Community Experience Distilled

Mastering Docker

Rethink what's possible with Docker—become an expert in the innovative containerization tool to unlock new opportunities in the way you use and deploy software

Scott Gallagher

[PACKT] PUBLISHING open source* community experience distilled

Mastering Docker

Rethink what's possible with Docker—become an expert in the innovative containerization tool to unlock new opportunities in the way you use and deploy software

Scott Gallagher

[PACKT] PUBLISHING
open source
community experience distilled

BIRMINGHAM - MUMBAI

Mastering Docker

First published: December 2015

Production reference: 1111215

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

ISBN 978-1-78528-703-9

www.packtpub.com

Credits

Author
Scott Gallagher

Reviewer
Tommaso Patrizi

Commissioning Editor
Edward Gordon

Acquisition Editor
Reshma Raman

Content Development Editor
Arshiya Ayaz Umer

Technical Editor
Ankita Thakur

Copy Editor
Akshata Lobo

Project Coordinator
Sanjeet Rao

Proofreader
Safis Editing

Indexer
Hemangini Bari

Graphics
Abhinash Sahu

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

About the Author

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

I would like to thank my family for their support not only while I worked on this book, but throughout my life and career. A special thank you goes to my wife, who is my soul mate, the love of my life, the most important person in my life, and the reason I push myself to be the best I can be each day. I would also like to thank my kids, who are the most amazing thing in this world; I truly am blessed to be able to watch them grow each day. And lastly, I want to thank my parents, who helped me become the person I am today.

About the Reviewer

Tommaso Patrizi is a Docker fan. He has been using the technology since its first releases, having machines in production with Docker since its version 0.6.0. He planned and deployed a basic private PaaS with Docker and Open vSwitch. He is an enthusiastic Ruby and Ruby on Rails coder. He is striving for simplicity as the perfect synthesis between code effectiveness, maintainability, and beauty. He is actually learning some functional tricks through Haskell.

Tommaso is a system administrator with broad OS (Microsoft Windows, Linux, and OS X), database (SQL Server, MySQL, PostgreSQL and PostGIS, and OrientDB), and virtualization and cloud (vSphere, VirtualBox, and Docker) knowledge.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

So hot off the presses, the latest buzz that has been on the tip of everyone's tongues and the topic of almost any conversation that includes containers these days is Docker! With this book, you will go from just being the person in the office who hears that buzz to the one who is tooting it around every day. Your fellow office workers will be flocking to you for anything related to Docker and shower you with gifts—well, maybe not gifts, but definitely tapping your brain for knowledge!

What this book covers

Chapter 1, *Docker Review*, will just be a review of Docker. If you are new to Docker, then this chapter will get you going for the future chapters. This chapter will cover the items you would see in the Docker command line as well as the purpose of Dockerfile and the contents that are contained inside it.

Chapter 2, *Up and Running*, will explain how to go from just reading the documentation and looking at the help contents of files to running some Docker commands. You will also learn how to create or build your own base containers, which will be the basis of all your future containers. Learn how to create and manage Docker volumes and how to pass environmental variables during the build process.

Chapter 3, *Container Image Storage*, will show the locations to store items such as Docker Hub and the Docker Hub Enterprise. What are the differences between the two. When should you use one over the other. It will help you answer these questions. Also, you'll learn how to set up automated image builds based off the code you have stored in places such as GitHub. What are the pieces you need to get all this set up and working.

Chapter 4, *Managing Containers*, will show how you can manage all the containers you have created and stored. In this chapter, the focus will be on using the command line. So, if you do decide to use a GUI application at a later time, you will understand what is happening in the background and also have a resource to fall back on if needed.

Chapter 5, *Docker Security*, covers security that has unfortunately become the main focus of not just systems administrators, but everyone involved in projects these days. What are the benefits of using containers over using traditional virtual machines. What is this new Docker security configuration tool that you can use to help you assist with your setup environments. What should you be looking out for? Dive in and let's take a look at it together!

Chapter 6, *Docker Machine*, talks about the future replacement of the boot2docker instance. Docker Machine is the future of creating your Docker Host environments. With Docker Machine, you can create the hosts of almost any environment from your local command line. You can create them to locally test in VMware Fusion or VirtualBox, or you can create some of them in cloud environments such as AWS, Azure, DigitalOcean, and many more. Come, learn how you can do this!

Chapter 7, *Docker Compose*, covers one of the most popular items when it comes to Docker—Docker Compose. So, what can you do with this magical tool? Docker Compose helps eliminate the "well it works just fine on my machine." With Compose, you can have the environments set up with all the resources tied together as you want them and hand them off to both the Dev side of the team as well as the Ops side. If it works for one person, it will work for others and vice versa. If something doesn't work, it will help you troubleshoot by replicating the issue with defined steps. You will learn how to use Compose to set up these environments as well as the file structure of the file that Compose references.

Chapter 8, *Docker Swarm*, is all about how you can cluster your containers together. With Docker Swarm, you can accomplish this task. You will learn how to install and set up these environments. By default, Docker Swarm uses HTTP for communication. You will learn how to set it up to use TLS for secure communication between all your cluster nodes and Swarm manager.

Chapter 9, *Docker in Production*, says it's time to deploy Docker in your production environment now that you have all the tools in your arsenal. But how do we go about doing this? Let's take a look at the first step on how to do this as well as monitor everything we have set up and running. You will learn items such as how to ensure containers restart when and if there was an error. Also, you will learn how extend to external platforms such as Heroku.

Chapter 10, *Shipyard*, will focus on one of the three GUI applications that you can utilize to set up and manage your Docker containers and images. We will do a complete walkthrough, from installation to every piece of the Shipyard UI. You will be able to see the benefits of using such a GUI to help manage your environment.

Chapter 11, *Panamax*, will focus on one of the three GUI applications that you can utilize to set up and manage your Docker containers and images. We will do a complete walkthrough, from installation to every piece of the Panamax UI. This will leave you with the ability to evaluate which GUI is right for your needs.

Chapter 12, *Tutum*, will focus on one of the three GUI applications that you can utilize to set up and manage your Docker containers and images. Tutum is the latest acquisition by Docker, so this software will only continue to evolve and become more baked into the Docker ecosystem. We will do a complete walkthrough, from installation to every piece of the Tutum UI.

Chapter 13, *Advanced Docker*, will explain some advance items such as:

- **Scaling Docker**: We'll look at how we can scale our environments.
- **Using discovery services**: We'll look at using discovery services to help scale our environments.
- **Debugging/Troubleshooting Docker**: We'll look at debugging and troubleshooting Docker issues that crop up.
- **Common issues and solutions**: We'll look at the common issues that are faced as well as the solutions to fix them.
- **Various Docker APIs**: We'll look at the Docker APIs that are out there and how to tie into them and use them to our advantage.
- **Keeping your containers in check**: We'll look at how we can keep our containers in check. If they fall out of check, how we can put them back in place.
- **Contributing to Docker**: We'll look at how we can contribute to Docker. If we can't contribute to the code, how we can help otherwise.
- **Advanced Docker networking**: We'll look at the future of Docker networking and what is coming next that will only enhance our environment.

What you need for this book

The book will walk you through the installation of any tool that you need. You will need a system with Windows, Mac OS, or Linux installed; preferably the latter of the three, as well as an Internet connection.

Who this book is for

The reader at the start of the book should be an experienced Linux developer with some understanding of the Linux filesystems as well as the concept of Linux Container Virtualization. They must have some experience developing services and applications. They should also have knowledge of the fundamentals of Docker, though we will re-establish these fundamentals in the first chapter or two for clarity.

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: "For example, in an Ubuntu-based system, if you want to install the Apache package, you would first do an `apt-get update` followed by an `apt-get install -y apache2`."

A block of code is set as follows:

```
master:
image:
scottpgallagher/galeramaster
hostname:
master
ports:
    - "3306:3306"
node1:
image:
scottpgallagher/galeranode
hostname:
    node1
links:
    - master
node2:
image:
scottpgallagher/galeranode
hostname:
    node2
links:
    - master
```

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

```
$ docker pull tutum/ubuntu
```

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: "You can search for prebuilt images on the Docker Hub and click on the **CREATE** button once you have found the one you want to use or test."

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.

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
Docker Review

Welcome to the *Mastering Docker* book! The first chapter will cover the Docker basics that you should already have a pretty good handle on. But if you don't already have the required knowledge at this point, this chapter will help give you the basics, so the future chapters don't feel as heavy. By the end of the book, you should be a Docker master able to implement Docker in your own environments, building and supporting applications on top of these environments.

In this chapter, we're going to review the following higher level topics with subtopics in each section:

- Understanding Docker
    - Docker versus typical VMs
    - The Dockerfile and its function
    - Docker networking/linking
- Docker installers/installation
    - Types of installers and how they operate
    - Controlling your Docker daemon
    - The Kitematic GUI
- Docker commands
    - Useful commands for Docker, Docker images, and Docker containers

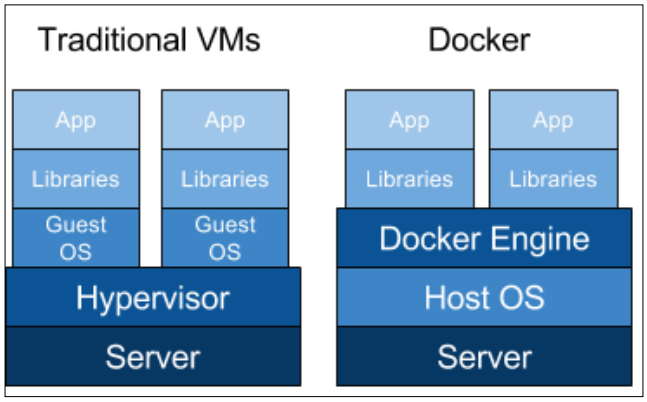
Understanding Docker

In this section, we will be covering the structure of Docker and the flow of what happens behind the scenes in this world. We will also take a look at Dockerfile and all the magic it can do. Lastly, in this section, we will look at the Docker networking/linking.

Difference between Docker and typical VMs

First, we must know what exactly Docker is and does. Docker is a container management system that helps easily manage **Linux Containers** (**LXC**) in an easier and universal fashion. This lets you create images in virtual environments on your laptop and run commands or operations against them. The actions you do to the containers that you run in these environments locally on your own machine will be the same commands or operations you run against them when they are running in your production environment. This helps in not having to do things differently when you go from a development environment like that on your local machine to a production environment on your server. Now, let's take a look at the differences between Docker containers and the typical virtual machine environments.

In the following illustration, we can see the typical Docker setup on the right-hand side versus the typical VM setup on the left-hand side:



This illustration gives us a lot of insight into the biggest key benefit of Docker, that is, there is *no need* for a complete operating system every time we need to bring up a new container, which cuts down on the overall size of containers. Docker relies on using the host OS's Linux kernel (since almost all the versions of Linux use the standard kernel models) for the OS it was built upon, such as Red Hat, CentOS, Ubuntu, and so on. For this reason, you can have almost any Linux OS as your host operating system (Ubuntu in the previous illustration) and be able to layer other OSes on top of the host. For example, in the earlier illustration, we could have Red Hat running for one app (the one on the left) and Debian running for the other app (the one on the right), but there would never be a need to actually install Red Hat or Debian on the host. Thus, another benefit of Docker is the size of images when they are born. They are not built with the largest piece: the kernel or the operating system. This makes them incredibly small, compact, and easy to ship.

Dockerfile

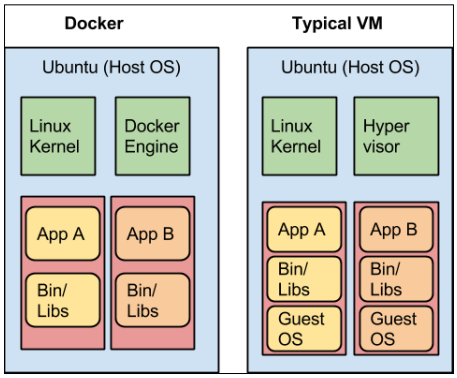
Next, let's take a look at the most important file pertaining to Docker: **Dockerfile**. Dockerfile is the core file that contains instructions to be performed when an image is built. For example, in an Ubuntu-based system, if you want to install the Apache package, you would first do an `apt-get update` followed by an `apt-get install -y apache2`. These would be the type of instructions you would find inside a typical Dockerfile. Items such as commands, calls to other scripts, setting environmental variables, adding files, and setting permissions can all be done via Dockerfile. Dockerfile is also where you specify what image is to be used as your base image for the build. Let's take a look at a very basic Dockerfile and then go over the individual pieces that make one up and what they all do:

```
FROM ubuntu:latest
MAINTAINER Scott P. Gallagher <email@somewhere.com>

RUN apt-get update && apt-get install -y apache2

ADD 000-default.conf /etc/apache2/sites-available/
RUN chown root:root /etc/apache2/sites-available/000-default.conf

EXPOSE 80
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

These are the typical items you would find in a basic Dockerfile. The first line states the image we want to start off with when we build the container. In this example, we will be using Ubuntu; the item after the colon can be called if you want a specific version of it. In this case, I am just going to say use the latest version of Ubuntu; but you will also specify `trusty`, `precise`, `raring`, and so on. The second line is the line that is relevant to the maintainer of Dockerfile. In this case, I just have my information in there; well, at least, my name is there. This is for people to contact you if they have any questions or find any errors in your file. Typically, most people just include their name and e-mail address. The next line is a typical line you will see while pulling updates and packages in an Ubuntu environment. You might think they should be separate and wonder why they should be put on the same line separated by `&&`. Well, in the Dockerfile, it helps by only having to run one process to encompass the entire line. If you were to split it into separate lines, it would have to run one process, finish the process, then start the next process, and finish it. With this, it helps speed up the process by pairing the processes together. They still run one after another, but with more efficiency. The next two lines complement each other. The first adds your custom configurations to the path you specified and changes the owner and group owner to the root user. The `EXPOSE` line will expose the ports to anything external to the container and to the host it is running on. (This will, by default, expose the container externally beyond the host, unless the firewall is enabled and protecting it.) The last line is the command that is run when the container is launched. This particular command in a Dockerfile should only be used once. If it is used more than once, the last CMD in the Dockerfile will be launched upon the container that is running. This also helps emphasize the one process per container rule. The idea is to spread out the processes so that each process runs in its own container, thus the value of the containers will become more understandable. Essentially, something that runs in the foreground, such as the earlier command to keep the Apache running in the foreground. If we were to use `CMD ["service apache2 start"]`, the container would start and then immediately stop. There is nothing to keep the container running. You can also have other instructions, such as `ENV` to specify the environmental variables that users can pass upon runtime. These are typically used and are useful while using shell scripts to perform actions such as specifying a database to be created in MySQL or setting permission databases. We will be covering these types of items in a later chapter, so don't worry about looking them up right now.

Docker networking/linking

Another important aspect that needs to be understood is how Docker containers are networked or linked together. The way they are networked or linked together highlights another important and large benefit of Docker. When a container is created, it creates a bridge network adapter for which it is assigns an address; it is through these network adapters that the communication flows when you link containers together. Docker doesn't have the need to expose ports to link containers. Let's take a look at it with the help of the following illustration:



In the preceding illustration, we can see that the typical VM has to expose ports for others to be able to communicate with each other. This can be dangerous if you don't set up your firewalls or, in this case with MySQL, your MySQL permissions correctly. This can also cause unwanted traffic to the open ports. In the case of Docker, you can link your containers together, so there is no need to expose the ports. This adds security to your setup, as there is now a secure connection between your containers.

We've looked at the differences between Docker and typical VMs, as well as the Dockerfile structure and the components that make up the file. We also looked at how Docker containers are linked together for security purposes as opposed to typical VMs. Now, let's review the installers for Docker and the structure behind the installation once they are installed, manipulating them to ensure they are operating correctly.

Docker installers/installation

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

- Apple OS X (Mac)
- Windows
- Linux (various Linux flavors)
- Cloud (AWS, DigitalOcean, Microsoft Azure, and so on)

Types of installers

With the various types of installers listed earlier, there are different ways Docker actually operates on the operating system. Docker natively runs on Linux; so if you are using Linux, then it's pretty straightforward how Docker runs right on your system. However, if you are using Windows or Mac OS X, then it operates a little differently, since it relies on using Linux. With these operating systems, they need Linux in some sort of way, thus enters the virtual machine needed to run the Linux part that Docker operates on, which is called boot2docker. The installers for both Windows and Mac OS X are bundled with the boot2docker package alongside the virtual machine software that, by default, is the Oracle VirtualBox.

Now, it is worthwhile to note that Docker recently moved away from offering boot2docker. But, I feel, it is important to understand the boot2docker terms and commands in case you run across anyone running the previous version of the Docker installer. This will help you understand what is going on and move forward to the new installer(s). Currently, they are offering up Docker Toolbox that, like the name implies, includes a lot of items that the installer will install for you. The installers for each OS contain different applications with regards to Docker such as:

Docker Toolbox piece	**Mac OS X**	**Windows**
Docker Client	X	X
Docker Machine	X	X
Docker Compose	X	
Docker Kitematic	X	X
VirtualBox	X	X

First, let's take a look at the older style commands of boot2docker. Then, we will take a look at the new commands or application that you can use to achieve these outcomes.

Controlling the Docker VM (boot2docker)

Now, there are ways to run boot2docker on different VM software. But to start off, VirtualBox is the best and easiest way to operate boot2docker:

```
$ boot2docker
Usage: boot2docker [<options>] {help|init|up|ssh|save|down|poweroff|reset
|restart|config|status|info|ip|shellinit|delete|download|upgrade|version}
[<args>]
```

Now, after we have installed Docker on Linux, OS X, or Windows, how do we go about controlling this virtual machine in the events when we need to start it up, restart it, or even shut it down? This is where the boot2docker command-line parameters come into play.

As you can see in the earlier illustration, there are a lot of options you can use for your boot2docker instance. The options you will use mostly are `up`, `down`, `poweroff`, `restart`, `status`, `ip`, `upgrade`, and `version`. Some of these commands you will use mostly to troubleshoot items when you are trying to see why the Docker commands might hang, or when you run into any other issues with your boot2docker virtual machine. You can see what each command does by executing the following command:

```
$ boot2docker help
```

The most useful command that I have found while troubleshooting is the boot2docker `status` command:

```
$ boot2docker status
```

Another useful boot2docker command is:

```
$ boot2docker version
```

This command will help see what version of boot2docker you are currently running. This is helpful in knowing when to use the boot2docker `upgrade` command. The last command we will look at with respect to boot2docker is the boot2docker `ip` command. This command is very useful when you need to know what IP address is to be used to access the machines you have been running on a particular host:

```
$ boot2docker ip
192.168.59.103
```

As you can see, the earlier command gives us the IP address of the boot2docker client running on my OS X machine inside VirtualBox. By using this IP, I can now access the containers I might have been running using the IP address alongside any of the open ports I have exposed.

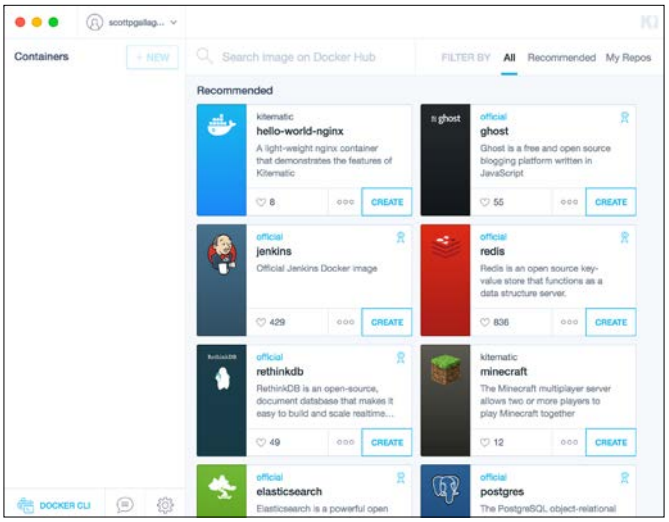
Docker Machine – the new boot2docker

So, with boot2docker on its way out, there needs to be a new way to do what boot2docker does. This being said, enter `Docker Machine`. With Docker Machine, you can do the same things you did with boot2docker, but now in Machine. The following table shows the commands you used in boot2docker and what they are now in Machine:

Command	boot2docker	Docker Machine
`command`	`boot2docker`	`docker-machine`
`help`	`boot2docker help`	`docker-machine help`
`status`	`boot2docker status`	`docker-machine status`
`version`	`boot2docker version`	`docker-machine version`
`ip`	`boot2docker ip`	`docker-machine ip`

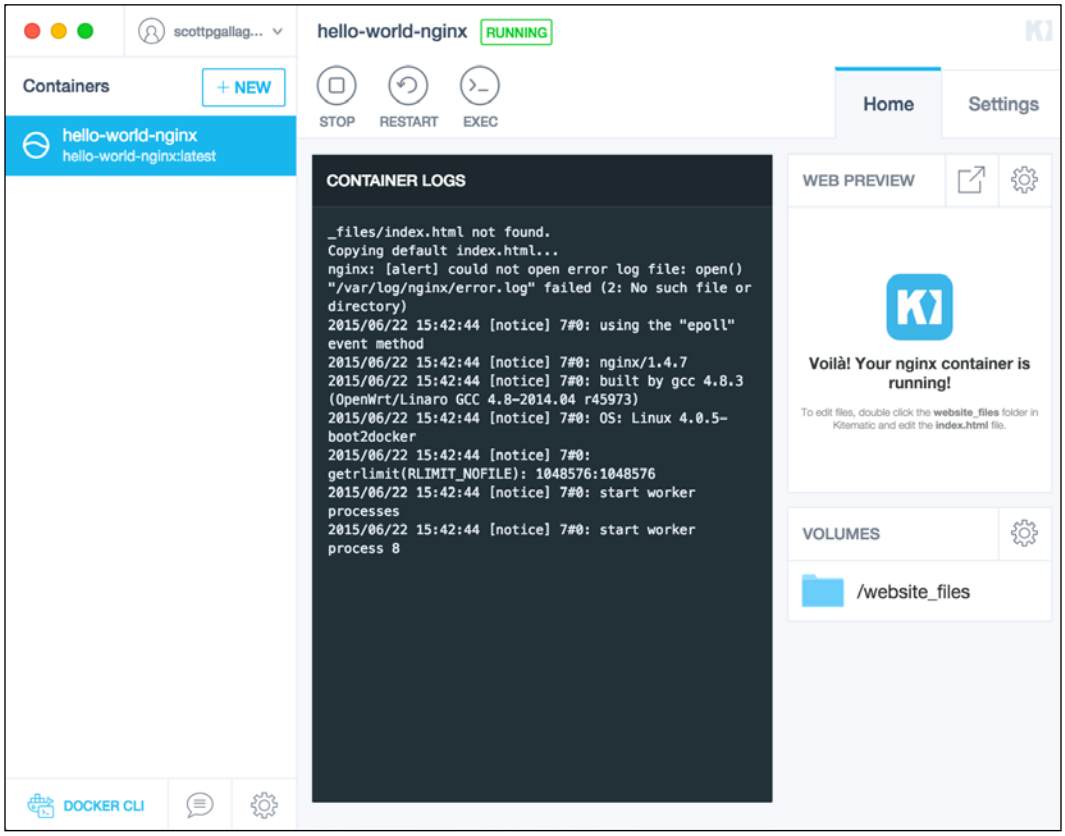
Kitematic

Now that we have covered all the basics of controlling your boot2docker VM, let's take a look at another way you can run Docker containers on your local machine. Let's take a look at **Kitematic**. Kitematic is a recent addition to the Docker portfolio. Up until now, everything we have done has been command line-based. With Kitematic, you can manage your Docker containers through a GUI. Kitematic can be used either on Windows or OS X, just not on Linux; besides who needs a GUI on Linux anyways! Kitematic, just like boot2docker, operates on a VM defaulting to VirtualBox. Pictures are worth a thousand words, so let's take a look at some screenshots of Kitematic:



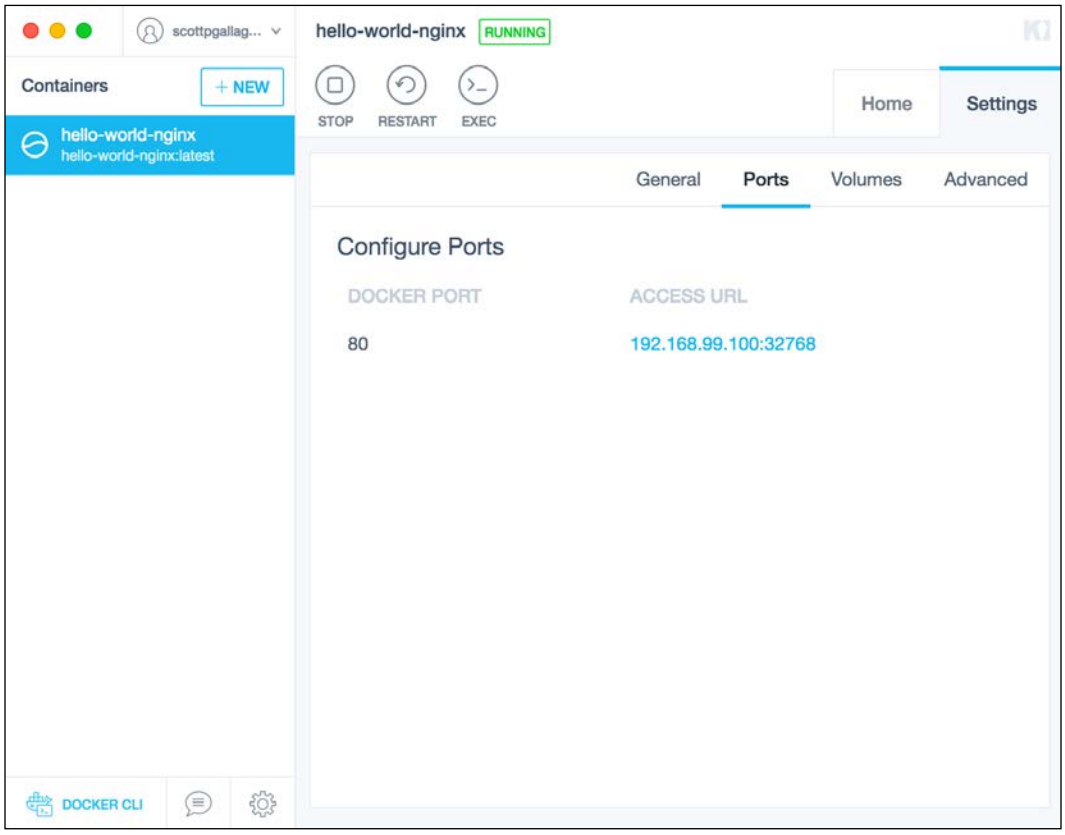
The previous screenshot depicts what you will see when you launch Kitematic for the first time.

After you start running the containers, they will show up on the left-hand side column. You can manipulate and get information about them through the GUI. You can search for prebuilt images on the Docker Hub and click on the **CREATE** button once you have found the one you want to use or test.

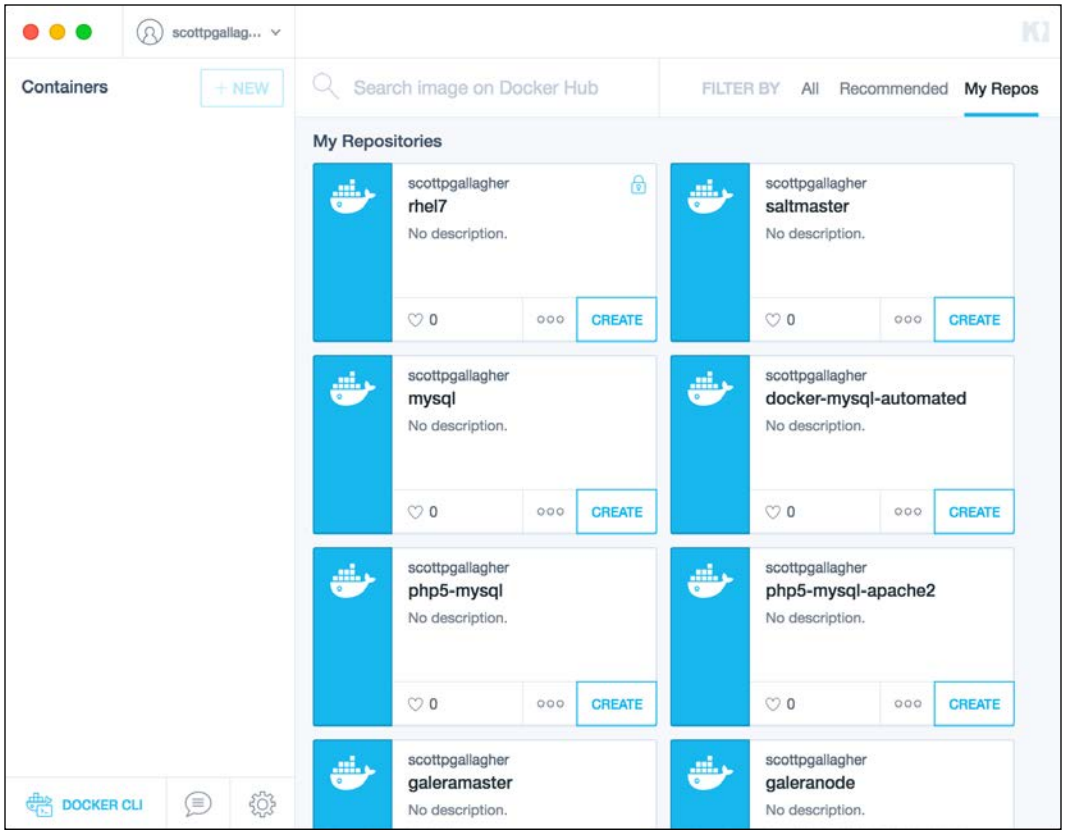


In the preceding screenshot, we have created and are running the `hello-world-nginx` image inside Kitematic. We can now use the **STOP**, **RESTART**, and **EXEC** commands against the container as well as view the settings of the running container.

In the following screenshot, we can go to settings and view what ports are exposed from the container to the outside:



In the following screenshot, you can see that you can use your login credentials to log in to the Docker Hub and view the repositories you have created and pushed there:



The Docker commands

We have covered the types of installers and what they can be run on. We have also seen how to control the Docker VM that gets created for you and how to use Kitematic. Let's look at some Docker commands that you should be familiar with already. We will start with some common commands and then take a peek at the commands that are used for the Docker images. We will then take a dive into the commands that are used for the containers.

The first command we will be taking a look at will be one of the most useful commands not only in Docker but in any command-line utility you use—the `help` command. It is run simply by executing the command as follows:

