled containerization is steadily growing to smartly empower IT-enabled businesses. In this chapter, we have discussed the prime capabilities and contributions of the Docker paradigm.

It is not often that you can summarize an entire book with a single meme, but I think that at the very least your journey into the world of containers will resolve this all too common problem:

Picture taken by Dave Roth

The days of developing code on version of a language with a configuration which is only local to a single developer which looks nothing like your production platform should now be over as you can easily develop, package and ship consistent containers which can run anywhere.

Index