```
$ docker help
```

The earlier command will give you a full list of all the Docker commands at your disposal and a brief description of what each command does. For further help with a particular command, you can run the following:

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

You will then receive additional information on using the command, such as the switches, arguments, and descriptions of the arguments. Similar to the boot2docker `version` command we ran earlier, there is also a `version` command for the Docker daemon:

```
$ docker version
```

Now, this command will give us a little bit more information than the boot2docker command output, as follows:

```
Client version: 1.7.0
Client API version: 1.19
Go version (client): go1.4.2
Git commit (client): 0baf609
OS/Arch (client): darwin/amd64
Server version: 1.7.0
Server API version: 1.19
Go version (server): go1.4.2
Git commit (server): 0baf609
OS/Arch (server): linux/amd64
```

This is helpful when you want to see the version of the Docker daemon you may be running to see if you need/want to upgrade.

The Docker images

Next, let's take a dive into the Docker images. You will learn how to view the images you currently have that you can run, search for images on the Docker Hub, and pull them down to your environment, so you can run them. Let's first take a look at the `docker images` command. Upon running the command, we will get an output similar to the following output:

```
REPOSITORY          TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
ubuntu                  14.10               ab57dbafeeea        11 days
ago         194.5 MB
ubuntu                  trusty               6d4946999d4f        11 days
ago         188.3 MB
ubuntu                  latest               6d4946999d4f        11 days
ago         188.3 MB
```

Your output will differ based on whether you have any images at all in your Docker environment or upon what images you do have. There are a few important pieces you need to understand from the output you see. Let's go over the columns and what is contained in each. The first column you see is the `REPOSITORY` column; this column contains the name of the repository as it exists in the Docker Hub. If you were to have a repository that was from someone's user account, it may show up as follows:

```
REPOSITORY              TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
scottpgallagher/mysql   latest              57df9c7989a1        9 weeks
ago         321.7 MB
```

The next column, the `TAG` column, will show you different versions of a repository. As you can see in the preceding example with the Ubuntu repository, there are tag names for the different versions. So, if you want to specify a particular version of a repository in your Dockerfile (as we saw earlier), you are able to. This is useful, so you're not always reliant on having to use the latest version of an operating system and can use the one your application supports the best. It can also help you do backward compatibility testing for your application.

The next column is labeled `IMAGE ID` and it is based on a unique 64 hexadecimal digit string of characters. The image ID simplifies this down to the first 12 digits for easier viewing. Imagine if you had to view all 64 bits on one line! You will learn when to use this unique image ID for later tasks.

The last two columns are pretty straightforward; the first being the creation date for the repository, followed by the virtual size of the image. The size is very important as you want to keep or use images that are very small in size if you plan to be moving them around a lot. The smaller the image, the faster is the load time; and who doesn't like it faster?

Searching for the Docker images

Okay, so let's look at how we can search for the images that are in the Docker Hub using the Docker commands. The command we will be looking at is `docker search`. With the `docker search` command, you can search based on the different criteria you are looking for. For example, we can search for all the images with the term `ubuntu` in them and see what all is available. Here is what we would get back in our results; it would go as follows:

```
$ docker search ubuntu
```

We would get back our results:

```
NAME                              DESCRIPTION
STARS     OFFICIAL   AUTOMATED

ubuntu                            Ubuntu is a Debian-based Linux operating
s...         1835          [OK]

ubuntu-upstart                    Upstart is an event-based replacement for
...             26           [OK]

tutum/ubuntu                           Ubuntu image with SSH access. For the
root...           25                              [OK]

torusware/speedus-ubuntu  Always updated official Ubuntu docker imag...
25                                [OK]

ubuntu-debootstrap                debootstrap --variant=minbase
--components...         10              [OK]

rastasheep/ubuntu-sshd      Dockerized SSH service, built on top of of...
4                       [OK]

maxexcloo/ubuntu                 Docker base image built on Ubuntu with
Sup...            2                                [OK]

nuagebec/ubuntu                  Simple always updated Ubuntu docker
images...         2                               [OK]

nimmis/ubuntu                       This is a docker images different LTS
vers...       1                               [OK]

alsanium/ubuntu                   Ubuntu Core image for Docker
1                                 [OK]
```

Based on these results, we can now decipher some information. We can see the name of the repository, a reduced description, how many people have starred and think it is a good repository, whether it's an official repository; which means it's been approved by the Docker team, as well as if it's an automated build. An automated build is typically a Docker image that is built automatically when a Git repository it is linked to is updated. The code gets updated, the web hook is called, and a new Docker image is built in the Docker Hub. If we find an image we want to use, we can simply pull it using its repository name with the `docker pull` command, as follows:

```
$ docker pull tutum/ubuntu
```

The image will be downloaded and show up in our list when we perform the `docker images` command we ran earlier.

We now know how to search for Docker images and pull them down to our machine. What if we want to get rid of them? That's where the `docker rmi` command comes into play. With the `docker rmi` command, you can remove unwanted images from your machine(s). So, let's take look at the images we currently have on our machine with the `docker images` command. We will get the following:

```
REPOSITORY          TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
ubuntu                  14.10               ab57dbafeeea        11 days
ago         194.5 MB
ubuntu                  trusty               6d4946999d4f        11 days
ago         188.3 MB
ubuntu                  latest               6d4946999d4f        11 days
ago         188.3 MB
```

We can see that we have duplicate images here taking up space. We can see this by looking at the image ID and seeing the exact image ID for both `ubuntu:trusty` and `ubuntu:latest`. We now know that `ubuntu:trusty` is the latest Ubuntu image, so there is no need to keep them both around. Let's free up some space by removing `ubuntu:trusty` and just keeping `ubuntu:latest`. We do this by using the `docker rmi` command, as follows:

```
$ docker rmi ubuntu:trusty
```

If you issue the `docker images` command now, you will see that `ubuntu:trusty` no longer shows up in your images list and has been removed. Now, you can remove machines based on their image ID as well. But be careful while you do so; in this scenario, not only will you remove `ubuntu:trusty`, but you will also remove `ubuntu:latest` as they have the same image ID.

Manipulating the Docker images

We have gone over the images and know how to obtain and manipulate them in some ways. Next, we are going to take a look at what it takes to fire them up and manipulate them. This is the part where the images become containers! Let's first go over the basics of the `docker run` command and how to run containers. We will cover some basic `docker run` items in this section and more advanced `docker run` items in the later chapters. So, let's just look at how to get images up, running, and turned into containers. The most basic way to run a container is as follows:

```
$ docker run -i -t <image_name>:<tag> /bin/bash
```

Upon closer inspection of the earlier command, we start off with the `docker run` command, followed by two switches: `-i` and `-t`. The `-i` gives us an interactive shell into the running container, the `-t` will allocate a `pseudo-tty` that, while using interactive processes, must be used together with the `-i` switch. You can also use switches together; for example, `-it` is commonly used for these two switches. This will help you test the container to see how it operates before running it as a daemon. Once you are comfortable with your container, you can test how it operates in the daemon mode:

```
$ docker run -d <image_name>:<tag>
```

If the container is set up correctly and has an entry point setup, you should be able to see the running container by issuing the `docker ps` command. You will see something similar to the following:

```
$ docker ps

CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
cc1fefcfa098        ubuntu:14.10        "/bin/bash"         3 seconds ago
Up 3 seconds                            boring_mccarthy
```

Based on the earlier command, we get a lot of other important information indicating that the container is running. We can see the container ID, the image name that is running, the command that is running to keep the image alive, when the container started, its current status, if any ports were exposed they would be listed here, as well as the name given to the container. Now, these names are random, unless it is specified otherwise by the `--name= switch`. You can also the expose the ports on your containers by using the `-p` switch as follows:

```
$ docker run -d -p  <host_port>:<container_port> <image>:<tag>
$ docker run -d -p 8080:80 ubuntu:14.10
```

This will run the `ubuntu 14.10` container in the demonized mode, exposing port `8080` on the Docker host to port `80` on the running container:

```
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS                           NAMES
55cfdcb6beb6        ubuntu:14.10        "/bin/bash"         2 seconds ago
Up 2 seconds        0.0.0.0:8080->80/tcp   babbage
```

Now, there will come a time when containers don't want to behave. For this, you can see the issues you have by using the `docker logs` command. The command is very straightforward. You specify the container you want to see the logs off. For this command, you need to use the container ID or the name of the container from the `docker ps` output:

```
$ docker logs 55cfdcb6beb6
```

Or:

```
$ docker logs babbage
```

You can also get this ID when you first initiate the `docker run` command:

```
$ docker run -d ubuntu:14.10 /bin/bash
da92261485db98c7463fffadb43e3f684ea9f47949f287f92408fd0f3e4f2bad
```

Stopping containers

Now, let's take a look at how we can stop these containers. For various reasons, we would want to do this. There are a few commands we could use; they are `docker kill`, `docker stop`, `docker pause`, and `docker unpause`. Let's cover them briefly as they are fairly straightforward. First, let's look at the difference between `docker kill` and `docker stop`. The `docker kill` command will do just that—kill the container immediately. For a graceful shutdown of the container, you would want to use the `docker stop` command. Mostly, when you are testing, you will be using `docker kill`. When you're in your production environments, you will want to use `docker stop` to ensure you don't corrupt any data you might have in the Docker volumes. The commands are used exactly like the `docker logs` command, where you can use the container ID, the random name given to the container, or the one you might specify with the `--name= switch`.

Now, let's take a dive into how we can execute some commands, view information on our running containers, and manipulate them in a small sense. We will cover more about container manipulation in the later chapters as well. The first thing we want to take a look at, which will make things a little easier with the upcoming commands, is the `docker rename` command. With the `docker rename` command, we can change the name that has been randomly generated for the container. When we performed the `docker run` command, a random name was assigned to our container; most times, these names are fine. But if you are looking for an easy way to manage the containers, a name can be sometimes easier to remember. For this, you can use the `docker rename` command as follows:

```
$ docker rename <current_container_name> <new_container_name>
```

Now that we have an easily recognizable and rememberable name, let's take a peek inside our containers with the `docker stats` and `docker top` commands, taking them in order:

```
$ docker stats <container_name>
```

```
CONTAINER           CPU %               MEM USAGE/LIMIT     MEM %
NET I/O
web1                        0.00%               1.016 MB/2.099 GB   0.05%
0 B/0 B
```

The other command `docker top` provides a list of all running processes inside the container. Again, we can use the name of the container to pull the information:

```
$ docker top <container_name>
```

We will receive an output similar to the following one based on what processes are running inside the container:

```
UID                 PID                 PPID                C
STIME               TTY                 TIME                CMD
root                8057                1380                0
13:02               pts/0               00:00:00            /bin/bash
```

We can see who is running the process (in this case, the `root` user), the command being run (in this case, `/bin/bash`), as well as the other information that might be useful.

Lastly, let's cover how we can remove the containers. The same way we looked at removing images earlier with the `docker rmi` command, we can use the `docker rm` command to remove unwanted containers. This is useful if you want to reuse a name you provided to a container:

```
$ docker rm <container_name>
```

Summary

In this chapter, we have covered what basic information you should already know or now know for the chapters ahead. We have gone over the basics of what Docker is and how it is compared to typical virtual machines. We looked at the Dockerfile structure and the networking and linking of containers. We went over the installers, how they operate on different operating systems, and how to control them through the command line. We briefly looked at the latest Docker addition Kitematic for those interested in a GUI version for Windows or OS X. Then, we took a small but deep dive into the basic Docker commands to get you started.

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

2
Up and Running

I am very glad you decided to flip the page and come to *Chapter 2*, *Up and Running*! In this chapter, we will get you up and running with your own base images, storing those images, using custom environmental variables and scripts, and using Docker volumes. Here is a short review of what all we will be covering in this chapter:

- Dockerfile
- Docker build
- Build base image using the Dockerfile
- Docker Hub (basic overviews; more in depth will be covered in the next chapter)
- Environmental variables
- Docker volumes

Dockerfile

In this section, we will cover the Dockerfile from a more in-depth perspective than the previous chapter along with the best practices to use. By the end of the section, you will be structuring your Dockerfile in the most practical and efficient method. You will also be able to read and troubleshoot both yours and others' Dockerfile.

A short review of Dockerfile

In the previous chapter, we did a review of the Dockerfile and its content. We looked at something like this:

```
FROM ubuntu:latest
MAINTAINER Scott P. Gallagher <email@somewhere.com>

RUN apt-get update && apt-get install -y apache2

ADD 000-default.conf /etc/apache2/sites-available/
RUN chown root:root /etc/apache2/sites-available/000-default.conf

EXPOSE 80
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

We saw earlier and in this example as well the basic items that are inside a Dockerfile. The `FROM` and `MAINTAINER` fields have information on what image is to be used and who is the maintainer of that image. The `RUN` instruction can be used to fetch and install packages along with other various commands. The `ADD` instruction allows you to add files or folders to the Docker image. The `EXPOSE` instruction allows you to expose ports from the image to the outside world. Lastly, the `CMD` instruction executes the said command and keeps the container alive. Now that we did a really short review, let's take a more in-depth look at Dockerfile.

Reviewing Dockerfile in depth

Let's take a look at the following commands in depth:

- `LABEL`
- `ADD` or `COPY`
- `ENTRYPOINT`
- `ENTRYPOINT with CMD`
- `USER`
- `WORKDIR`
- `ONBUILD`

LABEL

The `LABEL` command can be used to add additional information to the image. This information can be anything from a version number to a description. You will want to combine labels into a single line whenever possible. It's also recommended that you limit the number of labels you use. Every time you use a label, it will add a layer to the image, thus increasing the size of the image. Using too many labels can cause the image to become inefficient as well. You can view the containers' labels with the `docker inspect` command:

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

ADD or COPY

Now, in the previous chapter and in the preceding Dockerfile example, we used the `ADD` instruction to add a file to a folder location. There is also another instruction you can use in your Dockerfile and that is the `COPY` instruction. You can use the `ADD` instruction and specify a URL straight to a file; it will be downloaded when the container is built. The `ADD` instruction will also unpack or untar a file when added. The `COPY` instruction is the same as the `ADD` instruction, but without the URL handling or the unpacking/untarring of files.

ENTRYPOINT

In the Dockerfile example, we used the `CMD` instruction to make the container executable and to ensure that it stays alive and running. You can also use the `ENTRYPOINT` instruction instead. The benefit of using `ENTRYPOINT` over `CMD` is that you can use them in conjunction with each other.

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

```
FROM ubuntu:latest
ENTRYPOINT ["ps", "-au"]
CMD ["-x"]
```

USER

The `USER` instruction lets you specify the username to be used when a command is run. The `USER` instruction can be used on the `RUN` instruction, the `CMD` instruction, or the `ENTRYPOINT` instruction in the Dockerfile.

WORKDIR

The `WORKDIR` command sets the working directory for the same set of instructions that the `USER` instruction can use (`RUN`, `CMD`, and `ENTRYPOINT`). It will allow you to use the `CMD` and `ADD` instructions as well.

ONBUILD

The `ONBUILD` instruction lets you stash a set of commands that will be used when the image is used again as a base image for a container. For example, if you want to give an image to developers and they all have a different code they want to test, you can use the `ONBUILD` instruction to lay the groundwork ahead of the fact of needing the actual code. Then, the developer will simply add their code in the directory you tell them and, when they run a new `docker build` command, it will add their code to the running image. The `ONBUILD` instruction can be used in conjunction with the `ADD` and `RUN` instructions:

```
ONBUILD ADD
ONBUILD RUN
```

Dockerfile – best practices

Now that we have covered the Dockerfile instructions in depth, let's take a look at the best practices of writing these Dockerfile:

- You should try to get in the habit of using a `.dockerignore` file. We will cover the `.dockerignore` file in the next section; it will seem very familiar if you are used to using a `.gitignore` file. It will essentially ignore the items you have specified in the file during the build process.
- Minimize the number of packages you need per image. One of the biggest goals you want to achieve while building your images is to keep them as small as possible. Not installing the packages that aren't necessary will greatly help in achieving this goal.
- Execute only one application process per container. Every time you need a new application, it is a best practice to use a new container to run that application in. While you can couple commands into a single container, it's best to separate them out.
- Sort commands as follows:
    - Sort them based upon the actual command itself, that is, run the following command:

      ```
      apt-get update && apt-get install -y
      ```

- Sort them alphabetically, so it's easier to change them later, that is, run the following command:

```
apt-get update && apt-get install -y \
     apache2 \
                          git \
                          memcached \
                          mysql
```

Docker build

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

The docker build command

Now that you have learned how to create and properly write a Dockerfile, it's time to learn how to take it from just a file to an actual image. There are a lot of switches that you can use while using the `docker build` command. So, let's use the always handy `--help` switch on the `docker build` command to view what all we can do:

```
 $ docker build --help

Usage: docker build [OPTIONS] PATH | URL | -

Build a new image from the source code at PATH

  -c, --cpu-shares=0    CPU shares (relative weight)
  --cgroup-parent=      Optional parent cgroup for the container
  --cpu-period=0        Limit the CPU CFS (Completely Fair Scheduler)
period
  --cpu-quota=0         Limit the CPU CFS (Completely Fair Scheduler)
quota
  --cpuset-cpus=        CPUs in which to allow execution (0-3, 0,1)
  --cpuset-mems=        MEMs in which to allow execution (0-3, 0,1)
  -f, --file=           Name of the Dockerfile (Default is 'PATH/
Dockerfile')
```

```
  --force-rm=false      Always remove intermediate containers
  --help=false          Print usage
  -m, --memory=         Memory limit
  --memory-swap=        Total memory (memory + swap), '-1' to disable
swap
  --no-cache=false      Do not use cache when building the image
  --pull=false          Always attempt to pull a newer version of the
image
  -q, --quiet=false     Suppress the verbose output generated by the
containers
  --rm=true             Remove intermediate containers after a successful
build
  -t, --tag=            Repository name (and optionally a tag) for the
image
```

Now, it may seem like a lot to digest, but the most important ones will be the `-f` and the `-t` switches. You can use the other switches to limit how much CPU and memory the build process will use. In some cases, you may not want the `build` command to take as much CPU or memory as it can have. The process may run a little slower, but if you are running it on your local machine or a production server and it's a long build process, you may want to set a limit. Typically, you don't use the `-f` switch as you run the `docker build` command from the same folder that the Dockerfile is in. Keeping the Dockerfile in separate folders helps sort the files and keeps the naming convention of the files the same.

.dockerignore

The `.dockerignore` file, as we discussed earlier, is used to exclude those files or folders we don't want include in the docker build. We also discussed placing the Dockerfile in a separate folder and the same applies for `.dockerignore`. It should go in the folder where the Dockerfile was placed. Keeping all the items you want to use in an image in the same folder will help you keep the items, if any, in the `.dockerignore` file to a minimum.

Building images using Dockerfile

The first way we are going to look at to build your base Docker images is by creating a Dockerfile, populating the Dockerfile with some instructions, and then executing a `docker build` command against them to get ourselves a base container. So, let's first start off by looking at a typical Dockerfile:

```
FROM ubuntu:latest
MAINTAINER Scott P. Gallagher <email@somewhere.com>

RUN apt-get update && apt-get install -y apache2

EXPOSE 80
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

In the preceding Dockerfile, the code is pretty straightforward. We are going to use the latest Ubuntu image and then run an `apt-get update` as well as an `apt-get install` of the Apache web server. We will set the container to expose port `80` when it is run and then start Apache in the foreground of the container.

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

```
$ docker build -f <path_to_Dockerfile> -t <REPOSITORY>:<TAG>
```

Now, `<REPOSITORY>` is typically the username you signed up for on Docker Hub and the `<TAG>` is a unique container name you want to provide:

```
$ docker build -f <path_to_Dockerfile> -t scottpgallagher:ubuntu_apache
```

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

```
$ docker build -t scottpgallagher:ubuntu_apache
```

The most important thing to remember is the `.`—the dot (or period) at the very end. This is to tell the `docker build` command to build in the current folder.

If you are using your own registry to push your images, then you can use any naming convention that you would like to use. But try to keep it simple and easy to identify by looking at the name.

Building a base image using an existing image

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

The other way of doing it is to use an already existing image that requires a little bit more work, but is essentially the same method. We would first need to get the base image we want:

```
$ docker pull ubuntu:latest
```

Then, we would run the container in the foreground, so we could add packages to it:

```
$ docker run -it ubuntu:latest /bin/bash
```

Once the container runs, you can add the packages as necessary by using the `apt-get` command in this case, or whatever the package manager commands are for your Linux flavor. After you have installed the packages you require, you need to save the container. To do so, you first need to get the container ID. You can do this in the following manner:

```
$ docker ps
```

Once you have the container ID, you can save (or commit) the container. So, to save this container, you need to do something similar to the following:

```
$ docker commit <container_ID> <REPOSITORY>:<TAG>
```

Now, if you are planning on using the Docker Hub (that we will be discussing here shortly in the next section of this chapter), you will want to structure your image names as follows:

```
$ docker commit <container_ID> <Docker_Hub_Username>:<Unique_Name>
$ docker commit <container_ID> scottpgallagher:ubuntu_apache2
```

Now, there will be some downfall to doing it this way. If you do it this way, you would need to create a Dockerfile in the `FROM` part and use the image you just created in this section. This is because you can't change what `CMD` or `ENTRYPOINT` is being used on an already built container. So, you would want to create a new Dockerfile and add in what `CMD` or `ENTRYPOINT` you might want to use.

Building your own containers

There are two ways to go about building your own containers. They are as follows:

- Using tar
- Using a scratch image

Using tar

So, you have a machine already running as a virtual machine or on a bare metal box and you want to convert that to a Docker image. How do you go about doing this? The first thing you will need to do is to install something like `debootstrap`:

```
$ sudo apt-get install -y debootstrap
```

Next, you will need to get the release name of the distribution of Linux you are running. To do this, we can look at the contents of the `/etc/lsb-release` file:

```
$ cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=14.04
DISTRIB_CODENAME=trusty
DISTRIB_DESCRIPTION="Ubuntu 14.04.2 LTS"
```

We can tell from the preceding output that we are running the trusty release of Ubuntu. Now, we can execute the next command using the newly installed `debootstrap` command:

```
$ sudo debootstrap trusty <unique_name> > /dev/null
```

We can execute the next command after the previous one is completed:

```
$ sudo tar -C <unique_name> -c . | sudo docker import - <unique_name>
```

The preceding command will switch to the directory you specify after `-C`, create a new archive from that directory based off the `-c` switch, and specify `.` (for the current directory). It will then import the image into a Docker image with the `docker import` command.

You can see this image by issuing the `docker images` command:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
ubuntu_trusty       latest              376bfebd75cb        17 minutes
ago      228.3 MB
```

You can then use the image for base images and share them on the Docker Hub or on your own Docker Registry. We will be covering how to push these images to various locations in the next section. First, though, we need to look at the other method to create images and that is to build from scratch.

If you wish to use something other than Ubuntu (or Debian), Docker has created scripts that you can utilize to create images from as well. You can check them out at `https://github.com/docker/docker/tree/master/contrib`.

You will want to look at the `mkimage-` files based on what distribution you are using.

Using scratch

You also have the option to build from scratch. Now, when you usually hear the term scratch, it literally means that you start from nothing. That's what we have here—you get absolutely nothing and have to build upon it. Now this can be a benefit because it will keep the image size very small; but it can also not be beneficial if you are fairly new to the Docker game, as it may be a little complicated.

Docker has done the hard work for us already and created an empty tar file that is on the Docker Hub named `scratch`; you can use it in the `FROM` section of your Dockerfile. You can base your entire Docker build on this then and add parts as needed. So, your Dockerfile might look something like this:

```
FROM scratch
ADD <script_to_add> /<path_to_add_to_on_container>
CMD ["/<path_to_add_to_on_container>"]
```

Docker Hub

In this section, we will cover the locations you can store the images you will be creating. There are several different areas to store these, ranging from a location in the cloud that can be set to public, where anyone can access and use them, to private, again a place in the cloud that can only be accessed by those you give permission to. You can also host your own repository, where you can store your own images. You can also purchase a Docker subscription (Docker Hub Enterprise) that provides you with what you need to deploy to the cloud or locally, and also comes along with commercial support from Docker.

The Docker Hub location

The Docker Hub is a location on the cloud, where you can store and share images that you have created. You can also link your images to the GitHub or Bitbucket repositories that can be built automatically based on web hooks. We will be discussing web hooks in the next chapter and will go over all the pieces required for that setup. There are two types of repositories on the Docker Hub: the public and private repositories. You can also roll your own repository that we will cover more in depth in the next chapter.

Pushing to a repository is very straightforward. Once you have the image built on your machine, there are two commands you need to run. One you will only have to run once and the other command you will use every time:

```
$ docker login
```

This will prompt you for your Docker Hub credentials and the e-mail address you are using on Docker Hub:

```
$ docker push <REPOSITORY>:<TAG>
```

This will show the progress of your push, kicking back to the command prompt when completed. You will then be able to see the image in either the command-line search or the web-based GUI search. By default, repositories are pushed as public. If you want to set them to private, you need to log in to the Docker Hub website and set the repository to **Make Private**. You can also mark images as unlisted, so they don't show up in the Docker searches. You can also mark them as listed at a later date as well.

Public repositories

Public repositories are those on the Docker Hub that are open to anyone. Anyone can use the `docker pull` command to download an image to their local system and run or build further images from it. You can also add collaborators to your public repositories and users can then push to that repository or update it. There are two ways you can search for images on Docker Hub:

- `$ docker search <TERM>`: You can search for terms such as `ubuntu` or a particular package you are looking to deploy such as `salt` or `mysql`
- The Docker Hub website (`https://registry.hub.docker.com/`): A simple web-based search with terms of your choosing

Private repositories

Private repositories are just that private. You can set permissions for different users from which the users can push, as we saw with public repositories and collaborators, but they can also pull all the images in that repository and don't have administrative rights. Once you are logged in to Docker Hub, you will be able to see all the private repositories that you have permission to, both in the web GUI and the command line.

Docker Hub Enterprise

There is also an option for Docker Hub Enterprise that allows you to deploy a Docker repository to your local system or cloud environment. Now, there is an option to run your own Docker repository based on a Docker image that is managed by Docker. What Docker Enterprise offers you is access to the software, access to updates/patches/security fixes, and support relating to issues with the software. The open source Docker repository image doesn't offer these services at this level; you are at the mercy of when that image will be updated on Docker Hub. Docker does offer various service levels for the said services that you can purchase through them. They currently are recommending you contact their sales department for any and all the pricing.

Environmental variables

In this section, we will cover the very powerful environmental variables or ENVs, as you will be seeing a lot of them. You can use environmental variables for a lot of things from your Dockerfile. If you are familiar with coding, these will probably come as secondhand to you. For others like myself, at first, they may seem intimidating; but don't get discouraged. They will be your best resource once you get the hang of them. They can be used from creating MySQL users, passwords, and databases to setting application items such as memory limits. We will cover some examples that you can use for future reference.

Using environmental variables in your Dockerfile

To use environmental variables in your Dockerfile, you can use the `ENV` instruction. The structure of the `ENV` instruction is:

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

Else, you can always use an equals sign between the two:

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

Now, the question is why do they have two and what are the differences? With the first example, you can only set one `ENV` per line. With the second `ENV` example, you can set multiple environmental variables on the same line:

```
ENV username=admin database=db1 tableprefix=pr2_
```

You can view what environmental variables are set on an image by using the `docker inspect` command:

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

You can change their values when you initialize the `docker run` command by using the `-e` or `--env` switch:

```
$ docker run -e username=superuser
$ docker run --env username=superuser
```

Now that we know how they need to be set in our Dockerfile, let's take a look at them in action. We will go over two examples in the next section showing the Dockerfile. We then set the corresponding scripts that will be used in the `RUN` instructions to execute and perform an action based off the `docker run` command that we will use after the image is built.

Don't get too confused; we will list out all the steps in the upcoming sections.

Creating a MySQL username, database, and setting permissions

First, we need a Dockerfile that specifies the MySQL username and database we want to use:

```
FROM ubuntu:latest
MAINTAINER Scott P. Gallagher <someone@email.com>
RUN apt-get update && apt-get install -y mysql mysql-server
ENV username mysqluser
ENV password pass
ENV database db2
ADD databasesetup.sh /
```

```
RUN chmod 644 /databasesetup.sh
RUN "/usr/bin/sh databasesetup.sh"
EXPOSE 3306
CMD ["/usr/bin/mysqld_safe"]
```

Now, we need to create the `databasesetup.sh` file that will be added and then called from the `RUN` instruction:

```
#!/bin/bash
/usr/bin/mysqld_safe
  mysql -uroot -e "CREATE USER '${username}'@'%' IDENTIFIED BY
'${password}'"
  mysql -uroot -e "GRANT ALL PRIVILEGES ON '${database}'.* TO
'${username}'@'%' WITH GRANT OPTION"
mysqladmin -uroot shutdown
```

Okay, what all have we done so far? We created our Dockerfile and `databasesetup.sh` file in a folder together. We can then run Docker build against the Dockerfile and it will create the image we want to use. Now, the last part is to start the container and insert the values we want to use. Note that the values you put in your Dockerfile are simply meant to be placeholders. You can execute your container with the values that are in there; but this is not recommended for production environments:

```
$ docker run -d -e username <value> -e password <value> -e database
<value> <REPOSITORY>:<TAG>
```

`<REPOSITORY>` and `<TAG>` will be the names you specified when you used the `docker build` command.

This should be a good boiler plate to use when you want to set something in a database. Next, let's take a look at an example where we want to set memory limits on a file that might already exist (that we add to the image).

Adding a file to the system

For this example, we are going to add our memcached configuration file to the system and, instead of specifying an actual value in the configuration file, we are going to set it to a variable. This will allow us to utilize that variable in our Dockerfile. After we have built the image, we will be able to give that variable a value with the `-e` switch. When the container starts up and starts up the memcached service, it will set the value for that memory limit to the stated value.

First, we need our Dockerfile:

```
FROM ubuntu:latest
MAINTAINER Scott P. Gallagher <someone@email.com>
RUN apt-get update && apt-get install -y memcached
ADD memcached /etc/default/
ENV MEMCACHESIZE 2048
EXPOSE 11211
CMD ["/usr/bin/memcached -u root"]
```

This is the memcached configuration file (named `memcached`) that will be added to the system:

```
# Set this to no to disable memcached.
ENABLE_MEMCACHED=yes
CACHESIZE=$MEMCACHESIZE
```

After the build is completed, we can run our image as follows:

```
$ docker run -d -e MEMCACHESIZE 1024 <REPOSITORY>:<TAG>
```

Again, set `<REPOSITORY>` and `<TAG>` to the values used while running the `docker build` command.

Now, we have seen how to build our own images from various methods. We took a look at where we can store our images once we are done building them. And we just took a look at environmental variables and two different ways of using them. Lastly, for this chapter, we will be looking at Docker volumes.

Docker volumes

In the last section of this chapter, we will cover container storage or Docker volumes as they are referred to. We will take a look at data volumes and data volume containers, the differences between the two, and when to use which one. Lastly, we will also look at the best practices for Docker volumes. This is the data that we want to be persistent or shared between containers. We need to remember that, by default, when you exit a running container, the data isn't saved. When you start the container backup, it will start in its initial state, so Docker volumes become incredibly important in areas like databases or filesystems.

Another switch that we will be covering is the `-v` or `--volume= switch`. This switch allows you to provide a volume to the Docker container that you wish contained persistent data. Remember that, when you start a Docker container, the data inside doesn't remain persistent unless you save it (or commit in Docker terms). The `volumes` switch allows you to have persistent data inside your Docker container such that even if the container is stopped or deleted, the data remains intact. Let's take a look at the two ways we can provide persistent volumes to containers:

- Data volumes
- Data volume containers

Data volumes

The first volume storage we will look at is data volumes. Data volumes are mounted inside the container when you run the container. However, as stated before, the volume is not tied to the container in events when it stops, is killed, or is deleted. Let's see how we first mount a volume inside a container; then we can dive a little deeper:

```
$ docker run -it -v /tmp ubuntu /bin/bash
```

We are simply running an `ubuntu` container shelled into `/bin/bash`, so we can see the `/tmp` volume mounted. This will create a new volume inside the container at the specified path. Essentially, it overwrites or hides the folder inside the container if it does exist; and in our case, `/tmp` already exists, so any data the container might have had inside it is no longer there and `/tmp` will now be an empty folder or volume.

You can also use multiple `-v` volume switches on a single `docker run` line:

```
$ docker run -it -v /tmp -v /data ubuntu /bin/bash
```

It is nice to use the `-it` switch sometimes, so you can actually see how this works. In later times, you will want to be running your containers with the `-d` switch, so they are not running the foreground.

Now, you can also mount the directory from the local machine the Docker containers are running on into the Docker container. To do so, you can use the `-v` switch again, but you need to add `:/<path>` to the path:

```
$ docker run -it -v /tmp:/data ubuntu /bin/bash
```

This will mount the contents of `/tmp` (on the Docker host) to the `/data` directory inside the now running Docker container. If you were to look at the contents of `/tmp` on the Docker host and the contents of `/data` on the running Docker container, you will see that they match. Any changes you make inside the Docker containers `/data` folder will be reflected in the Docker host's `/tmp` folder.

By default, when you mount a directory from a Docker host to a Docker container, it will mount in the read/write mode. There is a way you can mount it in the read-only mode as well. Again, using the `-v` switch, we will just append `:ro` to our `volume` instruction:

```
$ docker run -it -v /tmp:/data:ro ubuntu /bin/bash
```

You can locate one or several volumes on a Docker container by using the `docker inspect` container:

```
$ docker inspect <CONTAINER_ID>
```

The line(s) you will be looking for will resemble the following:

```
    "Volumes": {
        "/tmp": "/mnt/sda1/var/lib/docker/volumes/5c4e1bff167ea1479dd9f33
f74aeaf5d7f9f4d252d096e95e87befdb9be23ea0/_data"
```

Remember, you can get the container ID by running:

```
$ docker ps
```

The preceding output shows how the `docker inspect` command actually works. It is mounting `/tmp` inside the container; but where does the data actually live? The data actually lives in the machine your container runs on in the path specified. If you were to populate data inside the container in the `/tmp` folder and then navigate from the machine running the Docker container to the `/mnt/sda1/var/lib/docker/volumes/5c4e1bff167ea1479dd9f33f74aeaf5d7f9f4d252d096e95e87befdb9be23ea0/_data` directory, the data would be there. Now, we will go into the details of how to manage data and move it around between Docker hosts in the next chapter.

On a side note, you can also use the `VOLUME` instruction inside the Dockerfile to specify volumes for a container. It would look similar to this:

```
FROM ubuntu:latest
MAINTAINER Scott P. Gallagher <someone@email.com>
VOLUME ["/datastore"]
```

You can also use the `-v` flag to mount a single file into a container. So, the discussion isn't just about directories, it's about files as well. Now, we have seen how we can use `Volumes` to create persistent data that is stored inside containers; but what other options do we have with regards to using `Volumes`? We can use data volume containers too.

Data volume containers

Data volume containers come in handy when you have data that you want to share between containers. There is another flag we can utilize on the `docker run` command. Let's take a look at the `--volumes-from` switch.

What we will be doing is using the `-v` switch on one of our Docker containers. Then, our other containers will be using the `--volumes-from` switch to mount the data to the containers that they run.

First step, let's fire up a container that has a data volume we can add to other containers.

For this example, we will be using the `busybox` image since it's very small in size. We are also going to use the `--name` switch to give the container a name that can be used later:

```
$ docker run -it -v /data --name datavolume busybox /bin/sh
```

We are going to create a volume and mount it in `/data` inside our container. We have also named our container `datavolume` so that we can leverage in our `--volumes-from` switch. While we're still inside the shell, let's add some data to the `/data` directory. So, when we mount it on the other systems, we know it's the right one:

```
$ touch /data/correctvolume
```

This will create the `correctvolume` file inside the `/data` directory in the `busybox` container we are running.

Now, we need to connect some containers to this `/data` directory in the container. This is where the name we gave it will come in handy:

```
$ docker run -it --volumes-from datavolume busybox /bin/sh
```

If we now perform `ls /data`, we should see the `correctvolume` file that we created earlier.

Something to note here is that when you use the `--volumes-from` switch, the directory will be mounted in the same place on both the containers. You can also specify multiple `--volumes-from` switches on a single command line.

There will come a time when you run into the following error:

```
$ docker run -it -v /data --name datavolume busybox /bin/bash
Error response from daemon: Conflict. The name "data" is already in use
by container 82af96592008. You have to delete (or rename) that container
to be able to reuse that name.
```

You can remove the volume if you want, but **USE IT CAUTIOUSLY**, as once you remove the volume, the data inside that volume will go away with it:

```
$ docker rm -v data
```

You can also use this to clean up the volumes that you no longer want on the system. But again, use extreme caution as stated before that once a volume is gone, the data will go with it.

Docker volume backups

It is important to remember that while your containers are immutable, the data inside your volumes is mutable. It changes, while the items inside your Docker containers do not. For this reason, you need to make sure that you are backing up your volumes in some manner.

Volumes are stored on the system at `/var/lib/docker/volumes/`.

The key to remember here is that the volumes are not named the way you named them in this directory. They are given unique hash values, so understanding what content is in them can be confusing if you are just looking at their name. If you are looking at managing volumes at this point, I would highly recommend this image from the Docker Hub: `https://hub.docker.com/r/cpuguy83/docker-volumes/`.

This container (once built) will allow you to list volumes as well as export them into a tarred up file.

Summary

In this chapter, we have looked at an in-depth view of the Dockerfile and the best practices to write them, the `docker build` command and the various ways we can build the said containers, and the various Docker Hubs to store the containers you have built. We also learned about the environmental variables that you can use to pass from your Dockerfile to the various items inside your containers and Docker volumes to store persistent or shared data.

Let's do a quick review of all the commands we have learned in this chapter.

- `docker inspect`: To inspect a running container
- `docker build`: To build a new image from a Dockerfile
- `docker login`: To login to the Docker Hub
- `docker commit`: To commit changes to a running container
- `docker search`: To search the Docker Hub from the command line
- `docker push`: To push a new image or changes to existing changes to the Docker Hub
- `docker run -e`: To run a new container and specify an environmental variable value
- `docker run -v`: To run a Docker container and mount a persistent volume inside it
- `docker run --volumes-from`: To mount a volume from an already running container inside this new container

In the next chapter, we will be taking a more in-depth look at the various Docker Hubs and a good look at web hooks that you can use to do automated builds. We will cover all the pieces required for these web hooks as well, and go through the process step by step. We will also look at the Docker Registry that is open sourced, so you can roll your own place to store images without the fees of Docker Enterprise.

3
Container Image Storage

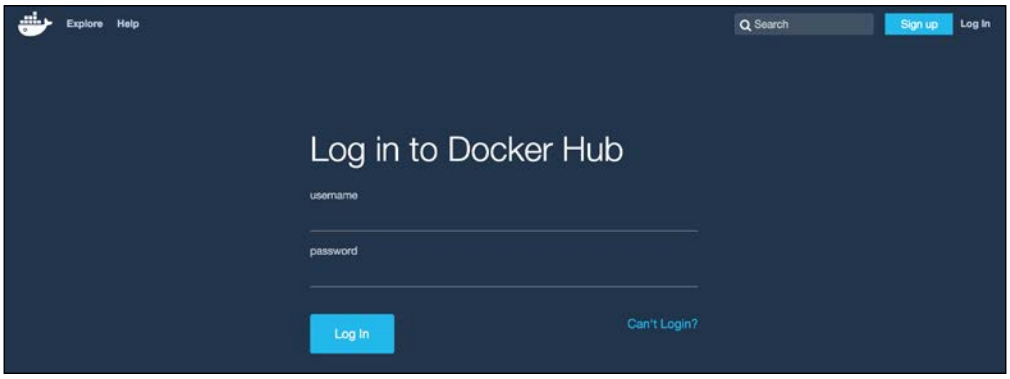
In the third chapter of the book, we will cover the places you store your containers, such as Docker Hub and Docker Hub Enterprises. We will also cover Docker Registry that you can use to run your own local storage for the Docker containers. We will review the differences between them all and when and how to use each of them. It will also cover how to set up automated builds using web hooks as well as the pieces that are all required to set them up. Lastly, we will run through an example of how to set up your own Docker Registry. Let's take a quick look at the topics we will be covering in this chapter:

- Docker Hub
- Docker Hub Enterprise
- Docker Registry
- Automated builds

Docker Hub

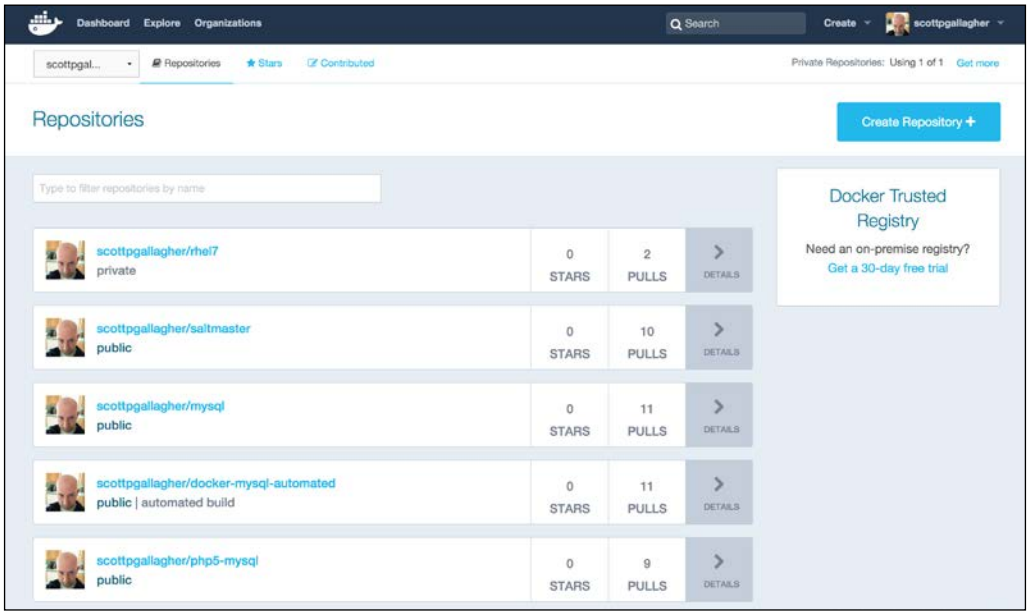
We will be covering Docker Hub in a little more detail than what we looked at in the previous chapter. In *Chapter 2, Up and Running*, we just glazed over Docker Hub as a storage location to push our images to. In this section, we will focus on that Docker Hub, which is a free public option, but also has a private option that you can use to secure your images. We will focus on the web aspect of Docker Hub and the management you can do there.

The login page is like the one shown in the following screenshot:



Dashboard

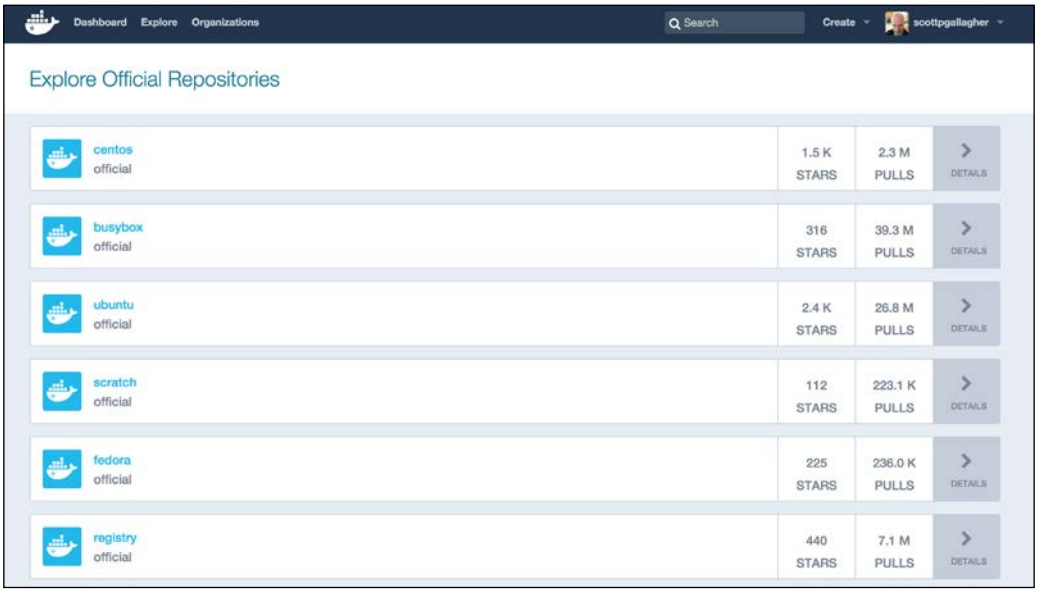
After logging into the Docker Hub, you will be taken to the following landing page. This page is known as the **Dashboard** of Docker Hub.



From here, you can get to all the other subpages of Docker Hub. In the upcoming sections, we will go through everything you see on the dashboard, starting with the dark blue bar you have on the top.

Explore the repositories page

The following is the screenshot of the **Explore** link you see next to **Dashboard** at the top of the screen:

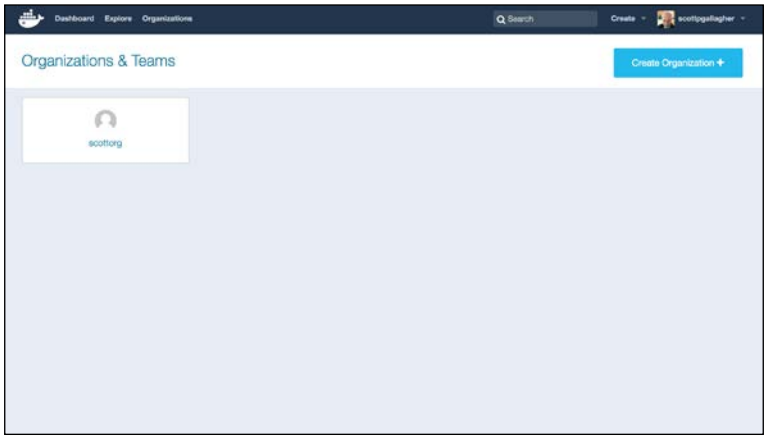


As you can see in the screenshot, this is a link to show you all the official repositories that Docker has to offer. Official repositories are those that come directly from Docker or from the company responsible for the product. They are regularly updated and patched as needed.

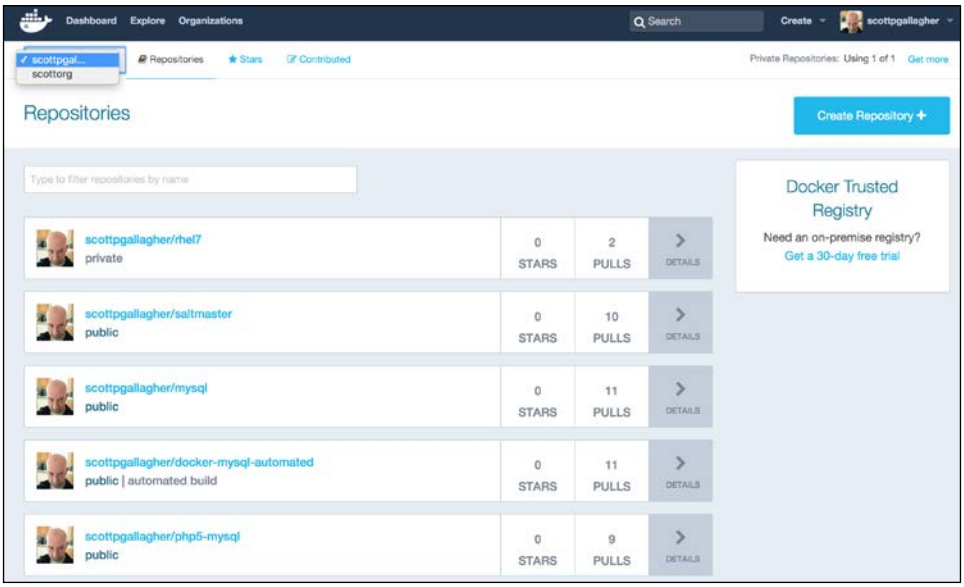
Organizations

Organizations are those that you have either created or have been added to. **Organizations** allow you to layer on control, for say, a project that multiple people are collaborating on.

The organization gets its own setting such as whether to store repositories as public or private by default, changing plans that will allow for different amounts of private repositories, and separate repositories all together from the ones you or others have.



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



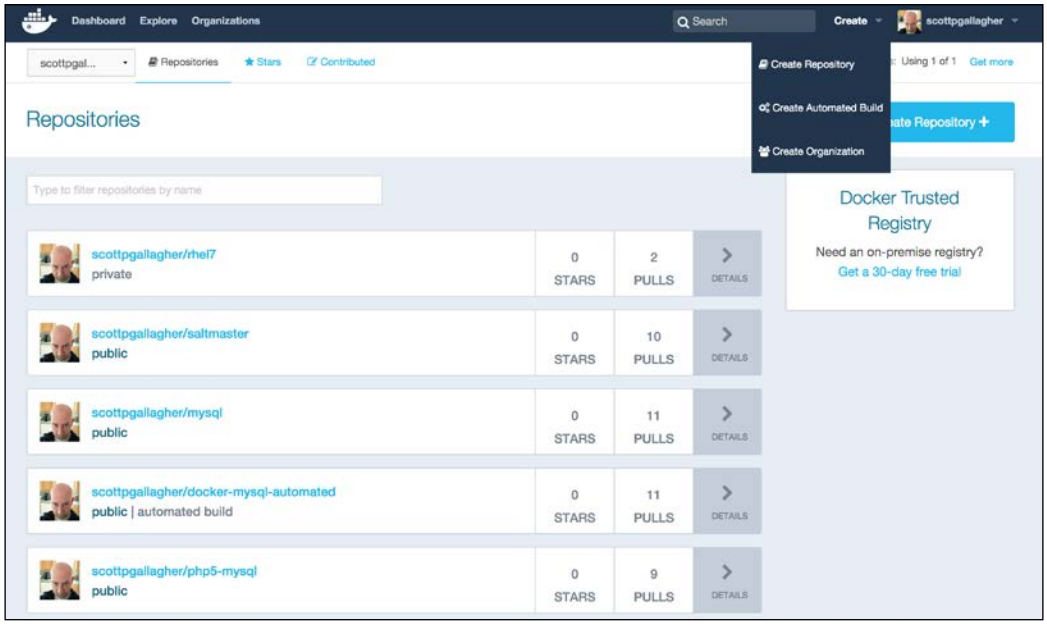
This is a drop-down list, where you can switch between all the organizations you belong to.

The Create menu

The **Create** menu is the new item along the top bar of the **Dashboard**. From this drop-down menu, you can perform three actions:

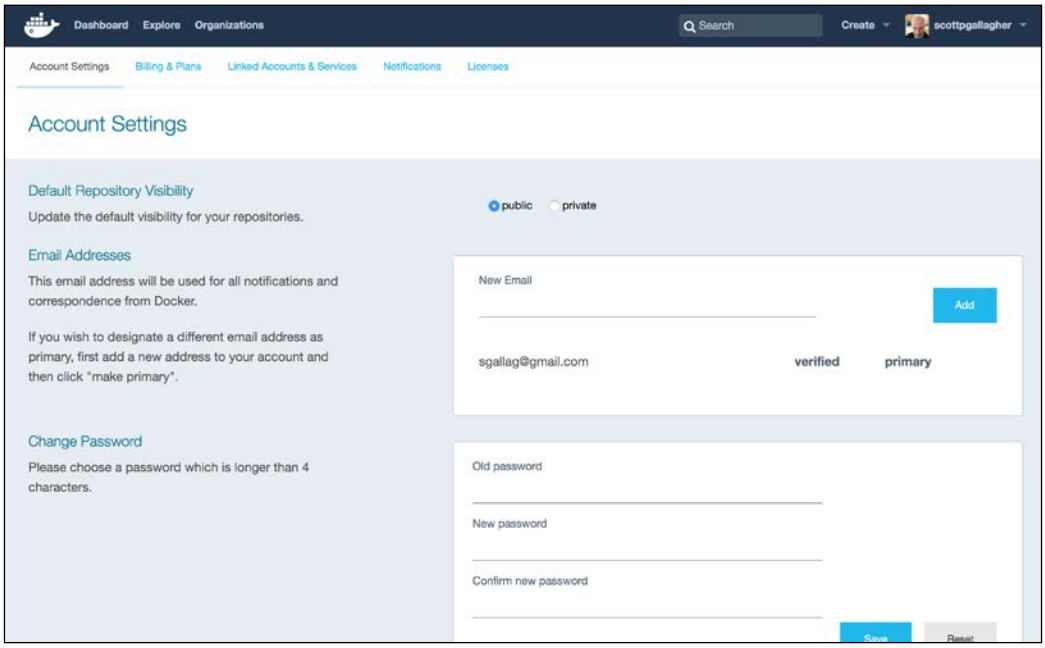
- Create repository
- Create automated build
- Create organization

A pictorial representation is shown in the following screenshot:

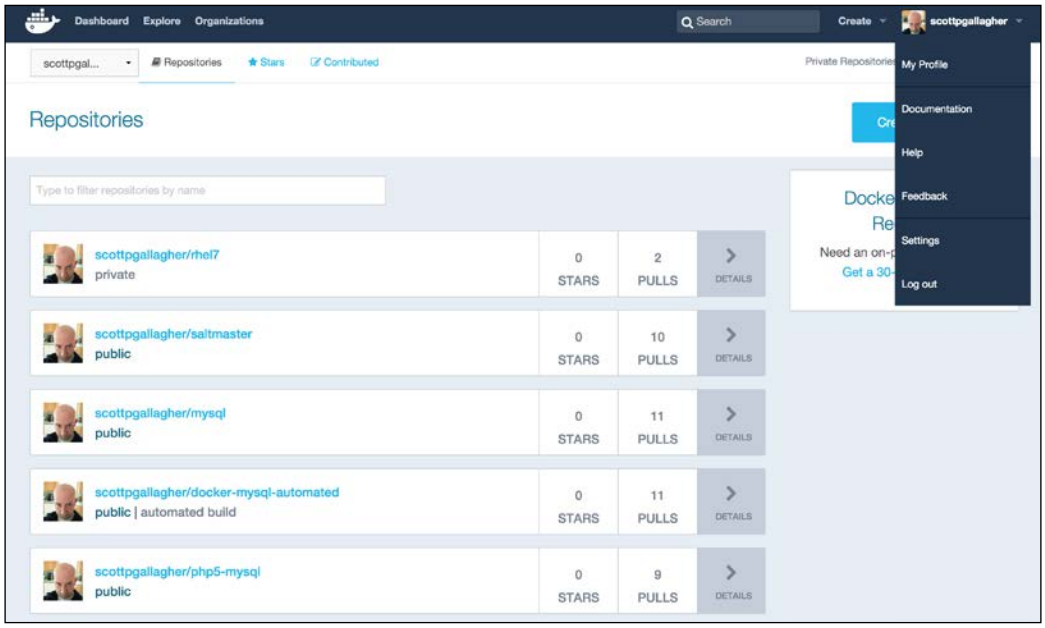


Settings

Probably, the first section everyone jumps to once they have created an account on the Docker Hub—the **Settings** page. I know, that's what I did at least.



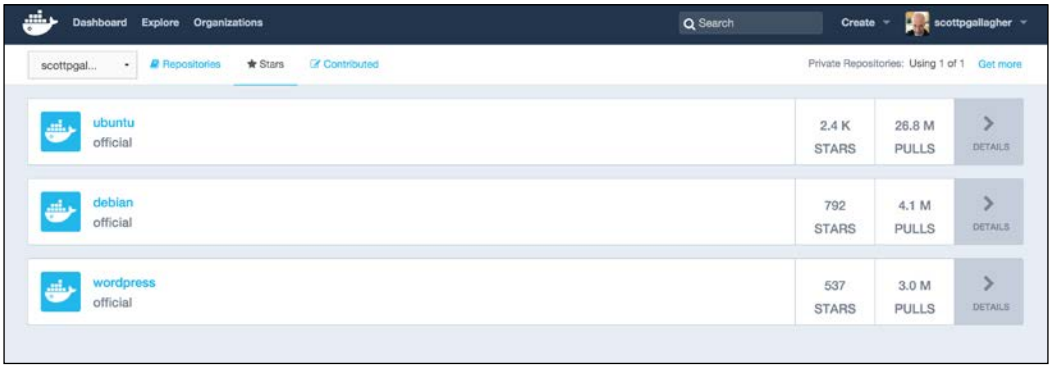
The **Account Settings** page can be found under the drop-down menu that is accessed in the upper-right corner of the dashboard on selecting **Settings**.



The page allows you to set up your public profile; change your password; see what organization you belong to, the subscriptions for e-mail updates you belong to, what specific notifications you would like to receive, what authorized services have access to your information, linked accounts (such as your GitHub or Bitbucket accounts); as well as your enterprise licenses, billing, and global settings. The only global setting as of now is the choice between having your repositories default to public or private upon creation. The default is to create them as public repositories.

The Stars page

Below the dark blue bar at the top of the **Dashboard** page are two more areas that are yet to be covered. The first, the **Stars** page, allows you to see what repositories you yourself have starred.



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

The second is a new setting in the new version of Docker Hub called **Contributed**. In this section, there will be a list of repositories you have contributed to outside of the ones within your **Repositories** list.

Docker Hub Enterprise

Docker Hub Enterprise, as it is currently known, will eventually be called **Docker Subscription**. We will focus on Docker Subscription, as it's the new and shiny piece. We will view the differences between Docker Hub and Docker Subscription (as we will call it moving forward) and view the options to deploy Docker Subscription.

Comparing Docker Hub to Docker Subscription

Let's first start off by comparing Docker Hub to Docker Subscription and see why each is unique and what purpose each serves:

Docker Hub

- Shareable image, but it can be private
- No hassle of self-hosting
- Free (except for a certain number of private images)

Docker Subscription

- Integrated into your authentication services (that is, AD/LDAP)
- Deployed on your own infrastructure (or cloud)
- Commercial support

Docker Subscription for server

Docker Subscription for server allows you to deploy both Docker Trusted Registry as well as Docker Engine on the infrastructure that you manage. Docker Trusted Registry is the location where you store the Docker images that you have created. You can set these up to be internal only or share them out publicly as well. Docker Subscription gives you all the benefits of running your own dedicated Docker hosted registry with the added benefits of getting support in case you need it.

Docker Subscription for cloud

As we saw in the previous section, we can also deploy Docker Subscription to a cloud provider if we wish. This allows us to leverage our existing cloud environments without having to roll our own server infrastructure up to host our Docker images.

The setup is the same as we reviewed in the previous section; but this time, we will be targeting our existing cloud environment instead.

Docker Registry

In this section, we will be looking at Docker Registry. Docker Registry is an open source application that you can run anywhere you please and store your Docker image in. We will look at the comparison between Docker Registry and Docker Hub and how to choose among the two. By the end of the section, you will learn how to run your own Docker Registry and see whether it's a true fit for you.

An overview of Docker Registry

Docker Registry, as stated earlier, is an open source application that you can utilize to store your Docker images on a platform of your choice. This allows you to keep them 100% private if you wish or share them as needed. The registry can be found at `https://docs.docker.com/registry/`.

This will run you through the setup and the steps to follow while pushing images to Docker Registry compared to Docker Hub. Docker Registry makes a lot of sense if you want to roll your own registry without having to pay for all the private features of Docker Hub. Next, let's take a look at some comparisons between Docker Hub and Docker Registry, so you can make an educated decision as to which platform to choose to store your images.

Docker Registry versus Docker Hub

Docker Registry will allow you to do the following:

- Host and manage your own registry from which you can serve all the repositories as private, public, or a mix between the two
- Scale the registry as needed based on how many images you host or how many pull requests you are serving out
- All are command-line-based for those that live on the command line

Docker Hub will allow you to:

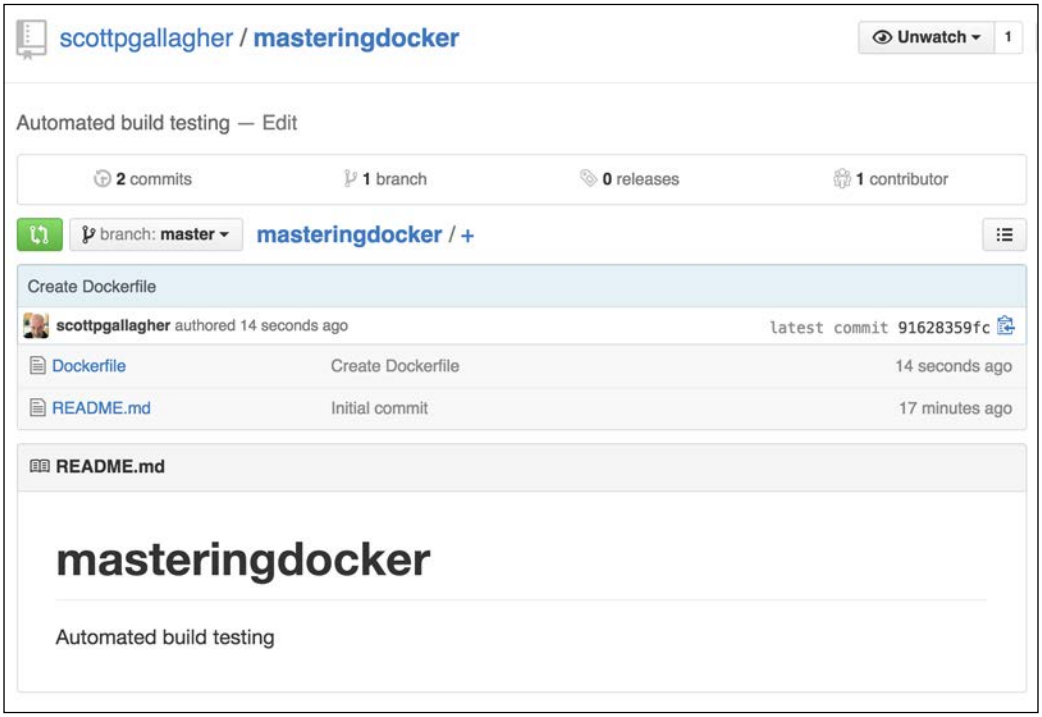
- Get a GUI-based interface that you can use to manage your images
- A location already set up on the cloud that is ready to handle public and/or private images
- Peace of mind of not having to manage a server that is hosting all your images

Automated builds

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

Setting up your code

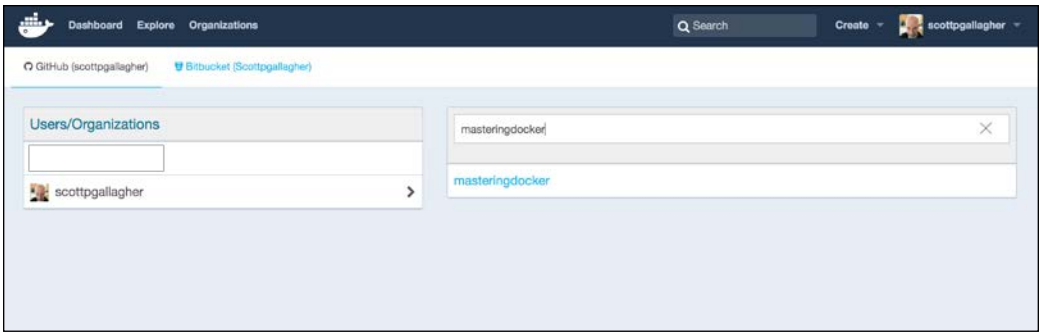
The first step to create automated builds is to set up your GitHub or Bitbucket code. These are the two options you have while selecting where to store your code. For our example, I will be using GitHub; but the setup will be the same for GitHub and Bitbucket.



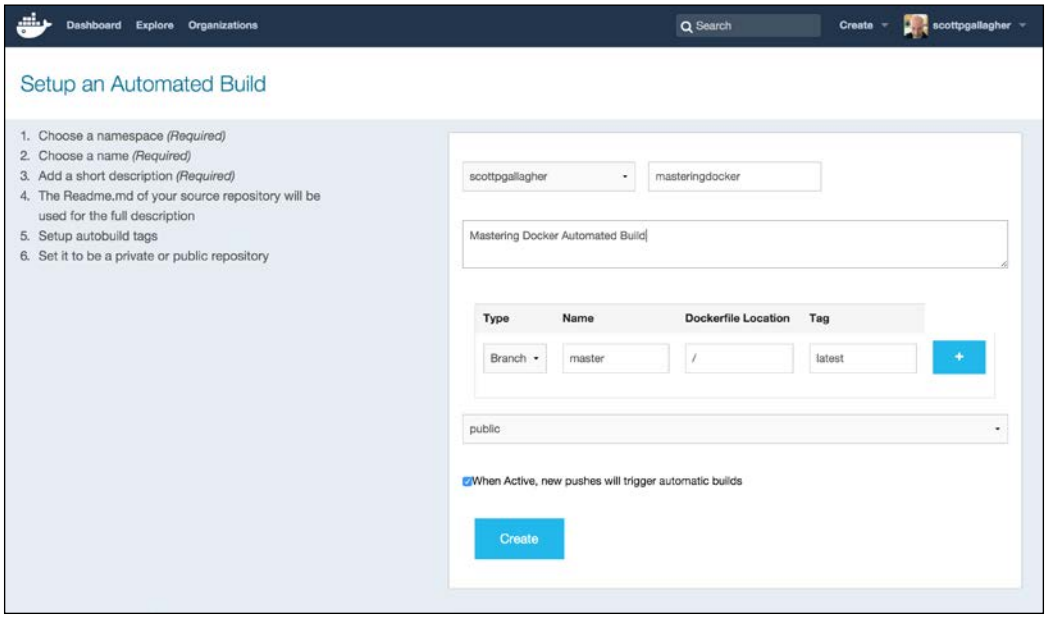
First, we set up our GitHub code that contains just a simple `README` file that we will edit for our purpose. This file could be anything as far as a script or even multiple files that you want to manipulate for your automated builds. One key thing is that we can't just leave the `README` file alone. One key piece is that a Dockerfile is required to do the builds when you want it to for them to be automated. Next, we need to set up the link between our code and Docker Hub.

Setting up Docker Hub

On Docker Hub, we are going to use the **Create** drop-down menu and select **Create Automated Build**. After selecting it, we will be taken to a screen that will show you the accounts you have linked to either GitHub or Bitbucket. You then need to search and select the repository from either of the locations you want to create the automated build from. This will essentially create a web hook that when a commit is done on a selected code repository, then a new build will be created on Docker Hub.

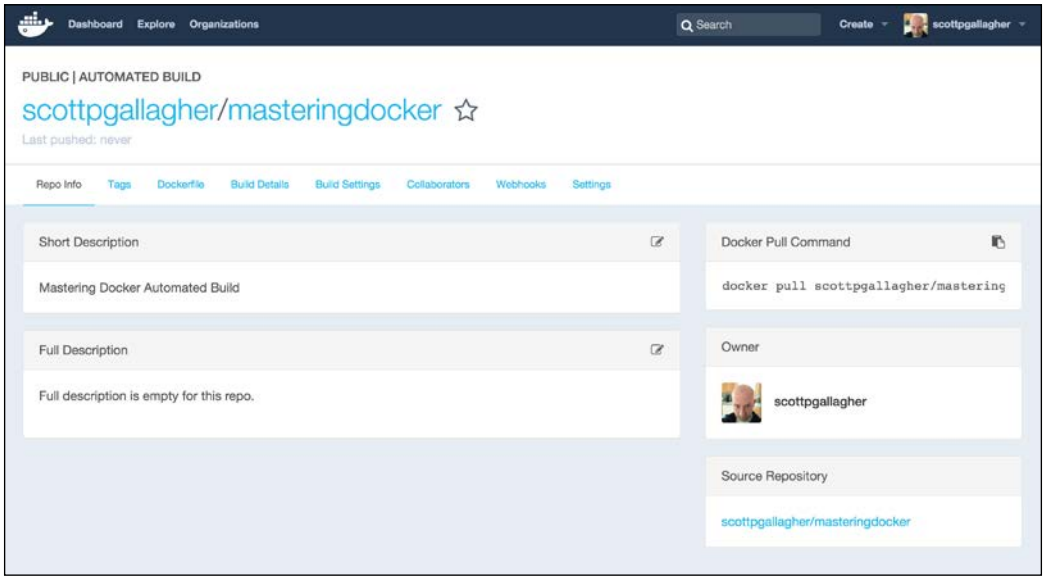


After you select the repository you would like to use, you will be taken to a screen similar to the following one:



For the most part, the defaults will be used by most. You can select a different branch if you want to use one, say a testing branch if you use one before the code may go to the master branch. The one thing that will not be filled out, but is required, is the description field. You must enter something here or you will not be able to continue past this page.

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



On this screen, you can see a lot of information on the automated build you have set up. Information such as tags, the Dockerfile in the code repository, build details, build settings, collaborators on the code, web hooks, and settings that include making the repository public or private and deleting the automated build repository as well.

Putting all the pieces together

So, let's take a run at doing a Docker automated build and see what happens when we have all the pieces in place and exactly what we have to do to kick off this automated build and be able to create our own magic:

1. Update the code or any file inside your GitHub or Bitbucket repository.
2. Upon committing the update, the automated build will be kicked off and logged in Docker Hub for that automated repository.

Creating your own registry

To create a registry of your own, use the following command:

```
$ docker-machine create --driver vmwarefusion registry

Creating SSH key...
Creating VM...
Starting registry...
Waiting for VM to come online...
```

To see how to connect Docker to this machine, run the following command:

```
$ docker-machine env registry
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://172.16.9.142:2376"
export DOCKER_CERT_PATH="/Users/scottpgallagher/.docker/machine/machines/
registry"
export DOCKER_MACHINE_NAME="registry"
# Run this command to configure your shell:
# eval "$(docker-machine env registry)"

$ eval "$(docker-machine env registry)"

$ docker pull registry
$ docker run -p 5000:5000 -v <HOST_DIR>:/tmp/registry-dev registry:2
```

This will specify to use version 2 of the registry.

For AWS (as shown in example from `https://hub.docker.com/_/registry/`):

```
$ docker run \
         -e SETTINGS_FLAVOR=s3 \
         -e AWS_BUCKET=acme-docker \
         -e STORAGE_PATH=/registry \
         -e AWS_KEY=AKIAHSHB43HS3J92MXZ \
         -e AWS_SECRET=xdDowwlK7TJajV1Y7EoOZrmuPEJlHYcNP2k4j49T \
         -e SEARCH_BACKEND=sqlalchemy \
         -p 5000:5000 \
         registry:2
```

Again, this will use version 2 of the self-hosted registry.

Then, you need to modify your Docker startups to point to the newly set up registry. Add the following line to the Docker startup in the `/etc/init.d/docker` file:

```
-H tcp://127.0.0.1:2375 -H unix:///var/run/docker.sock --insecure-
registry <REGISTRY_HOSTNAME>:5000
```

Most of these settings might already be there and you might only need to add `--insecure-registry <REGISTRY_HOSTNAME>:5000`:

To access this file, you will need to use `docker-machine`:

```
$ docker-machine ssh <docker-host_name>
```

Now, you can pull a registry from the public Docker Hub as follows:

```
$ docker pull debian
```

Tag it, so when we do a push, it will go to the registry we set up:

```
$ docker tag debian <REGISTRY_URL>:5000/debian
```

Then, we can push it to our registry:

```
$ docker push <REGISTRY_URL>:5000/debian
```

We can also pull it for any future clients (or after any updates we have pushed for it):

```
$ docker pull <REGISTRY_URL>:5000/debian
```

Summary

In this chapter, we dove deep into Docker Hub and also reviewed the new shiny Docker Subscription as well as the self-hosted Docker Registry. We have gone through the extensive review of each of them. You learned of the differences between them all and how to utilize each one. In this chapter, we also looked deep into setting up automated builds. We took a look at how to set up your own Docker Hub Registry. We have encompassed a lot in this chapter and I hope you have learned a lot and will like to put it all into good use.

In the next chapter, we will take a look at container management and how to manage all the containers that we create locally on our servers and in the cloud as well. We will also take a look at managing the images that keep piling up.

4
Managing Containers

In this chapter, you will learn how to manage your containers and the different ways you can go about doing so. This chapter will focus on the command line (as other chapters will cover other tools) to help lay the groundwork for understanding what the GUI-based apps are doing in the background. Sometimes, the command line is the best tool to help troubleshoot containers as well! Troubleshooting containers will be covered more in depth in *Chapter 10*, *Shipyard*. Apart from managing the containers, we will also cover topics on how to manage your images.

To be specific, the following topics will be covered:

- **Docker commands**: We will cover the Docker commands you can use to manage your containers
- **Using existing suite**: We will cover it using your existing management suites such as Chef or Puppet, plus some others to manage your containers
- **Docker Swarm**: You will have a brief overview of Docker Swarm, which we will be covering more in depth in a later chapter

The Docker commands

In this section, we will cover some Docker commands that you can use to manage your containers. These commands will range from looking at the status of containers and viewing what is going on inside the containers that are running to executing commands against the running containers. This will lay the groundwork for the GUI apps that we will be looking at in the later chapters. I believe it is important to understand what is going on behind the curtains when you run the GUI pieces.

docker attach

We will first take a look at the `docker attach` command. With this command, you can connect to the **standard input** (**STDIN**) of the container. We have a running container named `reposado`. Let's see how do we attach to it to see the STDIN:

```
$ docker attach reposado


192.168.59.3 - - [29/Jul/2015 13:40:15] "GET / HTTP/1.1" 200 -
192.168.59.3 - - [29/Jul/2015 13:40:15] "GET /products HTTP/1.1" 200 -
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET / HTTP/1.1" 200 -
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET /static/css/bootstrap.min.
css HTTP/1.1" 304 -
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET /static/css/bootstrap-
responsive.min.css HTTP/1.1" 304 -
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET /static/css/backgrid.min.css
HTTP/1.1" 304 -
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET /static/css/backgrid-
paginator.min.css HTTP/1.1" 304 -
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET /static/js/json2.js
HTTP/1.1" 304 -
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET /static/js/jquery.min.js
HTTP/1.1" 304 -
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET /static/js/underscore-min.js
HTTP/1.1" 304 -
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET /static/js/backbone-min.js
HTTP/1.1" 304 -
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET /static/js/backbone.wreqr.
min.js HTTP/1.1" 304 -
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET /static/js/backbone.
babysitter.min.js HTTP/1.1" 304 -
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET /static/js/backbone.
marionette.min.js HTTP/1.1" 304 -
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET /static/js/backbone-
pageable.min.js HTTP/1.1" 304 -
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET /static/js/backgrid.min.js
HTTP/1.1" 304 -
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET /static/js/backgrid-
paginator.min.js HTTP/1.1" 304 -
```

```
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET /static/js/margarita.js
HTTP/1.1" 304 -
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET /static/js/bootstrap.min.js
HTTP/1.1" 304 -
192.168.59.3 - - [29/Jul/2015 13:40:17] "GET /products HTTP/1.1" 200 -
192.168.59.3 - - [29/Jul/2015 13:40:18] "GET /static/img/glyphicons-
halflings-white.png HTTP/1.1" 304 -
```

In the previous example, we used the `docker attach` command to attach to the container named `reposado`. We can see the output as it happens in the container. You will stay attached to the container until you close your terminal window. This can help you troubleshoot error messages that might display when someone is trying to access the application that the container is serving up. It can also help track where the traffic might be coming from based on the output displayed.

docker diff

The next command is the `docker diff` command. With this command, we can view the changes that were made to a given container. We will again use the `reposado` container and take a look at the changes that were made to it:

```
$ docker diff reposado
C /Volumes
A /Volumes/reposado
A /Volumes/reposado/data
A /Volumes/reposado/data/html
C /opt
C /opt/reposado
C /opt/reposado/code
C /opt/reposado/code/reposadolib
A /opt/reposado/code/reposadolib/__init__.pyc
A /opt/reposado/code/reposadolib/reposadocommon.pyc
```

We can see that the command output is sorted into two columns. The first column will show us whether things changed (`C`), were added (`A`), or were deleted (`D`). In the earlier example, we don't have anything that was deleted, so we don't see any Ds in the first column. However, we do see that some items were changed as well as added. This can be helpful when you want to see what items might have been manipulated on the image that you are using.

docker exec

Next, let's take a look at one of the more recent commands that was introduced in Docker. This is one of the more powerful and more commonly used commands in the Docker command set. With the `docker exec` command, you can execute commands against your containers without the need to connect through something like SSH, like we would typically do.

There are two switches that are used:

- `docker exec -d`
- `docker exec -i`

What is the difference between the two? The difference is one will allow you to view the output of the command you are executing against the container (`docker exec -i`). The other will run it as a daemon in the background and not display any output (`docker exec -d`). After you execute this command, you can view the items that have changed by using the `docker diff` command we went over previously.

docker history

The `docker history` command will give you a full-blown history of everything that occurred on the image such as when and what created it as well as its size. As we can see in the following example, we ran the `docker history` command on the `reposado` image we created. We can see all the activity that went on for this image. We can see the activity that started 6 weeks ago, 21 hours ago, and then 4 hours ago. We can see the Git cloning, `pip` commands to install Python-related items, and symbolic links being created. We can see the size increase on running certain commands:

```
$ docker history scottpgallagher/reposado


IMAGE               CREATED             CREATED BY
SIZE                COMMENT
b61a1a023244        4 hours ago         /bin/sh -c #(nop) CMD ["/bin/sh"
"-c" "python                0 B
29dc8c2be431        4 hours ago         /bin/sh -c #(nop) EXPOSE 8089/tcp
       0 B
a02115b630cb        4 hours ago         /bin/sh -c ln -s /opt/reposado/
code/preferenc           36 B
6b568cd34339        4 hours ago         /bin/sh -c ln -s /opt/reposado/
code/reposadol           30 B
```

```
377509f5f585        4 hours ago         /bin/sh -c pip install simplejson
       485.7 kB
8b0312f24189        4 hours ago         /bin/sh -c pip install flask
                                4.071 MB
b1a301d9d39b        4 hours ago         /bin/sh -c git clone https://
github.com/jesse         791.9 kB
ea9b2533e044        4 hours ago        /bin/sh -c #(nop) ADD
file:ef8667f1286185255c   3.019 kB
1f875df3199b        21 hours ago        /bin/sh -c #(nop) ADD
file:58d34bd01478346ab1     393 B
2c283310dddd        21 hours ago        /bin/sh -c git clone https://
github.com/wdas/       326.8 kB
7e7e52de77bc        21 hours ago        /bin/sh -c #(nop) VOLUME [/
Volumes/data/repos       0 B
6f63b83840ff        21 hours ago        /bin/sh -c #(nop) VOLUME [/
Volumes/data/repos          0 B
136cc09dac1d        21 hours ago        /bin/sh -c apt-get update && apt-
get install        252.9 MB
2df9f745fbbc        21 hours ago        /bin/sh -c #(nop) MAINTAINER
Scott P. Gallagh            0 B
6d4946999d4f        6 weeks ago         /bin/sh -c #(nop) CMD ["/bin/
bash"]                      0 B
9fd3c8c9af32        6 weeks ago         /bin/sh -c sed -i 's/^#\s*\
(deb.*universe\)$/       1.895 kB
435050075b3f        6 weeks ago         /bin/sh -c echo '#!/bin/sh' > /
usr/sbin/polic       194.5 kB
428b411c28f0        6 weeks ago     /bin/sh -c #(nop) ADD
file:b3447f4503091bb6bb   188.1 MB
```

docker inspect

The next command we are looking at is `docker inspect`. We will take a look at the `busybox` image due to its size:

```
$ docker inspect busybox


[
{
    "Id":
"8c2e06607696bd4afb3d03b687e361cc43cf8ec1a4a725bc96e39f05ba97dd55",
```

```
    "Parent":
"6ce2e90b0bc7224de3db1f0d646fe8e2c4dd37f1793928287f6074bc451a57ea",
    "Comment": "",
    "Created": "2015-04-17T22:01:13.062208605Z",
    "Container":
"811003e0012ef6e6db039bcef852098d45cf9f84e995efb93a176a11e9aca6b9",
    "ContainerConfig": {
        "Hostname": "19bbb9ebab4d",
        "Domainname": "",
        "User": "",
        "AttachStdin": false,
        "AttachStdout": false,
        "AttachStderr": false,
        "PortSpecs": null,
        "ExposedPorts": null,
        "Tty": false,
        "OpenStdin": false,
        "StdinOnce": false,
        "Env": null,
        "Cmd": [
            "/bin/sh",
            "-c",
            "#(nop) CMD [\"/bin/sh\"]"
        ],
        "Image":
"6ce2e90b0bc7224de3db1f0d646fe8e2c4dd37f1793928287f6074bc451a57ea",
        "Volumes": null,
        "VolumeDriver": "",
        "WorkingDir": "",
        "Entrypoint": null,
        "NetworkDisabled": false,
        "MacAddress": "",
        "OnBuild": null,
        "Labels": {}
    },
```

```
    "DockerVersion": "1.6.0",
    "Author": "Jérôme Petazzoni \u003cjerome@docker.com\u003e",
    "Config": {
        "Hostname": "19bbb9ebab4d",
        "Domainname": "",
        "User": "",
        "AttachStdin": false,
        "AttachStdout": false,
        "AttachStderr": false,
        "PortSpecs": null,
        "ExposedPorts": null,
        "Tty": false,
        "OpenStdin": false,
        "StdinOnce": false,
        "Env": null,
        "Cmd": [
            "/bin/sh"
        ],
        "Image":
"6ce2e90b0bc7224de3db1f0d646fe8e2c4dd37f1793928287f6074bc451a57ea",
        "Volumes": null,
        "VolumeDriver": "",
        "WorkingDir": "",
        "Entrypoint": null,
        "NetworkDisabled": false,
        "MacAddress": "",
        "OnBuild": null,
        "Labels": {}
    },
    "Architecture": "amd64",
    "Os": "linux",
    "Size": 0,
    "VirtualSize": 2433303
}
]
```

We can see things such as:

- When the image was created
- Whether the container is using any volumes
- The particular network settings being established
- What architecture is being used
- The OS for the container

We can also see its size, plus a plethora of other items that are related to the running container.

docker logs

The `docker logs` command will allow you to look at what has been happening on your running container. There is a switch that you can use to get a running output from your container as well, which we will cover shortly. This is similar to the `docker attach` command that we went over earlier, but this will allow you to gather history from when the container started until the time you ran the `docker logs` command:

```
$ docker logs reposado


 Running on http://0.0.0.0:8089/ (Press CTRL+C to quit)
192.168.59.3 - - [29/Jul/2015 15:56:23] "GET / HTTP/1.1" 200 -
192.168.59.3 - - [29/Jul/2015 15:56:23] "GET /products HTTP/1.1" 200 -
192.168.59.3 - - [29/Jul/2015 15:56:23] "GET /favicon.ico HTTP/1.1" 404 -
192.168.59.3 - - [29/Jul/2015 15:56:29] "POST /new_branch/test HTTP/1.1"
200 -
192.168.59.3 - - [29/Jul/2015 15:56:29] "GET /products HTTP/1.1" 200 -
```

Now, `docker logs -f` will give you a running output of what is actively happening on the container. This is helpful when you are troubleshooting your containers. It will allow you to actively monitor your container while you execute, and the application it is running.

docker ps

We covered the `docker ps` command earlier, but we will now take a look at the switches we can add to the command.

Here are the switches we will be taking a look at:

- `docker ps -a`: This will give you a list of all the containers. By default, when you run the `docker ps` command, it will only show the ones that are running. It will also provide the status of the containers that were stopped and how long ago they were stopped. It will also give you the names of the containers as well as the respective commands that were running on these containers.
- `docker ps -l`: This will give you the latest created containers, including the ones that are not running. It again will give you the same information that the `docker ps -a` command provides to you. With the `docker ps -l` command, you can see what containers were running and then launch them again with the `docker start <container_name>` command. This will bring the image back to the state it was when it was stopped/halted.
- `docker ps -n=`: This will give you the power to slim down the previous command of `docker ps -l`. This is useful if the list becomes too long. The `docker ps -n=` command allows you to specify a number of how many of the previous containers you want to view. For example, `$ docker ps -n=5` will return the last five containers, whether they are running or not. There are also other switches you can use with the `docker ps` command. Don't forget that on every command, you can use the `--help` switch that will provide more information on each command, including all the switches you can utilize.

docker stats

The `docker stats` command will give you live running information on your container. It will provide information such as the container name, CPU activity, memory usage / memory limit, memory percentage being used, as well as the network input/output:

```
$ docker stats reposado


CONTAINER           CPU %               MEM USAGE/LIMIT     MEM %
NET I/O
reposado            0.06%               13.31 MB/2.099 GB   0.63%
5.549 kB/12.9 kB
```

This can be helpful if you have a container using up a lot of memory and want to put restrictions on it. You can exit this command by using the *Ctrl + C* key combination on your keyboard.

docker top

The `docker top` command will allow you to view what commands are currently running on your container. It will allow you to see what command is running as well as how long it has been running:

```
$ docker top reposado

UID                 PID                 PPID                C
STIME               TTY                 TIME                CMD
root                21094               825                 0
15:49               ?                   00:00:00            /bin/sh -c
python /opt/margarita/margarita.py
root                21098               21094               0
15:49               ?                   00:00:00            python /opt/
margarita/margarita.py
```

Using your existing management suite

In this section, we will look at what you can do with your already existing management suite(s) and how you can use them to target actions against your containers. We will cover most of the major ones: Puppet, Chef, Ansible, and SaltStack. There are surely more out there and more coming out daily! This will help you leverage your already existing management environment as well as understand other options that are available.

Puppet

Puppet (as of version 3.8) allows you to manage your Docker containers with your pre-existing Puppet environment. You simply need to include Docker to your manifests.

You can then use Puppet to install Docker on the hosts as well as run containers on these Docker hosts. For example, let's deploy the `nginx` container using the Puppet code:

```
docker::run { 'website':
  image   => 'nginx',
```

```
  command => '/usr/sbin/nginx -g "daemon off;"',
}
```

We can also execute the code against our already existing containers using Puppet:

```
docker::exec { 'update-nginx':
  detach    => true,
  container => 'nginx',
  command   => 'apt-get update -y nginx',
  tty       => true,
}
```

This will update the `nginx` package in the container named `nginx` and display the output on your screen, since `tty` is set to `true`.

You can also use other Docker commands in place of the previous `exec` statement. Simply refer to the Puppet documentation for more information on it.

Chef

Chef also allows you to manage your Docker infrastructure using your existing Chef infrastructure. Chef is a little different than Puppet, as it uses recipes to do its tasks. An example we can use to pull an image from Docker Hub to our Docker host is:

```
docker_image '<image_name>' do
  tag 'latest'
  action :pull
end
```

We can then run that pulled image and turn it into a container:

```
docker_container '<image_name>' do
  tag 'latest'
  action :run
end
```

With the Chef recipes, the possibilities are endless as to what you could do. The communities in Chef (as well as these other management suites) are very large and recipes are being shared all the time.

The easiest way to find a Chef recipe is to use ever-handy search engines such as Google or Yahoo to find an already written recipe that we can just drop in place or modify as needed.

To learn more about how to use Chef along with Docker to manage your environment, use the following link:

`https://supermarket.chef.io/cookbooks/docker`

Ansible

Like the others, we have explored Ansible that can do the many and same things as the others. If you already have Ansible in place, you have a leg up; you don't need to get a management suite in place.

If we want to use Ansible to manage Docker, we can use Ansible to spin up the containers:

```
- name: nginx-host
  docker:
    name: nginx-host
    image: nginx
    state: started
```

This will launch a Docker container named `nginx-host` using the `nginx` image on the Docker Hub, ensuring it starts. The catch is that, if there is already a container named `nginx-host`, it won't start a container.

We can also stop a running container:

```
- name: Stop a container
  docker:
    name: nginx-host
    state: stopped
```

We can also start containers:

```
- name: Start a container
  docker:
    name: test-container-stopped
    state: started
```

SaltStack

Lastly, we will take a look at SaltStack that, as you can guess, can manage Docker containers as well. Let's see how we can start a container using SaltStack:

```
nginx:
  docker.running:
    - container: nginx
    - image: nginx
    - port_bindings: "80/tcp":
            HostIp: ""
            HostPort: "80"
```

The previous example using SaltStack will start a container and name it `nginx` based off the `container:` section, then pull the `nginx` image from the Docker Hub from the `image:` section. It will set up the port bindings as well. It will set up TCP port `80` on the Docker container from the `port_bindings:` section and tie it to the host port of `80` based off of the `HostPort:` entry.

We can also stop these containers with SaltStack:

```
salt '*' docker.stop <container id>
```

This will fire off the `salt` command and use the `docker.stop` module. It will look for the container ID that you specify and stop it when it finds it. You can start a container in the same way as well:

```
salt '*' docker.start <image_name:tag>
```

There are many other SaltStack commands that you too can utilize. These can be found on the SaltStack website:

`http://docs.saltstack.com/en/latest/ref/modules/all/salt.modules.dockerio.html#salt.modules.dockerio.stop`

Docker Swarm

In this section, we will do a brief overview of Docker Swarm. We will take a look at what it is, what you can do with it to manage your containers, and what to look forward to in the later chapters with regards to Docker Swarm.

What is Docker Swarm?

The idea behind Docker Swarm is to have native clustering available inside Docker. This will allow you to both easily scale your environments as well as manage them from a central location. The best part is that, since it's tied so tightly with the Docker API, any command you use with Docker can be used in conjunction with managing the nodes in your Swarm cluster. The setup is very simple as follows:

1. You install the Swarm component through a `docker pull` command.
2. You then set up and configure the Swarm manager.
3. Lastly, you add the nodes to Docker Swarm.

This setup uses the TCP communication between all the Swarm nodes through an open TCP port. It also requires that you have Docker installed on each node (as if we'd not want it installed). Lastly, it requires that you create and manage TLS certificates that will allow secure communication between all the hosts.

What can Docker Swarm do?

Docker Swarm, as you previously learned, allows for clustering through secure TLS communication. It allows for discovery services to be set up as well. This will allow you to set up services such that, when new nodes are added to the Swarm, they can be automatically added to the correct corresponding service and allowed to join the service to help scale for its needs.

Swarm also allows advanced scheduling of jobs. This allows you to choose a strategy to rank all the nodes in your cluster. The three options to rank your nodes are:

- `spread`
- `binpack`
- `random`

The first two allocate jobs based on the machine's available CPU and RAM. The last one—`random`—does exactly as it says. It randomly chooses a node to run the requested job on.

You can review more in-depth examples of these on the Docker Docs website:

`https://docs.docker.com/swarm/scheduler/strategy/`

Summary

In this chapter, you looked at the Docker commands that can be used to manage your containers, viewing their status and looking inside them to see what they are doing.

To perform tasks, we looked at how we can execute commands against our running containers. This will lay the groundwork, so you understand what is going on behind the scenes if you use a GUI application to manage containers.

We also took a look at utilizing your existing management suite and using it to cover more ground, including your Docker containers. We took a look at four major management suites that you can use to manage your Docker containers.

We lastly took a look at Docker Swarm that hopefully got you excited for the later chapter on Docker Swarm. With Docker Swarm, we can cluster our containers, view where all our containers are running across multiple Docker hosts, and use it for discovery services to help scale our environments.

In the next chapter, we will be looking at Docker security—the topic that is always at the forefront of everyone's mind when it comes to any or all of technology. We will go over all the aspects of Docker security—the good, the not so bad, and what to look forward to.

5
Docker Security

In this chapter, we will be taking a look at Docker security—the topic on the forefront of everyone's minds these days. We will be splitting up the chapter into four sections:

- Containers versus VMs
- The Docker commands
- Docker security – best practices
- The Docker bench security application

Now, let's take a look at each of these sections one after the other.

Containers versus VMs

In this section, we will be looking at the differences in Docker containers and typical virtual machines. We will focus on the benefits that Docker containers have over typical virtual machines. We'll take a look at the good; the not so bad: those items that aren't bad but you will want keep an eye on them; and the items you want to look out for: those are the items that you will ultimately want to consider while using Docker containers over typical virtual machines.

The good

When you start a Docker container, there is a lot of work going on behind the scenes and two of those items are setting up namespaces and control groups. What does that mean? By setting up namespaces, Docker keeps the processes isolated in each container; not only from other containers, but also from the host system. The control groups ensure that each container gets its own share of items such as CPU, memory, and disk I/O. More importantly, they ensure that one container doesn't exhaust all the resources on a given Docker host.

Each container also gets its own network stack that again contributes to the idea of isolation. With each container getting its own network stack, other containers don't get access to each other, unless otherwise specified by Docker linking. Also, with this, you can accordingly set up access through items such as `iptables`.

Lastly, what I consider one of the biggest advantages of Docker over typical virtual machines is that you can finally turn off SSH in your containers. There is no need to enable SSH in your containers anymore to manage them or to issue commands against them. Docker has the tools to execute items against the containers and pull information that is needed to help troubleshoot containers as well. With commands such as `docker execute`, `docker top`, `docker logs`, `docker events`, and `docker stats`, you can do everything you need to do without exposing any more security holes than you need to.

The not so bad

Not so bad, as we will be calling this section, is just to keep you informed about the items that are in the technology.

What you need to realize is that, when you are dealing with virtual machines, you can control the required permissions, that is, who has access to what virtual machines. With Docker, you have a little disadvantage because whoever has access to the Docker daemon on your server has access to every Docker container that you are running. They can run new containers; they can stop existing containers and can delete images as well. Be careful who you grant permission to access the Docker daemon on your hosts. They essentially hold the keys to the kingdom with respect to all your containers. Knowing this, it is recommended to use Docker hosts only for Docker; keep other services separate from Docker.

Hopefully, you trust your organization and all those who do have access to these systems.

What to look out for

You will most likely be setting up virtual machines from scratch. It is probably impossible to get the virtual machine from someone else, due to its sheer size. So, you will be aware of what is inside the virtual machine and what isn't. This being said, with Docker containers, you will not be aware of what could be there inside the image you might be using for your container(s).

The Docker commands

Let's take a look at the Docker commands that can be used to help tighten up security as well as view information in the images you might be using. There are two commands that we are going to be focusing on.

The first will be the `docker run` command, so you can see some of the items you can use to your advantage with this command. Second, we will take a look at the `docker diff` command (that we went over in the previous chapter) that you can use to view what has been done with the image that you are planning to use.

docker run

With respect to the `docker run` command, we will mainly focus on the option that allows you to set everything inside the container as read-only instead of a specified directory or volume. Let's take a look at an example and break down what it exactly does:

```
$ docker run --name mysql --read-only -v /var/lib/mysql -v /tmp:/tmp:rw
-e MYSQL_ROOT_PASSWORD=password -d mysql
```

Here, we are running a `mysql` container and setting the entire container as read-only, except for the `/var/lib/mysql` directory. What this means is that the only location the data can be written inside the container is the `/var/lib/mysql` directory. Any other location inside the container won't allow you to write anything in it. If you try to run the following, it would fail:

```
$ docker exec mysql touch /opt/filename
```

This can be extremely helpful if you want to control where the containers can write to or not write to. Be sure to use this wisely. Test thoroughly, as it could have consequences when the applications can't write to certain locations.

Remember the Docker volumes we looked at in the previous chapters, where we were able to set the volumes to be read-only. Similar to the previous command with `docker run`, where we set everything to read-only except for a specified volume, we can now do the opposite and set just a single volume (or more if you use more `-v` switches) to read only. The thing to remember about volumes is that when you use a volume and mount it into a container, it will mount as an empty volume over the top of that directory inside the container, unless you use the `--volumes-from` switch or add data to the container in some other way after the fact:

```
$ docker run -d -v /opt/uploads:/opt/uploads:ro nginx
```

This will mount a volume in `/opt/uploads` and set it to read-only. This can be useful if you don't want a running container to write to a volume to keep the data or configuration files intact.

The last option we want to look at with regards to the `docker run` command is the `--device=` switch. This switch allows us to mount a device from the Docker host into a specified location inside the container. By doing so, there are some security risks we need to be aware of. By default, when you do this, the container will get full access: read, write, and the mknod access to the device's location. Now, you can control these permissions by manipulating `rwm` at the end of the switch command. Let's take a look at some of these and see how they work:

```
$ docker run --device=/dev/sdb1:/dev/sdc2 -it ubuntu:latest  /bin/bash
```

The previous command will run the latest Ubuntu image and mount the `/dev/sdb1` device inside the container in the `/dev/sdc2` location:

```
$ docker run --device=/dev/sdb1:/dev/sdc2:r -it ubuntu:latest /bin/bash
```

This command will run the latest Ubuntu image and mount the `/dev/sdb1` device inside the container in the `/dev/sdc2` location. But this one has the `:r` tag at the end of it that specifies it's read-only and can't be written to.

docker diff

Let's take another look at the `docker diff` command since it relates to the security aspects of the containers you may want to use from the images that are hosted on Docker Hub or other related repositories.

Remember that whoever has access to your Docker host and the Docker daemon has access to all of your running Docker containers. This being said, if you don't have monitoring in place, someone could be executing commands against your containers and doing malicious things:

```
$ docker diff <running_container_name>
```

Docker security – best practices

In this section, we will look at the best practices when it comes to Docker as well as the Center for Internet Security guide to properly secure all the aspects of your Docker environment. You will be referring to this guide when you actually run the scan (in the next section of this chapter) and get results back of what needs or should be fixed. The guide is broken down into the following sections:

- The host configuration
- The Docker daemon configuration
- The Docker daemon configuration files
- Container images/runtime
- Docker security operations

Docker – best practices

Before we dive into the Center for Internet Security guide, let's go over some of the best practices to use Docker:

- **One application per container**: Spread out your applications to one per container. Docker was built for this and it makes everything easier at the end of the day. That isolation we talked about earlier is where this is the key.
- **Review who has access to your Docker hosts**: Remember that whoever has access to your Docker hosts has access to manipulate all your images and containers on the host.
- **Use the latest version**: Always use the latest version of Docker. This will ensure that all security holes have been patched and you have the latest features as well.
- **Use the resources**: Use the resources available if you need help. The community within Docker is huge and immensely helpful. Use their website, documentation, and the IRC chat rooms to your advantage.

CIS guide – host configuration

This part of the guide is about the configuration of your Docker hosts. This is that part of the Docker environment where all your containers run. Thus, keeping it secure is of the utmost importance. This is the first line of defense against attackers.

CIS guide – Docker daemon configuration

This part of the guide has the recommendations that secure the running Docker daemon. Everything you do to the Docker daemon configuration affects each and every container. These are the switches you can attach to the Docker daemon we saw previously, and to the items you will see in the next section when we run through the tool.

CIS guide – Docker daemon configuration files

This part of the guide deals with the files and directories that the Docker daemon uses. This ranges from permissions to ownerships. Sometimes, these areas may contain information you don't want others to know about that could be in a plain text format.

CIS guide – container images/runtime

This part of the guide contains both the information for securing the container images as well as the container runtime.

The first part contains images, cover base images, and the build files that were used. As we covered previously, you need to be sure about the images you are using not only for your base images, but for any aspect of your Docker experience. This section of the guide covers the items you should follow while creating your own base images to ensure they are secure.

The second part, the container runtime, covers a lot of security-related items. You have to take care with the runtime variables you are providing. In some cases, attackers can use them to their advantage, while you think you are using them to your own advantage. Exposing too much in your container can compromise the security of not only that container, but the Docker host and the other containers running on that host.

CIS guide – Docker security operations

This part of the guide covers the security areas that involve deployment. These items are more closely tied to the best practices and the recommendations of items that are recommended to be followed.

The Docker bench security application

In this section, we will cover the Docker benchmark security application that you can install and run. The tool will inspect:

- The host configuration
- The Docker daemon configuration
- The Docker daemon configuration files
- Container images and build files
- Container runtime
- The Docker security operations

Looks familiar? It should, as these are the same items that we reviewed in the previous section only built into an application that will do a lot of heavy lifting for you. It will show you what warnings arise with your configurations and provide information on other configuration items and even the items that have passed the test.

We will look at how to run the tool, a live example, and what the output of the process will mean.

Running the tool

Running the tool is simple. It's already been packaged up for us inside a Docker container. While you can get the source code and customize the output or manipulate it in some way (say, e-mail the output), the default may be all you need.

The code is found here:

`https://github.com/docker/docker-bench-security`

To run the tool, we will simply copy and paste the following into our Docker host:

```
$ docker run -it --net host --pid host --cap-add audit_control \
    -v /var/lib:/var/lib \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -v /usr/lib/systemd:/usr/lib/systemd \
    -v /etc:/etc --label docker_bench_security \
    diogomonica/docker-bench-security
```

If you don't already have the image, it will first download the image and then start the process for you. Now that we've seen how easy it is to install and run it, let's take a look at an example on a Docker host to see what it actually does. We will then take a look at the output and take a dive into dissecting it.

There is also an option to clone the Git repository, enter the directory from the `git clone` command, and run the provided shell script. So, we have multiple options!

Let's take a look at an example and break down each section:

```
$ docker run -it --net host --pid host --cap-add audit_control \
>     -v /var/lib:/var/lib \
>     -v /var/run/docker.sock:/var/run/docker.sock \
>     -v /usr/lib/systemd:/usr/lib/systemd \
>     -v /etc:/etc --label docker_bench_security \
>     diogomonica/docker-bench-security
# ------------------------------------------------------------------------------
# Docker Bench for Security v1.0.0
#
# Docker, Inc. (c) 2015
#
# Checks for dozens of common best-practices around deploying Docker containers in production.
# Inspired by the CIS Docker 1.6 Benchmark:
# https://benchmarks.cisecurity.org/tools2/docker/CIS_Docker_1.6_Benchmark_v1.0.0.pdf
# ------------------------------------------------------------------------------

Initializing Thu Aug 13 07:37:22 EDT 2015
```

- **The host configuration**:

```
[INFO] 1 - Host Configuration
[WARN] 1.1  - Create a separate partition for containers
[PASS] 1.2  - Use an updated Linux Kernel
[PASS] 1.5  - Remove all non-essential services from the host - Network
[PASS] 1.6  - Keep Docker up to date
[INFO] 1.7  - Only allow trusted users to control Docker daemon
[INFO]      * docker:x:999:
[WARN] 1.8  - Failed to inspect: auditctl command not found.
[WARN] 1.9  - Failed to inspect: auditctl command not found.
[WARN] 1.10 - Failed to inspect: auditctl command not found.
[INFO] 1.11 - Audit Docker files and directories - docker-registry.service
[INFO]      * File not found
[INFO] 1.12 - Audit Docker files and directories - docker.service
[INFO]      * File not found
[WARN] 1.13 - Failed to inspect: auditctl command not found.
[INFO] 1.14 - Audit Docker files and directories - /etc/sysconfig/docker
[INFO]      * File not found
[INFO] 1.15 - Audit Docker files and directories - /etc/sysconfig/docker-network
[INFO]      * File not found
[INFO] 1.16 - Audit Docker files and directories - /etc/sysconfig/docker-registry
[INFO]      * File not found
[INFO] 1.17 - Audit Docker files and directories - /etc/sysconfig/docker-storage
[INFO]      * File not found
[WARN] 1.18 - Failed to inspect: auditctl command not found.
```

- **The Docker daemon configuration**:

```
[INFO] 2 - Docker Daemon Configuration
[PASS] 2.1  - Do not use lxc execution driver
[WARN] 2.2  - Restrict network traffic between containers
[PASS] 2.3  - Set the logging level
[PASS] 2.4  - Allow Docker to make changes to iptables
[PASS] 2.5  - Do not use insecure registries
[INFO] 2.6  - Setup a local registry mirror
[INFO]      * No local registry currently configured
[WARN] 2.7  - Do not use the aufs storage driver
[PASS] 2.8  - Do not bind Docker to another IP/Port or a Unix socket
[INFO] 2.9  - Configure TLS authentication for Docker daemon
[INFO]      * Docker daemon not listening on TCP
[INFO] 2.10 - Set default ulimit as appropriate
[INFO]      * Default ulimit doesn't appear to be set
```

- **The Docker daemon configuration files**:

```
[INFO] 3 - Docker Daemon Configuration Files
[INFO] 3.1  - Verify that docker.service file ownership is set to root:root
[INFO]      * File not found
[INFO] 3.2  - Verify that docker.service file permissions are set to 644
[INFO]      * File not found
[INFO] 3.3  - Verify that docker-registry.service file ownership is set to root:root
[INFO]      * File not found
[INFO] 3.4  - Verify that docker-registry.service file permissions are set to 644
[INFO]      * File not found
[INFO] 3.5  - Verify that docker.socket file ownership is set to root:root
[INFO]      * File not found
[INFO] 3.6  - Verify that docker.socket file permissions are set to 644
[INFO]      * File not found
[INFO] 3.7  - Verify that Docker environment file ownership is set to root:root
[INFO]      * File not found
[INFO] 3.8  - Verify that Docker environment file permissions are set to 644
[INFO]      * File not found
[INFO] 3.9  - Verify that docker-network environment file ownership is set to root:root
[INFO]      * File not found
[INFO] 3.10 - Verify that docker-network environment file permissions are set to 644
[INFO]      * File not found
[INFO] 3.11 - Verify that docker-registry environment file ownership is set to root:root
[INFO]      * File not found
[INFO] 3.12 - Verify that docker-registry environment file permissions are set to 644
[INFO]      * File not found
[INFO] 3.13 - Verify that docker-storage environment file ownership is set to root:root
[INFO]      * File not found
[INFO] 3.14 - Verify that docker-storage environment file permissions are set to 644
[INFO]      * File not found
[PASS] 3.15 - Verify that /etc/docker directory ownership is set to root:root
[PASS] 3.16 - Verify that /etc/docker directory permissions are set to 755
[INFO] 3.17 - Verify that registry certificate file ownership is set to root:root
[INFO]      * Directory not found
[INFO] 3.18 - Verify that registry certificate file permissions are set to 444
[INFO]      * Directory not found
[INFO] 3.19 - Verify that TLS CA certificate file ownership is set to root:root
[INFO]      * No TLS CA certificate found
[INFO] 3.20 - Verify that TLS CA certificate file permissions are set to 444
[INFO]      * No TLS CA certificate found
[INFO] 3.21 - Verify that Docker server certificate file ownership is set to root:root
[INFO]      * No TLS Server certificate found
[INFO] 3.22 - Verify that Docker server certificate file permissions are set to 444
[INFO]      * No TLS Server certificate found
[INFO] 3.23 - Verify that Docker server key file ownership is set to root:root
[INFO]      * No TLS Key found
[INFO] 3.24 - Verify that Docker server key file permissions are set to 400
[INFO]      * No TLS Key found
[INFO] 3.25 - Verify that Docker socket file ownership is set to root:docker
[INFO]      * File not found
[INFO] 3.26 - Verify that Docker socket file permissions are set to 660
[INFO]      * File not found
```

- **Container images and build files**:

```
[INFO] 4 - Container Images and Build Files
[WARN] 4.1  - Create a user for the container
[WARN]      * Running as root: suspicious_mccarthy
```

- **Container runtime**:

```
[INFO] 5  - Container Runtime
[WARN] 5.1  - Verify AppArmor Profile, if applicable
[WARN]      * No AppArmorProfile Found: suspicious_mccarthy
[PASS] 5.2  - Verify SELinux security options, if applicable
[WARN] 5.3  - Verify that containers are running only a single main process
[WARN]      * Too many proccesses running: suspicious_mccarthy
[WARN] 5.4  - Restrict Linux Kernel Capabilities within containers
[WARN]      * Capabilities added: CapAdd=[audit_control] to suspicious_mccarthy
[PASS] 5.5  - Do not use privileged containers
[WARN] 5.6  - Do not mount sensitive host system directories on containers
[WARN]      * Sensitive directory /etc mounted in: suspicious_mccarthy
[WARN]      * Sensitive directory /lib mounted in: suspicious_mccarthy
[WARN] 5.7  - Do not run ssh within containers
[WARN]      * Container running sshd: suspicious_mccarthy
[PASS] 5.8  - Do not map privileged ports within containers
[WARN] 5.10 - Do not use host network mode on container
[WARN]      * Container running with networking mode 'host': suspicious_mccarthy
[WARN] 5.11 - Limit memory usage for container
[WARN]      * Container running without memory restrictions: suspicious_mccarthy
[WARN] 5.12 - Set container CPU priority appropriately
[WARN]      * Container running without CPU restrictions: suspicious_mccarthy
[WARN] 5.13 - Mount container's root filesystem as read only
[WARN]      * Container running with root FS mounted R/W: suspicious_mccarthy
[PASS] 5.14 - Bind incoming container traffic to a specific host interface
[PASS] 5.15 - Do not set the 'on-failure' container restart policy to always
[WARN] 5.16 - Do not share the host's process namespace
[WARN]      * Host PID namespace being shared with: suspicious_mccarthy
[PASS] 5.17 - Do not share the host's IPC namespace
[PASS] 5.18 - Do not directly expose host devices to containers
[INFO] 5.19 - Override default ulimit at runtime only if needed
[INFO]      * Container no default ulimit override: suspicious_mccarthy
```

- **The Docker security operations**:

```
[INFO] 6  - Docker Security Operations
[INFO] 6.6 - Avoid image sprawl
[INFO]      * There are currently: 2 images
[INFO] 6.7 - Avoid container sprawl
[INFO]      * There are currently a total of 2 containers, with 1 of them currently running
```

Wow! A lot of output and tons to digest; but what does it all mean? Let's take a look and break down each section.

Understanding the output

There are three types of output that we will see:

- `[PASS]`: These items are solid and good to go. They don't need any attention, but are good to read to make you feel warm inside. The more of these, the better!
- `[INFO]`: These are items that you should review and fix if you feel they are pertinent to your setup and security needs.
- `[WARN]`: These are items that need to be fixed. These are the items we don't want to be seeing.

Remember, we had the six main topics that were covered in the scan:

- The host configuration
- The Docker daemon configuration
- The Docker daemon configuration files
- Container images and build files
- Container runtime
- The Docker security operations

Let's take a look at what we are seeing in each section of the scan. These scan results are from a default Ubuntu Docker host with no tweaks made to the system at this point. We want to focus again on the `[WARN]` items in each section. Other warnings may come up when you run yours, but these will be the ones that come up most if not for everyone at first.

- **Host configuration**:

  ```
  [WARN] 1.1  - Create a separate partition for containers
  ```

  For this one, you will want to `map /var/lib/docker` to a separate partition.

  ```
  [WARN] 1.8  - Failed to inspect: auditctl command not found.
  [WARN] 1.9  - Failed to inspect: auditctl command not found.
  [WARN] 1.10 - Failed to inspect: auditctl command not found.
  [WARN] 1.13 - Failed to inspect: auditctl command not found.
  [WARN] 1.18 - Failed to inspect: auditctl command not found.
  ```

- **The Docker daemon configuration**:

```
[WARN] 2.2  - Restrict network traffic between containers
```

  By default, all the containers running on the same Docker host have access to each other's network traffic. To prevent this, you would need to add the `--icc=false` flag to the Docker daemon's start up process.

```
[WARN] 2.7  - Do not use the aufs storage driver
```

  Again, you can add a flag to your Docker deamon start up process that will prevent Docker from using the `aufs` storage driver. By using `-s <storage_driver>` on your Docker daemon startup, you can tell Docker not to use `aufs` for storage. It is recommended that you use the best storage driver for the OS on the Docker host you are using.

- **The Docker daemon configuration files**:

  If you are using the stock Docker daemon, you should not see any warnings. If you have customized the code in some way, you may get warnings here. This is one area you hope to never see warnings.

- **Container images and build files**:

```
[WARN] 4.1  - Create a user for the container
[WARN]      * Running as root: suspicious_mccarthy
```

  This is stating that the container named `suspicious_mccarthy` is running as the `root` user and it is recommended to create another user to run your containers.

- **Container Runtime**:

```
[WARN] 5.1:   - Verify AppArmor Profile, if applicable
[WARN]      * No AppArmorProfile Found: suspicious_mccarthy
```

  This states that the container named `suspicious_mccarthy` does not have `AppArmorProfile`, which is the additional security provided in Ubuntu in this case.

```
[WARN] 5.3  - Verify that containers are running only a single
main process
[WARN]      * Too many processes running: suspicious_mccarthy
```

This error is pretty straightforward. You will want to make sure you are only running one process per container. If you are running more than one, you will want to spread them out across multiple containers and use container linking.

```
[WARN] 5.4  - Restrict Linux Kernel Capabilities within containers
[WARN]      * Capabilities added: CapAdd=[audit_control] to
suspicious_mccarthy
```

This is stating that the `audit_control` capability has been added to this running container. You can use `--cap-drop={}` from your `docker run` command to remove additional capabilities on a container.

```
[WARN] 5.6  - Do not mount sensitive host system directories on
containers
[WARN]      * Sensitive directory /etc mounted in: suspicious_
mccarthy
```

This again goes back to looking at mounting the items inside the containers as read-only. The `--read-only` flag would come in handy in this scenario, when you issue your `docker run` command.

```
[WARN]      * Sensitive directory /lib mounted in: suspicious_
mccarthy
```

This too goes back to looking at mounting the items inside the containers as read-only. The `--read-only` flag would come in handy in this scenario, when you issue your `docker run` command.

```
[WARN] 5.7  - Do not run ssh within containers
[WARN]      * Container running sshd: suspicious_mccarthy
```

It is straight to the point. No need to run SSH inside your containers. You can do everything you want to with your containers using the tools provided by Docker. Ensure that SSH is not running in any container.

```
[WARN] 5.10 - Do not use host network mode on container
[WARN]      * Container running with networking mode 'host':
suspicious_mccarthy
```

The issue with this one is that, when the container was running, the `--net=host` switch was passed along. It is not recommended to use this, as it allows the container to open low port numbers as well as access networking services on the Docker host.

```
[WARN] 5.11 - Limit memory usage for the container
[WARN]      * Container running without memory restrictions:
suspicious_mccarthy
```

By default, the containers don't have memory restrictions. This can be dangerous if you are running multiple containers per Docker host. You can use the `-m` switch while issuing your `docker run` commands to limit the containers to a certain amount of memory. Values are set in megabytes (that is, 512 MB or 1024 MB).

```
[WARN] 5.12 - Set container CPU priority appropriately
[WARN]      * The container running without CPU restrictions:
suspicious_mccarthy
```

Like the memory option, you can also set the CPU priority on a per container basis. This can be done using the `-c` switch while issuing your docker run command. The CPU share is based off of the number 1024. So, half would be 512 and 25% would be 256. Use 1024 as the base number to determine the CPU share.

```
[WARN] 5.13 - Mount container's root filesystem as readonly
[WARN]      * Container running with root FS mounted R/W:
suspicious_mccarthy
```

You really want to be using your containers as mutable environments; meaning they don't write any data inside them. Data should be written out to volumes. Again, you can use the `--read-only` switch, followed by the `-v /` switch to specify that the `root` directory is read-only for the running container.

```
[WARN] 5.16 - Do not share the host's process namespace
[WARN]      * Host PID namespace being shared with: suspicious_
mccarthy
```

This error arises when you use the `--pid=host` switch. It is not recommended to use this switch, as it breaks the isolation of processes between the container and Docker host.

- **The Docker security operations**:

  Again, another section you hope to or never should see warnings if you are using stock Docker. Mostly here you will see information and should review them to make sure it's all kosher.

Summary

In this chapter, we covered some aspects of Docker security. First, we took a look at containers versus typical virtual machines with regards to security. We looked at the good, the not so bad, and what to look out for.

We then took a look at what Docker commands we can use for security purposes. We first took a look at read-only containers, so we can minimize what we are exposing to other containers. We then viewed what is done to the images that you have running. It is important to know what is done on these containers, so you have a trail of activity.

Next, we took a look at the Center for Internet Security guidelines for Docker. This guide will assist you in setting up multiple aspects of your Docker environment. Lastly, we took a look at the Docker bench for security. We looked at how to get it up and running and ran through an example of what the output would look like once it has been run. We then took a look at the said output to see what all it meant. Remember the six items that the application covered: the host configuration, Docker daemon configuration, Docker daemon configuration files, container images and build files, container runtime, and Docker security operations.

In the next chapter, we will be taking a look at Docker Machine. Docker Machine allows you to create Docker hosts locally on items such as VirtualBox or VMWare Fusion or to cloud providers such as Amazon AWS, Microsoft Azure, DigitalOcean, as well as others. Saving time is the key here. Instead of having to go to a host, spin up a virtual machine, and get Docker installed on it, Docker Machine will do it all for you and give you more time to do what you should be doing.

6
Docker Machine

In this chapter, we will take a look at Docker Machine. Docker Machine is a tool that supersedes boot2docker. It can be used to create Docker hosts on various platforms, including locally or in a cloud environment. You can control your Docker hosts with it as well. Let's take a look at what we will be covering in this chapter:

- Installing Docker Machine
- Using Docker Machine to set up the Docker hosts
- Various Docker commands

Installation

Installing Docker Machine is very straightforward. There is a simple `curl` command to run and install it. It is recommended to install Docker Machine in `/usr/local/bin`, as this will allow you to issue the Docker Machine commands from any directory on your machine:

```
$ curl -L https://github.com/docker/machine/releases/download/v0.4.0/
docker-machine_linux-amd64 > /usr/local/bin/docker-machine
```

After issuing the `curl` command, you need to set the permissions in the `docker-machine` file that was just created in `/usr/local/bin/`:

```
$ chmod +x /usr/local/bin/docker-machine
```

You can then verify that Docker Machine is installed by issuing a simple `docker-machine` command:

```
$ docker-machine --help
```

You should get back all the commands and switches you can use while operating the `docker-machine` command.

Now these instructions are great if you are on Linux. But what if you are using Mac or even Windows? Then, you would want to use the Docker Toolbox to do your installation. This will not only install Docker Machine, but other pieces of the Docker ecosystem as well. To view a list of what all comes in the Docker Toolbox per platform, visit `https://www.docker.com/docker-toolbox`.

Using Docker Machine

Let's take a look at how we can use Docker Machine to deploy Docker hosts on your local infrastructure, on your own machine, as well as on various cloud providers.

Local VM

Docker Machine uses the `--driver` switch to specify the location you want to set up and install the Docker host. So, we can set up a Docker host in VirtualBox:

```
$ docker-machine create --driver virtualbox <name>
```

Or, we can set it up on VMware Fusion:

```
$ docker-machine create --driver vmwarefusion <name>
```

The previous command is structured as the `docker-machine` command, followed by what we want to do: `create`. We will use the `--driver` switch next. Then, we need to specify where we are going to place the Docker host. In our case, we specified `virtualbox` and `vmwarefusion`. Lastly, we need to give the Docker host a name. This name is to be unique; so when you issue other Docker Machine commands, they are distinguishable.

There are various other switches we can use to tell how much memory the Docker host to use and also how much disk space to use as well. You can see all the available switches by issuing our trustworthy and helpful `docker-machine create --help` command. Remember that everything has a `--help` switch that can be utilized to gain more information to get the help you need. It should be the first thing you turn to when you are looking for assistance.

Cloud environment

Now, let's take a look at how we deploy to a cloud environment of our choosing. When you start deploying to cloud environments, it can get tricky, as it requires some form of authentication to ensure you are who you say you are. For example, DigitalOcean requires an access token to launch a Docker host in its system. We will be taking a look at how we can deploy a Docker host in AWS.

For AWS, we need a couple of switches. We would need to get the information from AWS before we can deploy to this cloud provider:

- `--amazonec2-access-key`
- `--amazonec2-secret-key`
- `--amazonec2--vpc-id`
- `--amazonec2-zone`
- `--amazonec2-region`

We can create these drivers by executing the following command:

```
$ docker-machine create \
   --driver amazonec2 \
   --amazonec2-access-key <aws_access_key> \
   --amazonec2-secret-key <aws_secret_key> \
   --amazonec2-vpc-id <vpc_id> \
   --amazonec2-subnet-id <subnet_id> \
   --amazonec2-zone <zone> \
     <name>
```

Docker Machine commands

Now that we can deploy Docker hosts locally and to the cloud environments, we need to know how we can manage and manipulate these Docker hosts. Let's take a look at all the commands Docker Machine has to offer.

Note that as we previously created these hosts we were given output on how to target them for use with Docker Machine.

On running the `docker-machine create` command, you should receive an output similar to this:

```
INFO[0041] To point your Docker client at it, run this in your shell:
$(docker-machine env dev2)
```

This is how you can set the default to target Docker hosts with Docker Machine. Keep this in mind, when we are looking at the following commands.

active

You can use the `active` subcommand to see which Docker host is currently active and commands that you execute will be executed on that Docker host:

```
$ docker-machine active
dev2
```

config

You can use the `config` subcommand to view what the current configuration is for the Docker Machine setup on the currently active host:

```
$ docker-machine config
--tls --tlscacert=/Users/scott/.docker/machine/machines/dev2/
ca.pem --tlscert=/Users/scott/.docker/machine/machines/dev2/cert.
pem --tlskey=/Users/scott/.docker/machine/machines/dev2/key.pem
-H=tcp://192.168.50.158:2376
```

env

You can view the environmental variables on each Docker host with the `env` subcommand:

```
$ docker-machine env
export DOCKER_TLS_VERIFY=1
export DOCKER_CERT_PATH=/Users/spg14/.docker/machine/machines/dev2
export DOCKER_HOST=tcp://192.168.50.158:2376
```

inspect

You can inspect each Docker host using the Docker Machine `inspect` subcommand. This subcommand will give you a lot of information on the Docker host, such as the certificate paths, Swarm host, disk size, memory, CPUs, and much more:

```
$ docker-machine inspect
{
    "DriverName": "vmwarefusion",
    "Driver": {
        "MachineName": "dev2",
        "IPAddress": "192.168.50.158",
        "Memory": 1024,
```

```
        "DiskSize": 20000,
        "CPUs": 8,
        "ISO": "/Users/scott/.docker/machine/machines/dev2/boot2docker-
1.5.0-GH747.iso",
        "Boot2DockerURL": "",
        "CaCertPath": "/Users/scott/.docker/machine/certs/ca.pem",
        "PrivateKeyPath": "/Users/scott/.docker/machine/certs/ca-key.
pem",
        "SwarmMaster": false,
        "SwarmHost": "tcp://0.0.0.0:3376",
        "SwarmDiscovery": "",
        "CPUS": 8
    },
    "CaCertPath": "/Users/scott/.docker/machine/certs/ca.pem",
    "ServerCertPath": "",
    "ServerKeyPath": "",
    "PrivateKeyPath": "/Users/scott/.docker/machine/certs/ca-key.pem",
    "ClientCertPath": "",
    "SwarmMaster": false,
    "SwarmHost": "tcp://0.0.0.0:3376",
    "SwarmDiscovery": ""
}
```

ip

The `ip` subcommand will give you the IP address of the active host you are pointing to with Docker Machine:

```
$ docker-machine ip <name>
192.168.50.158
```

kill

If a host is acting up, you can kill the Docker hosts with the `kill` subcommand of Docker Machine:

```
$ docker-machine kill
INFO[0000] Forcibly halting dev2...
```

ls

You can use the `ls` subcommand to view all the running Docker hosts you have used to create with Docker Machine. The information will include:

- The name of the host
- Whether the machine is active
- The driver that is being used
- The state of the host
- The URL that is being used for communication
- If the host is a part of the Docker Swarm cluster, then that information will be shown as well

Let's take a look at a sample command output when you use `docker-machine ls`:

```
$ docker-machine ls
NAME   ACTIVE   DRIVER          STATE     URL
SWARM
dev                        virtualbox        Stopped
dev2       *             vmwarefusion Running   tcp://192.168.50.158:2376
```

As you can see, you get the list of Docker hosts you can control. As well as the driver, its state, URL, and its part of a Swarm cluster.

restart

You can restart the hosts as well using the `restart` subcommand:

```
$ docker-machine restart <name>
INFO[0000] Gracefully restarting dev2...
```

rm

You can remove the hosts you no longer need by using the `rm` subcommand of Docker Machine:

```
$ docker-machine rm <name>
```

scp

There are multiple ways to use the Docker Machine `scp` command. You can copy files or folders from the local host to a Docker host:

```
$ docker-machine scp <file_name> <name>:/<path>/<to>/<folder>/
```

It can be copied from one machine to another:

```
$ docker-machine scp <host1>:/<path>/<to>/<file>
<host2>:/<path>/<to>/<folder>/
```

It can also be copied from the machine back to the host:

```
$ docker-machine scp <name>:/<path>/<to>/<file> .
```

ssh

You can SSH into your Docker hosts as well by using the `ssh` subcommand. This can be useful if you need to troubleshoot why the commands you push against your hosts might not be working:

```
$ docker-machine ssh <name>
```

start

The `start` subcommand can be used to start the Docker hosts that have been stopped:

```
$ docker-machine start <name>
INFO[0000] Starting dev2...
```

stop

You can stop the hosts as well by using the `stop` subcommand:

```
$ docker-machine stop <name>
INFO[0000] Gracefully shutting down dev2...
```

upgrade

If you have a Docker host that is running Docker version 1.7 (let's say) and you want to upgrade it to the latest version, you could use the `upgrade` subcommand of Docker Machine:

```
$ docker-machine upgrade <name>
```

This will upgrade the version of Docker that is running on the Docker hostname you provide.

url

The `url` subcommand will give you the URL that is being used for communication for the Docker host:

```
$ docker-machine url <name>
tcp://192.168.50.158:2376
```

TLS

Docker Machine also has the option to run everything over TLS. This is the most secure way of using Docker Machine to manage your Docker hosts. This setup can be tricky if you start using your own certificates. By default, Docker Machine stores your certificates that it uses in `/Users/<user_id>/.docker/machine/certs/`. You can view these items simply by running:

```
$ docker-machine --help
```

This will give you a `global Options` section at the bottom of the listing that lists this information. These are the locations of the intermediate certificate, intermediate key, and the certificate that Docker Machine uses as well as its corresponding key. You would need to update these files with your own certificates if you don't want to be using the self-signed certificates that Docker Machine creates.

Summary

In this chapter, we looked at Docker Machine. We first looked at how to use Docker Machine to create the Docker hosts locally on virtualization software such as VirtualBox or VMware Fusion. We also looked at how to use Docker Machine to deploy Docker hosts to your cloud environments.

We then took a look at all the commands that are in the Docker Machine Toolbox. With all these commands, you can manage your entire fleet of Docker hosts. You can manipulate them from creating new Docker hosts to managing all the configuration aspects of the Docker hosts. We really dove deep into all the Docker Machine commands, so you should have a good understanding of this Docker component.

In the next chapter, we will be looking at Docker Compose. Docker Compose is very complex and has a lot of pieces that you can leverage to your advantage. We will be focusing very heavily on Docker Compose and it's a core piece of the Docker ecosystem that you will find yourself using almost daily. Docker Compose is very powerful and very useful with all the aspects of managing Docker.

7
Docker Compose

In this chapter, we will be taking a look at Docker Compose. We will break the chapter down into the following sections:

- Installing Docker Compose
- Docker Compose YAML file
- Docker Compose usage
- The Docker Compose commands
- The Docker Compose examples

Installing Docker Compose

Let's take a look at how we can get Docker Compose installed on to our machine, so we can start utilizing its full feature set and power.

Installing on Linux

Let's take a look at how easy it is to install on Linux:

```
$ curl -L https://github.com/docker/compose/releases/download/VERSION_
NUM/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
```

The reason we install this in the `/usr/local/bin/` folder is that this folder is where global commands are stored in Linux. For example, when you type a command and hit *Enter*, Linux does a search in a few common areas to see if the command you typed exists. If it does, execution starts, else you will get an error stating that the command can't be found. This makes it easier, so you don't have to use full paths to the `docker-compose` binary or be in a certain directory each time to run it:

```
$ chmod +x /usr/local/bin/docker-compose
```

This will set the downloaded binary to executable.

Installing on OS X and Windows

The installation for OS X and Windows is different than it originally was. For OS X in particular, the installation was done using the `curl` command. Now, Docker has created what they call Docker Toolbox that is used to install not only Docker Compose but multiple components of the service for you to use.

To install Docker Compose on these platforms, we need the Docker Toolbox installer. This can be found on the Docker website. Simply download the installer for your platform and follow the installer instructions to get up and running.

Docker Compose YAML file

For building your YAML files, I definitely recommend looking at the Docker documentation for this. There are a plethora of items that can be added to your `docker-compose.yml` file and it's always changing.

The key thing to note about a basic YAML file is that it has to contain either a name for each service, an `image:`, or a `build:` section. There are many other options to do inside the compose file, such as:

- Container linking
- Exposing ports
- Specifying the volumes to be used
- Specifying the environmental variables
- Setting the DNS servers to be used
- Setting the log driver to be used and much more

The Docker Compose usage

We can start by using the ever-so-helpful `--help` switch on the `docker-compose` command. We will see a lot of output and will sift through it after the following output:

```
$ docker-compose --help

Define and run multi-container applications with Docker.

Usage:
  docker-compose [options] [COMMAND] [ARGS...]
```

```
  docker-compose -h|--help

Options:
  -f, --file FILE           Specify an alternate compose file (default:
docker-compose.yml)
  -p, --project-name NAME   Specify an alternate project name (default:
directory name)
  --verbose                 Show more output
  -v, --version             Print version and exit

Commands:
  build              Build or rebuild services
  help               Get help on a command
  kill               Kill containers
  logs               View output from containers
  port               Print the public port for a port binding
  ps                 List containers
  pull               Pulls service images
  restart            Restart services
  rm                 Remove stopped containers
  run                Run a one-off command
  scale              Set number of containers for a service
  start              Start services
  stop               Stop services
  up                 Create and start containers
  migrate-to-labels  Recreate containers to add labels
  version            Show the Docker-Compose version information
```

The Docker Compose options

Looking at the `help` output, we can see that the list is categorized as `Usage`, `Options`, and `Commands`. The `Usage` section is how you will need to structure your commands to run them successfully. Next is the `Options` section that we will look at now:

```
Options:
  -f, --file FILE           Specify an alternate compose file (default:
docker-compose.yml)
```

```
  -p, --project-name NAME   Specify an alternate project name (default:
directory name)
  --verbose                 Show more output
  -v, --version             Print version and exit
```

So, as we can see from the previous output of the `docker-compose --help` command, there are two sections: an `Options` section as well as a `Commands` section. We will first look at the items in the `Options` section and next look at the `Commands` section.

There are four items in the `Options` section:

- `-f`: If you are using Docker Compose outside the folder where the `docker-compose.yml` file exists or if you are not naming it `docker-compose.yml`, then you will need to specify the `-f` flag. By default, when you initiate the Docker Compose commands, they are meant to be done in the directory where your `docker-compose.yml` file is located. This helps in keeping things consistent, organized, as well as less convoluted.
- `-p`, `--project-name`: The `-p` option will allow you to give a name to your project. By default, Docker Compose uses the name of the folder you are currently running the Docker Compose commands from. This allows you to override it.
- `--verbose`: The `--verbose` switch allows you to run Docker Compose in a way that you can see the output of items about the image(s) being used, such as:
    - The command used to start the containers
    - The CPU shares being used in the container
    - The domain name being used
    - Whether an entry point was used and if so, what it is
- `-v`, `--version`: This will simply print the version number of the Docker Compose client being used.

The Docker Compose commands

We can tell by running the previous `docker-compose --help` command that there are many subcommands that can be used with the main `docker-compose` command. Let's break them down individually and provide examples of each subcommand, starting at the top and working our way down the list. Remember that there are also switches for each subcommand that can be found using the `--help` option. For example, `docker-compose <subcommand> --help`. These commands will also seem very similar as the commands we saw in the Docker commands section in *Chapter 4*, *Managing Containers*. Also, note that some of these commands need to be run in the folder where `docker-compose` and/or the Dockerfile for that service are located.

For the command examples, we will be using the following as the contents of our `docker-compose.yml` file called `example 1`:

```
master:
  image:
    scottpgallagher/galeramaster
  hostname:
    master
  ports:
    - "3306:3306"
node1:
  image:
    scottpgallagher/galeranode
  hostname:
    node1
  links:
    - master
node2:
  image:
    scottpgallagher/galeranode
  hostname:
    node2
  links:
    - master
```

We will also be creating this file (`example 2`):

```
web:
  build: .
  command: php -S 0.0.0.0:8000 -t /code
  ports:
    - "8000:8000"
  links:
```

```
      - db
    volumes:
      - .:/code
  db:
    image: orchardup/mysql
    environment:
      MYSQL_DATABASE: wordpress
```

We will create our Dockerfile for this `docker-compose.yml` file:

```
FROM orchardup/php5
ADD . /code
```

build

The `build` command of Docker Compose is used when you have changed the contents of a Dockerfile that you are using and need to rebuild one of the systems in the `docker-compose.yml` file.

For example, if you review our `example 2` code, in the previous section, we have a `web` container that we are specifying in our `docker-compose.yml` file. Now, if were to update the contents of the Dockerfile, we would need to rebuild the container named `web`, so it knows about the change. We may want to change the image we are using or, if the image has been updated, we would want to do a rebuild of the `web` container:

```
$ docker-compose build web
```

It will look for the name `web` in the `docker-compose.yml` file, then jump to the Dockerfile, and rebuild the `web` container based on the contents of the Dockerfile. This also can be useful; if the container in question has disappeared, you can rebuild just that image. There is just one switch that can be used with this subcommand and that is `--no-cache`, which allows you to build the image without using local cache.

kill

The `kill` subcommand does exactly what its name suggests. It will kill a running container without gracefully stopping it. This can have unattended consequences with the data that is being written, such as MySQL database tables, to at the time of issuing this command. Remember that containers are made to be immutable environments; but if you start diving into the volumes, then you are referring to data that is mutable and might change. In an event where you do have a volume and data is being written to it, the best practice would be to use the `stop` subcommand.

Using the example 2 code in the *The Docker Compose commands* section, let's say that both the `web` and `db` containers are running and we want to stop the `web` container. In this case, we could use the `kill` subcommand:

```
$ docker-compose kill web
```

logs

Next up is logs! This subcommand will print the output from the specified service. Let's take a look at example 1. We have three running containers in this case: `master`, `node1`, and `node2`. We can tell that `node2` is doing something strange with its MySQL replication and we need to see whether we can find out why. Our first stop is to check its logs:

```
$ docker-compose logs node2
```

You will receive an output similar to the following (but not exactly the same):

```
node2_1 |    at gcomm/src/gmcast.cpp:connect_precheck():282
node2_1 | 150904 16:47:56 [ERROR] WSREP: gcs/src/gcs_core.cpp:long int
gcs_core_open(gcs_core_t*, const char*, const char*, bool)():206: Failed
to open backend connection: -131 (State not recoverable)
node2_1 | 150904 16:47:56 [ERROR] WSREP: gcs/src/gcs.cpp:long int gcs_
open(gcs_conn_t*, const char*, const char*, bool)():1379: Failed to
open channel 'my_wsrep_cluster' at 'gcomm://master': -131 (State not
recoverable)
node2_1 | 150904 16:47:56 [ERROR] WSREP: gcs connect failed: State not
recoverable
node2_1 | 150904 16:47:56 [ERROR] WSREP: wsrep::connect() failed: 7
node2_1 | 150904 16:47:56 [ERROR] Aborting
node2_1 |
node2_1 | 150904 16:47:56 [Note] WSREP: Service disconnected.
node2_1 | 150904 16:47:57 [Note] WSREP: Some threads may fail to exit.
node2_1 | 150904 16:47:57 [Note] mysqld: Shutdown complete
node2_1 |
```

We can see that this node has an issue talking to `master` and shuts down its MySQL. Now that sure helps us!

You will notice that the output is colored as well. This is something you will see while using Docker Compose, as it separates running containers using different colors. You can get the output of the logs without color as well by appending the `--no-color` switch to the command:

```
$ docker-compose logs --no-color node2


node2_1 |    at gcomm/src/gmcast.cpp:connect_precheck():282
node2_1 | 150904 16:47:56 [ERROR] WSREP: gcs/src/gcs_core.cpp:long int
gcs_core_open(gcs_core_t*, const char*, const char*, bool)():206: Failed
to open backend connection: -131 (State not recoverable)
node2_1 | 150904 16:47:56 [ERROR] WSREP: gcs/src/gcs.cpp:long int gcs_
open(gcs_conn_t*, const char*, const char*, bool)():1379: Failed to
open channel 'my_wsrep_cluster' at 'gcomm://master': -131 (State not
recoverable)
node2_1 | 150904 16:47:56 [ERROR] WSREP: gcs connect failed: State not
recoverable
node2_1 | 150904 16:47:56 [ERROR] WSREP: wsrep::connect() failed: 7
node2_1 | 150904 16:47:56 [ERROR] Aborting
node2_1 |
node2_1 | 150904 16:47:56 [Note] WSREP: Service disconnected.
node2_1 | 150904 16:47:57 [Note] WSREP: Some threads may fail to exit.
node2_1 | 150904 16:47:57 [Note] mysqld: Shutdown complete
node2_1 |
```

port

The `port` subcommand allows you to use Docker Compose to get you the public-facing port from the private port the server is displaying. This can be useful if you either forget what port privately maps or what port publicly maps. If you have used autoassigned ports, then you might want to be looking that information up as well. The command is very straightforward. Again, looking at example 1, we will this time look at `master`. The thing to note with this command is that the container must be running in order to get this information. The structure of this command is:

```
$ docker-compose <name-from-compose> <port-to-lookup>
$ docker-compose port master 3306
```

There are also two switches to utilize with this subcommand:

- `--protocol`: This is used to display either the TCP or UDP port to look up the port that you specify on the command line. This will default to display TCP. The reason for this switch would be if you are looking for the UDP port:

  ```
  $ docker-compose --port udp master 3306
  ```

- `--index`: This is used if you have scaled containers and you want to look up what a certain image in the list is using. For example, if we were specifying two masters, we could do:
  - `$ docker-compose --index 1 master 3306`: This would display the public-facing port for the `master` container in index position 1.
  - `$ docker-compose --index 2 master 3306`: This would display the information for the `master` container in index spot two.

We know for this example that port `3306` is being used for the MySQL service. However, if you don't know what ports it was running on the private or public side, you can use the `ps` subcommand that we will be looking at next.

ps

The Docker Compose `ps` subcommand can be used to display information on the containers running within a particular Docker Compose folder. For instance, in our last subcommand, we talked about not knowing the private port. This command will help us get that information. We will now take a look at the output of the `docker-compose ps` subcommand using example 2 code in the *The Docker Compose commands* section:

```
$ docker-compose ps

          Name                        Command                       State
Ports
-------------------------------------------------------------------------------
--------------------------------
galeracompose_master_1     /entrypoint.sh             Up
0.0.0.0:3306->3306/tcp,

4444/tcp, 4567/tcp,

4568/tcp, 53/tcp,
```

```
53/udp, 8300/tcp,

8301/tcp, 8301/udp,

8302/tcp, 8302/udp,

8400/tcp, 8500/tcp
galeracompose_node1_1      /entrypoint.sh             Exit 1
galeracompose_node2_1      /entrypoint.sh             Exit 137
```

We can get a lot of information from this output. We can get the name of the containers running. These names are assigned based upon `folder_name` + `_name_used_in_yml_file` + `_<number_of_each_name_running>`. For example, `galeracompose_master_1`, where:

- `galeracompose` is our folder name
- `master` is the name being used in the `docker-compose.yml` file
- `1` is how many times this container is being run

We also see the command that is running inside the container as well as the state of each container. In our earlier example, we see that one container is up and two are in an `Exit` status, which means they are off. From the one that is up, we see all the ports that are being utilized on the backend, including the protocol. Then, we see the ports that are exposed to the outside and also the backend port they are connected to.

When you use various commands with Docker Compose, you can specify either the name given from the output using the `ps` subcommand or by the name given in the `docker-compose.yml` file.

pull

The `pull` subcommand can be used in two ways. One you could run:

```
$ docker-compose pull
```

Or you could run:

```
$ docker-compose pull <service_name>
```

What's the difference? The difference in the first one is that it will pull all the images that are referenced in the `docker-compose.yml` file. In the second one, it will pull just the image that is specified for the service asked to be pulled.

If we look back at example 1 in the *Docker Compose commands* section, we have `master`, `node1`, and `node2` in our `docker-compose.yml` file. If we wanted to retrieve all the images, we would use the first example. If we just wanted the image being used by `master`, we would use the second one:

```
$ docker-compose pull master
```

Remember that these commands need to be run in the folder where the `docker-compose.yml` file is located.

restart

Restart does exactly what it says it does. As with the `pull` subcommand, it can be used in two ways. You can run:

```
$ docker-compose restart
```

It will restart all the containers that are being used in the `docker-compose.yml` file. You can also specify which container to restart:

```
$ docker-compose restart <service>
```

Again, using example 1 in the *The Docker Compose commands* section, we only want to restart one of the node services:

```
$ docker-compose restart node1
```

The `restart` command will only restart the containers that are currently running. If a container is in an exit state, then it won't start that container up to a running state.

rm

The `rm` subcommand can be used to remove containers for specific Docker Compose instances. By default, it will ask you to confirm whether you really want to remove the container in question. It is a good practice to use the subcommand in this way. However, if you are comfortable enough, you can also use the `-f` switch with the subcommand to force removal and you won't be prompted to for yes as an answer:

```
$ docker-compose rm <service>
$ docker-compose rm node2
Going to remove galeracompose_node2_1
Are you sure? [yN] y
Removing galeracompose_node2_1... done
```

You can use this command, as we have seen with the previous commands, without specifying a service name. If you do so, it will prompt you to remove each of the stopped containers. It will not try to remove the containers that are running however. Again, you could use the `-f` switch to specify the removal of all the stopped containers without asking for approval.

run

The `run` subcommand is used to run a one-time command against a service, not against an already running container. When you use the `run` subcommand, you are actually starting up a new container and executing the specified command. This is one command that you do need to pay attention to, including the switches that are available for the subcommand.

Specifically, there are two to remember:

- `--no-deps`: This will not start up containers that may be linked to the container being used with the `run` subcommand. By default, when you use the `run` subcommand, any linked containers will start up as well.
- `--service-ports`: By default, ports that are being specified in the `docker-compose.yml` file are not exposed during the execution of the run subcommand. This is to avoid issues with the ports that are already in use. However, this switch will allow you to expose the ports that are being specified. This can be helpful if the ports in question aren't already being exposed.

The structure of the subcommand is as follows:

```
$ docker-compose run <service> <command>
```

scale

The `scale` subcommand allows you just to do that: scale. With the `scale` subcommand, you can specify how many instances you want to start up. Using example 1, if we want to load up a bunch of nodes, we could do that using the `scale` subcommand:

```
$ docker-compose scale node1=3
```

This would fire up three nodes and link them back to the `master` container. You can also specify multiple containers to scale per line as well. If we had a difference in `node1` and `node2`, we could scale them accordingly on the same line.

```
$ docker-compose scale node1=3 node2=3
```

start

We will use this for our example with the `start` subcommand:

```
$ docker-compose ps

          Name                        Command                       State
Ports
-------------------------------------------------------------------------
-----------------------------------
galeracompose_master_1      /entrypoint.sh              Exit 137
galeracompose_node2_run_1   /entrypoint.sh              Up
3306/tcp, 4444/tcp,

4567/tcp, 4568/tcp,

53/tcp, 53/udp, 8300/tcp,

8301/tcp, 8301/udp,

8302/tcp, 8302/udp,

8400/tcp, 8500/tcp
```

From the preceding `ps` subcommand, we can see that the `master` node is stopped. That's not good! We need to get it started as soon as possible:

```
$ docker-compose start master


$ docker-compose ps
          Name                        Command                       State
Ports
-------------------------------------------------------------------------
-----------------------------------
galeracompose_master_1      /entrypoint.sh              Up
0.0.0.0:3306->3306/tcp,

4444/tcp, 4567/tcp,

4568/tcp, 53/tcp, 53/udp,

8300/tcp, 8301/tcp,
```

```
8301/udp, 8302/tcp,

8302/udp, 8400/tcp,

8500/tcp
galeracompose_node2_run_1   /entrypoint.sh              Up
3306/tcp, 4444/tcp,

4567/tcp, 4568/tcp,

53/tcp, 53/udp, 8300/tcp,

8301/tcp, 8301/udp,

8302/tcp, 8302/udp,

8400/tcp, 8500/tcp
```

Phew, it is much better now! Let's take a look at what we need to do if we need to stop a running container.

stop

The `stop` subcommand stops running containers gracefully. Using our example from the last subcommand, let's stop the `master` container:

```
$ docker-stop master


docker-compose ps
          Name                        Command                       State
Ports
-------------------------------------------------------------------------
-----------------------------------
galeracompose_master_1      /entrypoint.sh              Exit 137
galeracompose_node2_run_1   /entrypoint.sh              Up
3306/tcp, 4444/tcp,

4567/tcp, 4568/tcp,

53/tcp, 53/udp, 8300/tcp,
```

```
8301/tcp, 8301/udp,

8302/tcp, 8302/udp,

8400/tcp, 8500/tcp
```

up

The `up` subcommand is used to start all the containers specified in a `docker-compose.yml` file. It can also be used to start up a single container as well from a compose file. By default, when you issue the `up` subcommand, it will keep everything in the foreground. However, you can use the `-d` switch to push all that information into a daemon and just get information on the container names on the screen:

Let's use example 2 in this test case. We will take a look at `docker-compose up -d` and `docker-compose up`:

```
$ docker-compose up -d


Starting wordpresstest_db_1...
Starting wordpresstest_web_1...


$ docker-compose up
Starting wordpresstest_db_1...
Starting wordpresstest_web_1...
Attaching to wordpresstest_db_1, wordpresstest_web_1
db_1  | 150905 14:39:02 [Warning] Using unique option prefix key_buffer
instead of key_buffer_size is deprecated and will be removed in a future
release. Please use the full name instead.
db_1  | 150905 14:39:02 [Warning] Using unique option prefix key_buffer
instead of key_buffer_size is deprecated and will be removed in a future
release. Please use the full name instead.
db_1  | 150905 14:39:03 [Warning] Using unique option prefix key_buffer
instead of key_buffer_size is deprecated and will be removed in a future
release. Please use the full name instead.
db_1  | 150905 14:39:03 [Warning] Using unique option prefix myisam-
recover instead of myisam-recover-options is deprecated and will be
removed in a future release. Please use the full name instead.
........
db_1  | 150905 14:41:36 [Note] Plugin 'FEDERATED' is disabled.
db_1  | 150905 14:41:36 InnoDB: The InnoDB memory heap is disabled
```

```
db_1  | 150905 14:41:36 InnoDB: Mutexes and rw_locks use GCC atomic
builtins
db_1  | 150905 14:41:36 InnoDB: Compressed tables use zlib 1.2.3.4
db_1  | 150905 14:41:36 InnoDB: Initializing buffer pool, size = 128.0M
db_1  | 150905 14:41:36 InnoDB: Completed initialization of buffer pool
db_1  | 150905 14:41:36 InnoDB: highest supported file format is
Barracuda.
db_1  | 150905 14:41:36  InnoDB: Waiting for the background threads to
start
db_1  | 150905 14:41:37 InnoDB: 5.5.38 started; log sequence number
1595675
db_1  | 150905 14:41:37 [Note] Server hostname (bind-address): '0.0.0.0';
port: 3306
db_1  | 150905 14:41:37 [Note]   - '0.0.0.0' resolves to '0.0.0.0';
db_1  | 150905 14:41:37 [Note] Server socket created on IP: '0.0.0.0'.
db_1  | 150905 14:41:37 [Note] Event Scheduler: Loaded 0 events
db_1  | 150905 14:41:37 [Note] /usr/sbin/mysqld: ready for connections.
db_1  | Version: '5.5.38-0ubuntu0.12.04.1-log'  socket: '/var/run/mysqld/
mysqld.sock'  port: 3306  (Ubuntu)
```

You can see a huge difference. Remember that, if you don't use the `-d` switch and hit *Ctrl + C* in the terminal window, it will start shutting down the running containers. While it's good for testing purposes, if you are going into a production environment, it is recommended to use the `-d` switch.

version

The `version` subcommand will give you the version of Docker Compose that you are running. It's very straightforward and can also be utilized with the `-v` switch:

```
$ docker-compose version
$ docker-compose -v
```

The difference is that the subcommand version will show you a little more information such as the `docker-py` version, Python version, and OpenSSL version, while the `-v` switch will just show you the version of Docker Compose.

Docker Compose – examples

In this section, we will take a look at some examples and break them to understand what we can do within the `docker-compose.yml` file. Remember, earlier we discussed that in the YAML file, there needs to be either an `image` section or a `build` section. Let's take a look at an example using each. Then, we will look at an example using as many of the options available for the Docker Compose YAML file as possible.

Here is a breakdown of an example `docker-compose.yml` file. We will break the contents into sections to help you understand each entry.

image

The `image` section tells Docker Compose that you are going to define the configuration of your containers and what settings each will have:

```
haproxy:#container name
  image: tutum/haproxy #image to use from the Docker Hub
  ports: #defining our port setup
    - "80:80" #port to map from Docker Host: to container
  links: #what containers to link to/with
    - varnish1
    - varnish2
varnish1:
  image: jacksoncage/varnish
  ports:
    - "82:80"
  links:
    - web1
    - web2
    - web3
    - web4
  environment: # you use environment to specify variable to pass to the
container with values
    VARNISH_BACKEND_PORT: 80
    VARNISH_BACKEND_IP: web1
    VARNISH_BACKEND_PORT: 80
    VARNISH_BACKEND_IP: web2
```

```
    VARNISH_BACKEND_PORT: 80
    VARNISH_BACKEND_IP: web3
    VARNISH_BACKEND_PORT: 80
    VARNISH_BACKEND_IP: web4
    VARNISH_PORT: 80

varnish2:
  image: jacksoncage/varnish
  ports:
    - "81:80"
  links:
    - web1
    - web2
    - web3
    - web4
  environment:
    VARNISH_BACKEND_PORT: 80
    VARNISH_BACKEND_IP: web1
    VARNISH_BACKEND_PORT: 80
    VARNISH_BACKEND_IP: web2
    VARNISH_BACKEND_PORT: 80
    VARNISH_BACKEND_IP: web3
    VARNISH_BACKEND_PORT: 80
    VARNISH_BACKEND_IP: web4
    VARNISH_PORT: 80

web1:
   image: scottpgallagher/php5-mysql-apache2
   volumes: # you can specify volumes for the container to use. This will
allow for multiple containers to share a volume
     - .:/var/www/html/ # specify the location of the volume
   links:
     - master
     - node1
     - node2
     - nfs1
```

```
    - mcrouter1
    - mcrouter2

web2:
  image: scottpgallagher/php5-mysql-apache2
  volumes:
    - .:/var/www/html/
  links:
    - master
    - node1
    - node2
    - nfs1
    - mcrouter1
    - mcrouter2
web3:
  image: scottpgallagher/php5-mysql-apache2
  volumes:
    - .:/var/www/html/
  links:
    - master
    - node1
    - node2
    - nfs1
    - mcrouter1
    - mcrouter2
web4:
  image: scottpgallagher/php5-mysql-apache2
  volumes:
    - .:/var/www/html/
  links:
    - master
    - node1
    - node2
    - nfs1
    - mcrouter1
    - mcrouter2
```

```
master:
  image:
    scottpgallagher/galeramaster
  hostname: # you can specify a hostname to assign to the container
    master #hostname to use
  environment:
    MARIADB_DATABASE: wordpressmu
    MARIADB_USER: replica
    MARIADB_PASSWORD: replica
node1:
  image:
    scottpgallagher/galeranode
  hostname:
    node1
  environment:
    MARIADB_DATABASE: wordpressmu
    MARIADB_USER: replica
    MARIADB_PASSWORD: replica
  links:
    - master
node2:
  image:
    scottpgallagher/galeranode
  hostname:
    node2
  environment:
    MARIADB_DATABASE: wordpressmu
    MARIADB_USER: replica
    MARIADB_PASSWORD: replica
  links:
    - master
nfs1:
  image: cpuguy83/nfs-server
  volumes:
   - /var/www/wp-content/uploads
mcrouter1:
```

```
  image: jmck/mcrouter-docker
  command: mcrouter --config-str='{"pools":{"A":{"servers":["memcach
ed1:11211", "memcached2:11211"]}},"route":"PoolRoute|A"}' -p 5000 # here
you can specify a command to run on the container when it's started
  links:
   - memcached1
   - memcached2
mcrouter2:
  image: jmck/mcrouter-docker
  command: mcrouter --config-str='{"pools":{"A":{"servers":["memcach
ed1:11211", "memcached2:11211"]}},"route":"PoolRoute|A"}' -p 5000
  links:
   - memcached1
   - memcached2
memcached1:
  image: memcached
  links:
   - db0
memcached1:
  image: memcached
  links:
   - db0
memcached2:
  image: memcached
  links:
   - db0
```

In this very long example, you can see that we are specifying a name for each service as well as the image that is going to be used from the Docker Hub Registry. You can also see a lot of container linking being done in it. Remember that container linking removes the exposition off ports and keeps the communication secure between the said linked containers. We are specifying volumes as well as running some commands in the containers as well.

build

The easiest example of something that uses build is a `wordpress` instance:

```
web:
  build: .
  command: php -S 0.0.0.0:8080 -t /wordpress
  ports:
    - "80:8080"
  links:
    - database
  volumes:
    - .:/wordpress
database:
  image: mysql
  environment:
    MYSQL_DATABASE: wordpress
    MYSQL_ROOT_PASSWORD: password
```

Now, there are other files that are required for this setup; but we are just focusing on the `docker-compose.yml` file right now. In the earlier example, we are specifying two services: a `web` service and a `database` service. In the `database` service, we see that we are using the `image` option; but in the `web` service, we are doing something different. We are building based off the contents of the folder and then placing the files in the `/wordpress` directory inside the container.

The last example

Following is an example just for the sake of it. It's probably something that would not actually run, but you could use it for reference for the different options that you can set within your `docker-compose.yml` file:

```
node2:
  image:
    scottpgallagher/galeranode
  hostname:
    database
  environment:
    MARIADB_DATABASE: wordpressmu
    MARIADB_USER: replica
```

```
    MARIADB_PASSWORD: replica
nfs1:
  image: scottpgallagher/php5-mysql-apache2
  ports:
    - "2049"
  volumes:
    - .:/var/www/html/
web1:
  image: apache
  links:
    - node2
    - nfs1
  volumes_from:
    - nfs1
  expose:
    - "80"
  log_driver: "syslog"
  dns: 8.8.8.8
  restart: always
  hostname: webserver
  read_only: true
```

In the previous example, we specified a lot of things:

- `image`: This specifies what image to use from Docker Hub
- `volumes`: This specifies what paths to use for the volumes that live outside the container
- `volumes-from`: This specifies what volume from another container to mount into the container
- `links`: This links containers together, so the need to expose ports isn't there
- `log_driver`: This selects what logging driver to use
- `dns`: This specifies the ability to add additional DNS servers per container
- `restart`: This states that the container needs to restart when or if it fails
- `hostname`: This sets a hostname for the container
- `read_only`: This allows you to specify that a container is read-only

- `ports`: This specifies what ports can be attached to (from the Docker host to the Docker container)
- `expose`: This specifies what ports are actually exposed externally
- `environment`: This sets the values to the specified variables

Summary

In this chapter, we have looked at how to install Docker Compose on various platforms. We also looked at the file that Docker Compose uses, `YAML` file, for its operation. We took a dive into the Docker Compose usage and commands, and some examples for what you can use Compose.

In the next chapter, we will be looking at Docker Swarm. Docker Swarm is another piece of the Docker ecosystem that can be used to do multiple things; but at its core, it is used for Docker container clustering. It can also use discovery services and advanced scheduling methods. The chapter will also cover the Docker Swarm API, creating a Swarm environment and some Swarm strategies while setting up the environments.

8
Docker Swarm

In this chapter, we will be taking a look at Docker Swarm. With Docker Swarm, you can create and manage Docker clusters. Swarm can be used to disperse containers across multiple hosts. It also has the ability to know how to scale containers as well. In this chapter, we will be covering the following topics:

- Installing Docker Swarm
- The Docker Swarm components
- Docker Swarm usage
- The Docker Swarm commands
- The Docker Swarm topics

Docker Swarm install

Let's get things started by the typical way of installing Docker Swarm. Docker Swarm is only available for Linux and Mac OS X. The installation process for both is the same. Let's take a look at how we install Docker Swarm.

Installation

Ensure that you already have Docker installed, either through the `curl` command on Linux or through Docker Toolbox on Mac OS X. Once you have the Docker daemon installed, installing Docker Swarm will be simple:

```
$ docker pull swarm
```

One command and you are up and running. That's it!

Docker Swarm components

What components are involved with Docker Swarm? Let's take a look at the three components of Docker Swarm:

- Swarm
- Swarm manager
- Swarm host

Swarm

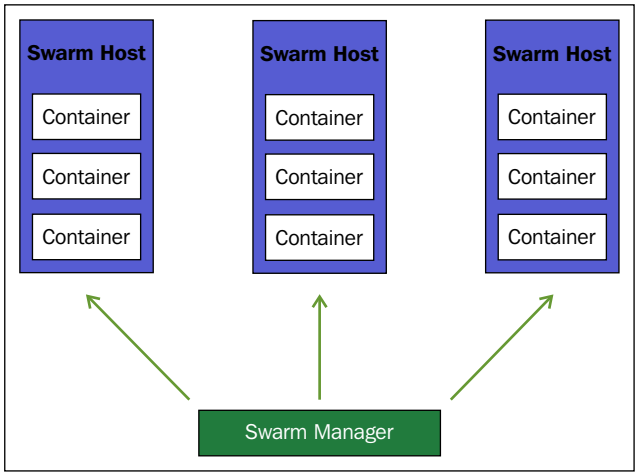
Docker Swarm is the container that runs on each Swarm host. Swarm uses a unique token for each cluster to be able to join the cluster. The Swarm container itself is the one that communicates on behalf of that Docker host to the other Docker hosts that are running Docker Swarm as well as the Docker Swarm manager.

Swarm manager

The Swarm manager is the host that is the central management point for all the Swarm hosts. The Swarm manager is where you issue all your commands to control nodes. You can switch between the nodes, join nodes, remove nodes, and manipulate the hosts.

Swarm host

Swarm hosts, which we saw earlier as the Docker hosts, are those that run the Docker containers. The Swarm host is managed from the Swarm manager.



The preceding figure is an illustration of all the Docker Swarm components. We see that the Docker Swarm manager talks to each Swarm host that is running the Swarm container.

Docker Swarm usage

Let's now take look at Swarm usage and how we can do the following tasks:

- Creating a cluster
- Joining nodes
- Removing nodes
- Managing nodes

Creating a cluster

Let's start by creating the cluster, which starts with a Swarm manager. We first need a token that can be used to join all the nodes to the cluster:

```
$ docker run --rm swarm create
85b335f95e9a37b679e2ea9e6ad8d6361
```

We can now use that token to create our Swarm manager:

```
$ docker-machine create \
        -d virtualbox \
        --swarm \
        --swarm-master \
        --swarm-discovery token://85b335f95e9a37b679e2ea9e6ad8d6361 \
        swarm-master
Creating VirtualBox VM...
Creating SSH key...
Starting VirtualBox VM...
Starting VM...
```

To see how to connect Docker to this machine, run `docker-machine env swarm-master`.

The `swarm-master` node is now in VirtualBox. We can see this machine by doing as follows:

```
$ docker-machine ls

NAME           ACTIVE   DRIVER       STATE     URL
SWARM
swarm-master            virtualbox   Running   tcp://192.168.99.101:2376
swarm-master (master)
```

Now, let's point Docker Machine at the new Swarm master. The earlier output we saw when we created the Swarm master tells us how to point to the node:

```
$ docker-machine env swarm-master

export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.102:2376"
export DOCKER_CERT_PATH="/Users/spg14/.docker/machine/machines/swarm-
master"
export DOCKER_MACHINE_NAME="swarm-master"
# Run this command to configure your shell:
# eval "$(docker-machine env swarm-master)"
```

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

```
$ eval "$(docker-machine env swarm-master)"
```

Now, if we look at what machines are on our host, we can see that we have the `swarm-master` host as well. It is set to `ACTIVE`, which means that we can now run commands against this host:

```
$ docker-machine ls

NAME           ACTIVE   DRIVER       STATE     URL
SWARM
swarm-master   *         virtualbox   Running   tcp://192.168.99.101:2376
swarm-master (master
```

Joining nodes

Again using the token, which we got from the earlier commands, used to create the Swarm manager, we need that same token to join nodes to that cluster:

```
$ docker-machine create \
-d virtualbox \
--swarm \
--swarm-discovery token://85b335f95e9a37b679e2ea9e6ad8d6361 \
swarm-node1
```

Now, if we look at the machines on our system, we can see that they are both part of the same Swarm:

```
$ docker-machine ls

NAME           ACTIVE   DRIVER       STATE     URL
SWARM
swarm-master    *       virtualbox   Running   tcp://192.168.99.102:2376
swarm-master(master)
swarm-node1             virtualbox   Running   tcp://192.168.99.103:2376
swarm-master
```

Listing nodes

First, ensure you are pointing at the Swarm master:

```
$ docker-machine ls

NAME           ACTIVE   DRIVER       STATE     URL
SWARM

swarm-master   *        virtualbox   Running   tcp://192.168.99.102:2376
swarm-master(master)
swarm-node1             virtualbox   Running   tcp://192.168.99.103:2376
swarm-master
```

Now, we can see what machines are joined to this cluster based off the token used to join them all together:

```
$ docker run --rm swarm list token://85b335f95e9a37b679e2ea9e6ad8d6361

192.168.99.102:2376
192.168.99.103:2376
```

Managing a cluster

Let's see how we can do some management of all of the cluster nodes we are creating.

So, there are two ways you can go about managing these Swarm hosts and the containers on each host that you are creating. But first, you need to know some information about them, so we will turn to our Docker Machine command again:

```
$ docker-machine ls

NAME            ACTIVE   DRIVER       STATE     URL
SWARM
swarm-master     *       virtualbox   Running   tcp://192.168.99.102:2376
swarm-master(master)
swarm-node1              virtualbox   Running   tcp://192.168.99.103:2376
swarm-master
```

You can switch to each Swarm host like we have seen earlier by doing something similar to the following—changing the values—and by following the instructions from the output of the command:

```
$ docker-machine env <Node_Name>
```

But this is a lot of tedious work. There is another way we can manage these hosts and see what is going on inside them. Let's take a look at how we can do it. From the previous `docker-machine ls` command, we see that we are currently pointing at the `swarm-master` node. So, any Docker commands we issue would go against this host.

But, if we run the following, we can get information on the `swarm-node1` node:

```
$ docker -H tcp://192.168.99.103:2376 info

Containers: 1
Images: 8
Storage Driver: aufs
```

```
 Root Dir: /mnt/sda1/var/lib/docker/aufs
 Backing Filesystem: tmpfs
 Dirs: 10
 Dirperm1 Supported: true
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 4.0.9-boot2docker
Operating System: Boot2Docker 1.8.2 (TCL 6.4); master : aba6192 - Thu Sep
10 20:58:17 UTC 2015
CPUs: 1
Total Memory: 996.2 MiB
Name: swarm-node1
ID: SDEC:4RXZ:O3VL:PEPC:FYWM:IGIK:CFM5:UXPS:U4S5:PNQD:5ULK:TSCE
Debug mode (server): true
File Descriptors: 18
Goroutines: 29
System Time: 2015-09-16T09:32:27.67035212Z
EventsListeners: 1
Init SHA1:
Init Path: /usr/local/bin/docker
Docker Root Dir: /mnt/sda1/var/lib/docker
Labels:
 provider=virtualbox
```

So, we can see the information on this host such as the number of containers, the numbers of images on the host, as well as information about the CPU, memory, and so on.

We can see from the earlier information that one container is running. Let's take a look at what is running on the `swarm-node1` host:

```
$ docker -H tcp://192.168.99.103:2376 ps

CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS               NAMES
12a400424c87        swarm:latest        "/swarm join --advert"   17 hours
ago         Up 17 hours         2375/tcp            swarm-agent
```

Now, you can use any of the Docker commands using this method against any Swarm host that is listed in the output of your `docker-machine ls` output.

The Docker Swarm commands

Now, let's take a look at some Docker Swarm-specific commands that we can use. Let's revert to the ever-so-helpful—the `help` switch on the Docker Swarm command:

```
$ docker run --rm swarm --help

Usage: swarm [OPTIONS] COMMAND [arg...]

A Docker-native clustering system

Version: 0.4.0 (d647d82)

Options:
  --debug      debug mode [$DEBUG]
  --log-level, -l "info"  Log level (options: debug, info, warn, error,
fatal, panic)
  --help, -h      show help
  --version, -v      print the version

Commands:
  create, c  Create a cluster
  list, l    List nodes in a cluster
  manage, m  Manage a docker cluster
  join, j  join a docker cluster
  help, h  Shows a list of commands or help for one command
Using TLS
```

Let's take a look at the options you can use for Docker Swarm as well as the commands that are associated with it.

Options

Looking over the options from the preceding output, we can see the `--debug` and `--log` level switches. The other two are straightforward, as one will just print out the help information and the other one will print out the version number that we can see in the previous output. The options are used after each of the following subcommands of Docker Swarm.

For example:

```
$ docker run --rm swarm list --debug
$ docker run --rm swarm manage --debug
$ docker run --rm swarm create --debug
```

list

We looked at the Swarm `list` command before:

```
$ docker run --rm swarm list token://85b335f95e9a37b679e2ea9e6ad8d6361

192.168.99.102:2376
192.168.99.103:2376
```

But there is also a switch that we can tack onto the `list` command and that is the `--timeout` switch:

```
$ docker run --rm swarm list --timeout 20s token://85b335f95e9a37b679e2ea
9e6ad8d6361
```

This will allow more time to find the nodes that are a part of Swarm. It could take time for the hosts to check, depending upon things such as network latency or if they are running in different parts of the globe.

create

We have seen how we can create a Swarm cluster as well. What this command actually does is it gives us the token that we need to create the cluster and join all the nodes to it. There are no other switches that can be used with this command as we have seen with other commands:

```
$ docker run --rm swarm create

85b335f95e9a37b679e2ea9e6ad8d6361
```

manage

We can manage a cluster with the `manage` subcommand in Docker Swarm. An example of this command would look like the following, replacing the information to align with your IP address and Swarm token:

```
$ docker run --rm swarm manage -H tcp://192.168.99.104:2376 token://85b33
5f95e9a37b679e2ea9e6ad8d6361
```

The Docker Swarm topics

There are three advanced topics we will take a look at in this section:

- Discovery services
- Advanced scheduling
- The Docker Swarm API

Discovery services

You can also use services such as etcd, ZooKeeper, consul, and many others to automatically add nodes to your Swarm cluster as well as do other things such as list the nodes or manage them. Let's take a look at consul and how you can use it. This will be the same for each discovery service that you might use. It just involves switching out the word consul with the discovery service you are using.

On each node, you will need to do something different in how you join the machines. Earlier, we did something like this:

```
$ docker-machine create \
-d virtualbox \
--swarm \
--swarm-discovery token://85b335f95e9a37b679e2ea9e6ad8d6361 \
swarm-node1
```

Now, we would do something similar to the following (based upon the discovery service you are using):

```
$ docker-machine create \
-d virtualbox \
--swarm \
join --advertise=<swarm-node1_ip:2376> \
consul://<consul_ip> \
swarm-node1
```

You can now start `manage` on your laptop or the system that you will be using as the Swarm manager. Before, we would run something like this:

```
$ docker run --rm swarm manage -H tcp://192.168.99.104:2376 token://85b33
5f95e9a37b679e2ea9e6ad8d6361
```

Now, we run this with regards to discovery services:

```
$ docker run --rm swarm manage -H tcp://192.168.99.104:2376
consul://<consul_ip>
```

We can also list the nodes in this cluster as well as the discovery service:

```
$ docker run --rm swarm list -H tcp://192.168.99.104:2376
consul://<consul_ip>
```

You can easily switch out consul for another discovery service such as etcd or ZooKeeper; the format will still be the same:

```
$ docker-machine create \
-d virtualbox \
--swarm \
join --advertise=<swarm-node1_ip:2376> \
etcd://<etcd_ip> \
swarm-node1


$ docker-machine create \
-d virtualbox \
--swarm \
join --advertise=<swarm-node1_ip:2376> \
zk://<zookeeper_ip> \
swarm-node1
```

Advanced scheduling

What is advanced scheduling with regards to Docker Swarm? Docker Swarm allows you to rank nodes within your cluster. It provides three different strategies to do this. These can be used by specifying them with the `--strategy` switch with the `swarm manage` command:

- `spread`
- `binpack`
- `random`

`spread` and `binpack` use the same strategy to rank your nodes. They are ranked based off of the node's available RAM and CPU as well as the number of containers that it has running on it.

`spread` will rank the host with less containers higher than a container with more containers (assuming that the memory and CPU values are the same). `spread` does what the name implies; it will spread the nodes across multiple hosts. By default, `spread` is used with regards to scheduling.

`binpack` will try to pack as many containers on as few hosts as possible to keep the number of Swarm hosts to a minimal.

`random` will do just that—it will randomly pick a Swarm host to place a node on.

The Swarm scheduler comes with a few filters that can be used as well. These can be assigned with the `--filter` switch with the `swarm manage` command. These filters can be used to assign nodes to hosts. There are five filters that are associated with it:

- `constraint`: There are three types of constraints that can be assigned to nodes:
    - `storage=`: This is used if you want to specify a node that is put on a host and has SSD drives in it
    - `region=`: This is used if you want to set a region; mostly used for cloud computing such as AWS or Microsoft Azure
    - `environment=`: This can set a node to be put into production, development, or other created environments
- `affinity`: This filter is used to create attractions between containers. This means that you can specify a filter name and then have all those containers run on the same node.
- `port`: The `port` filter finds a host that has the open port needed for the node to run; it then assigns the node to that host. So, if you have a MySQL instance and need port `3306` open, it will find a host that has port `3306` open and assign the node to that host for operation.
- `dependency`: The `dependency` filter schedules nodes to run on the same host based off of three dependencies:
    - `--volumes-from=dependency`
    - `--link=dependency:<alias>`
    - `--net=container:dependency`
- `health`: The `health` filter is pretty straightforward. It will prevent the scheduling of nodes to run on unhealthy hosts.

The Swarm API

Before we dive into the Swarm API, let's first make sure you understand what an API is. An API is defined as an application programming interface. An API consists of routines, protocols, and tools to build applications. Think of an API as the bricks used to build a wall. This allows you to put the wall together using those bricks. What APIs allow you to do is code in the environment you are comfortable in and reach into other environments to do the work you need. So, if you are used to coding in Python, you can still use Python to do all your work while using the Swarm API to do the work in Swarm that you would like done.

For example, if you wanted to create a container, you would use the following in your code:

```
POST /containers/create HTTP/1.1
Content-Type: application/json

{
       "Hostname": "",
       "Domainname": "",
       "User": "",
       "AttachStdin": false,
       "AttachStdout": true,
       "AttachStderr": true,
       "Tty": false,
       "OpenStdin": false,
       "StdinOnce": false,
       "Env": null,
       "Cmd": [
               "date"
       ],
       "Entrypoint": "",
       "Image": "ubuntu",
       "Labels": {
               "com.example.vendor": "Acme",
               "com.example.license": "GPL",
               "com.example.version": "1.0"
       },
       "Mounts": [
         {
           "Source": "/data",
           "Destination": "/data",
           "Mode": "ro,Z",
           "RW": false
         }
```

```
        ],
        "WorkingDir": "",
        "NetworkDisabled": false,
        "MacAddress": "12:34:56:78:9a:bc",
        "ExposedPorts": {
                "22/tcp": {}
        },
        "HostConfig": {
          "Binds": ["/tmp:/tmp"],
          "Links": ["redis3:redis"],
          "LxcConf": {"lxc.utsname":"docker"},
          "Memory": 0,
          "MemorySwap": 0,
          "CpuShares": 512,
          "CpuPeriod": 100000,
          "CpusetCpus": "0,1",
          "CpusetMems": "0,1",
          "BlkioWeight": 300,
          "MemorySwappiness": 60,
          "OomKillDisable": false,
          "PortBindings": { "22/tcp": [{ "HostPort": "11022" }] },
          "PublishAllPorts": false,
          "Privileged": false,
          "ReadonlyRootfs": false,
          "Dns": ["8.8.8.8"],
          "DnsSearch": [""],
          "ExtraHosts": null,
          "VolumesFrom": ["parent", "other:ro"],
          "CapAdd": ["NET_ADMIN"],
          "CapDrop": ["MKNOD"],
          "RestartPolicy": { "Name": "", "MaximumRetryCount": 0 },
          "NetworkMode": "bridge",
          "Devices": [],
          "Ulimits": [{}],
          "LogConfig": { "Type": "json-file", "Config": {} },
          "SecurityOpt": [""],
          "CgroupParent": ""
      }
  }
```

You would use the preceding example to create a container; but there are also other things you can do such as inspect containers, get the logs from a container, attach to a container, and much more. Simply put, if you can do it through the command line, there is more than likely something in the API that can be used to tie into to do it through the programming language you are using.

The Docker documentation states that the Swarm API is mostly compatible with the Docker Remote API. Now we could list them out in this section. But seeing that the list could change as things could be added into the Docker Swarm API or removed, I believe, it's best to refer to the link to the Swarm API documentation here instead of listing them out, so the information is not outdated:

`https://docs.docker.com/swarm/api/swarm-api/`

The Swarm cluster example

We will now go through an example of how to set up a Docker Swarm cluster:

```
# Create a new Docker host with Docker Machine
$ docker-machine create --driver virtualbox swarm

# Point to the new Docker host
$ eval "$(docker-machine env swarm)"

# Generate a Docker Swarm Discovery Token
$ docker run swarm create

# Launch the Swarm Manager
$ docker-machine create \
        --driver virtualbox \
        --swarm \
        --swarm-master \
        --swarm-discovery token://<DISCOVERY_TOKEN> \
        swarm-master

# Launch a Swarm node
$ docker-machine create \
    --driver virtualbox \
    --swarm \
```

```
    --swarm-discovery token://<DISCOVERY_TOKEN> \
    swarm_node-01

# Launch another Swarm node
$ docker-machine create \
    --driver virtualbox \
    --swarm \
    --swarm-discovery token://<DISCOVERY_TOKEN> \
    swarm_node-02

# Point to our Swarm Manager
$ eval "$(docker-machine env swarm-master)"

# Execute 'docker info' command to view information about your
environment
$ docker info

# Execute 'docker ps -a'; will show you all the containers running as
well as how they are joined to the same Swarm cluster
$ docker ps -a

# Run simple test
$ docker run hello-world

# You can then execute the 'docker ps -a' command again to see what node
it ran on
$ docker ps -a
# You will want to look at the column labeled 'NAMES'. If you continue
to re-run the 'docker run hello-world' command/container you will see it
will run on a different Swarm node
```

Summary

In this chapter, we took a dive into Docker Swarm. We took a look at how to install Docker Swarm and the Docker Swarm components; these are what make up Docker Swarm. We took a look at how to use Docker Swarm; joining, listing, and managing Swarm nodes. We reviewed the Swarm commands and how to use them. We also covered some advanced Docker Swarm topics such as advanced scheduling for your jobs, discovery services to discover new containers to add to Docker Swarm, and the Docker Swarm API that you can use to tie your own code to perform the Swarm commands.

In the next chapter, we will take a look at running Docker in production. We will take everything you have learned in all of the previous chapters and put them into production. We will look at how to monitor your containers and the safeguards you can put into place to help with container recovery. We will also look at how you can extend into external platforms such as Heroku.

9
Docker in Production

In this chapter, we will be looking at Docker in production, pulling all the pieces together so you can start using Docker in your production environments and feel comfortable doing so. Let's take a peek at what we will be covering in this chapter:

- Setting up hosts and nodes
- Managing hosts and containers
- Using Docker Compose
- Extending to external platforms
- Security

Where to start?

When we start thinking about putting Docker into our production environment, we first need to know where to start. This sometimes can be the hardest part of any project. We first need to start by setting up our Docker hosts and then start running containers on them. So, let's start here!

Setting up hosts

Remember, as it was mentioned in the earlier chapter, that setting up hosts will require us to tap into our Docker Machine knowledge. We can deploy these hosts to different environments, including cloud hosting. To take a walk down memory lane, let's look at how we go about doing this:

```
$ docker-machine create --driver <driver_name> <host_name>
```

Now, there are two values that we can manipulate: `<driver_name>` and `<host_name>`. The host name can be whatever you want it to be. But I recommend that it should be something that would help you understand its purpose. The driver name on the other hand has to be the location where you want to create the host. If you are looking at doing something locally, then you can use VirtualBox or VMware Fusion. If you are looking at deploying your application to a cloud service, you can use something like Amazon EC2, Azure, or DigitalOcean. Most of these cloud services will require additional details to authenticate who you are and where to place the host:

For example, for AWS, you would use:

```
$ docker-machine create --driver amazonec2 --amazonec2-access-key <AWS_
ACCESS_KEY> --amazonec2-secret-key <AWS_SECRET_KEY> --amazonec2-subnet-id
east-1b amazonhost
```

You can see that you will need the following:

- Amazon access key
- Amazon secret key
- Amazon subnet ID

Setting up nodes

Next, we want to set up the nodes or containers to run on the hosts that we have recently created. Again, using a combination of Docker Machine with the Docker daemon, we can do this. We first must use Docker Machine to point to the Docker host that we want to deploy some containers on:

```
$ docker-machine env <host_name>
$ eval "$(docker-machine env <host_name>)"
```

Now we can run our normal Docker commands against this Docker host. To do this, we will simply use the Docker command-line tools. To deploy the containers, we can pull the following images:

```
$ docker pull <image_name>
```

Or, we can run a container on a host:

```
$ docker run -d -p 80:80 nginx
```

Host management

In this section, we will focus on host management, that is, the ways we can manage our hosts, what we should use to manage our hosts, how we can monitor our hosts, and container failover, which is very important when something happens to the host that is running critical containers.

Host monitoring

With host monitoring you can do so via the command line using Docker Machine as also there are some GUI applications out there that can be useful as well. For Machine, you can use the `ls` subcommand:

```
$ docker-machine ls
NAME            ACTIVE   DRIVER       STATE     URL
SWARM
amazonhost               amazonec2    Error
swarm-master    *        virtualbox   Running   tcp://192.168.99.102:2376
swarm-master(master)
swarm-node1              virtualbox   Running   tcp://192.168.99.103:2376
swarm-master
```

You can use some GUI applications out there as well, such as:

- **Shipyard**: `https://shipyard-project.com/`
- **DockerUI**: `https://github.com/crosbymichael/dockerui`
- **Panamax**: `http://panamax.io/`

Docker Swarm

Another tool that you can use for node management is that of Docker Swarm. We saw previously how helpful Swarm can be. Remember that you can use Docker Swarm to manage your hosts as well as to create and list them. The most useful command to remember for Swarm is the `list` subcommand. With the `list` subcommand, you can get a view of all the nodes and their statuses:

Remember that you will need either the discovery service IP or the token number that is used for Swarm:

```
$ docker run swarm list token://<swarm_token>
```

Swarm manager failover

With Docker Swarm, you can set up your manager node to be highly available. That is, if the managing host dies, you can have it failover to another host. If you don't have it set up, there will be a service interruption, as you won't be able to communicate to your hosts anymore and will need to reset them up to point to the new Docker Swarm manager. You can set up as many replicas as you want.

To set this up, you will need to use the `--replication` and `--advertise` flags. This tells Swarm that there will be other managers for failover. It will also tell Swarm what address to advertise on, so the other managers know on what IP address to connect for other Swarm managers.

Container management

Now, let's look at container management. This includes questions such as where to store the images that we will be creating, how to use these images, and what commands and GUI applications we can use. It also covers how we can easily monitor our running containers, automatically restart containers upon a failure, and how to roll the updates of our containers.

Container image storage

In *Chapter 3*, *Container Image Storage*, we looked at the various locations to store the images you are creating. Remember that there are three major locations to store them:

- **Docker Hub**: A location that is run by Docker and can contain public and private repositories
- **Docker Trusted Registry**: A location that is again run by Docker, but provides the ability to get support from Docker
- **The locally run Docker registry**: Locally run by yourself to storage images

You will want to consider where you want your images to be stored. If you are running containers that might contain data that you do not want anybody to be able to access, such as private code, you may want to run your own Docker registry to keep the data locked. If you are testing, then you may only want to use Docker Hub. If you are in an enterprise environment where uptime is necessary, then the second option of having Docker there for support would be immensely beneficial. Again, it all depends on your setup and needs. The best thing is that no matter what you choose at first, you can easily change and push your images to these locations without having to jump through a lot of extra hoops or other configurations.

Image usage

The most important thing to remember about Docker images is the four Ws:

- **Who**: Who made the image?
- **What**: What is contained in the image?
- **Why**: Why are these things created?
- **Where**: Where are the items such as the Dockerfile or the other code for the image?

The Docker commands and GUIs

Remember that there are many commands that you can use to control your containers. With tools such as the Docker daemon, Docker Machine, Docker Compose, and Docker Swarm, there is almost nothing that can stop you from achieving the goal you want. Remember, however, that some of these tools are not available on all the platforms yet. I stress yet as I assume that Docker will eventually have their tools available for all the environments. Be sure to use the `--help` flag on all the commands to see the additional switches that might be available. I myself am always finding new switches to use every day on various commands.

There are also many GUI applications out there; they can be beneficial to your container's management needs. One that has been at the forefront of this since the beginning is Panamax. Panamax provides the ability to set up your environments in a GUI-based application for you to deploy, monitor, and manipulate your container environments. With the popularity of Docker growing each day, there will be many, many, many others that you can use to help set up and tune your environment.

Container monitoring

We can also monitor our containers using methods similar to monitoring hosts: using Docker commands as well as GUIs that are built by others.

First, the Docker commands that you can use:

- `docker stats`
- `docker port`
- `docker logs`
- `docker inspect`
- `docker events`

In the *Host monitoring* section, you can see that the same GUI applications can monitor both your Docker hosts and your containers. It is a double bonus as you don't need separate applications to monitor each service.

Automatic restarts

Another great thing you can do with your Docker images is you can set them to automatically restart upon a failure or a reboot of a Docker host. There is a flag that can be set at runtime: the `--restart` flag. There are three options you can set, one of which is set by default by not setting the flag.

These three options are:

- `no`: The default by not using the flag.
- `on-failure:max_retires`: Sets the container to restart, but not indefinitely if there is a major problem. It will try to restart the container a number of times based on the value set for `max_retires`. After it has passed that value, it will not try to restart anymore.
- `always`: Will always restart the container. It could cause a looping issue if the container continues to just restart.

Rolling updates

One of the benefits I have learned to love about Docker is the ability to control it the same way I control the code that I write. Just like Git, remember that your Docker images are version-controlled as well. This being said, you can do things such as rolling updates to them. There are two ways you can go about doing it. You can keep your images as a hosted code on something like GitHub. You can then update your code, build your image, and deploy your containers. If something goes wrong, you can simply use another version of that image to redeploy. There is also another way you can do this. You can get the new image up and running; when you are ready, stop the old container from running and then start up the new one. If you use items such as discovery services, it becomes even easier; you can roll your newly updated images into the discovery service while rolling out the old images. This makes for seamless upgrades and a great peace of mind for zero downtime.

Docker Compose usage

One of the more useful tools, and one I find myself using a lot, is Docker Compose. Compose has a lot of powerful usage, which in turn is great for you. In this section, we will look at two of its usages:

- Developer environments
- Scaling environments

Developer environments

You can use Docker Compose to set up your developer environments. How is this any different from setting up a virtual machine for them to use or letting them use their own setup? With Docker Compose, you control the setup, you control what is linked to what, and you know how the environment is set up. So, there is no more "well it works on my system" or need to troubleshoot error messages that are appearing on one system setup but not another.

Scaling environments

Docker Compose also allows you to scale containers that are located in the `docker-compose.yml` file. For example, let's say our Compose file looks as follows:

```
varnish:
  image: jacksoncage/varnish
  ports:
    - "82:80"
  links:
    - web
environment:
    VARNISH_BACKEND_PORT: 80
    VARNISH_BACKEND_IP: web
    VARNISH_PORT: 80
web:
   image: scottpgallagher/php5-mysql-apache2
   volumes:
     - .:/var/www/html/
```

With the Compose setup, you can easily scale the containers from your `docker-compose.yml` file. For instance, if you need more web containers to help with the backend load, you can do so with Docker Compose. Be sure that you are in the folder where your `docker-compose.yml` file is located:

```
$ docker-compose scale web=3
```

This will add three extra web containers and do all the linking as well as the traffic forwarding from the `varnish` server that is necessary. This can be immensely helpful if you are looking at figuring out how many instances you might need to help scale for load or service usage.

Extending to external platform(s)

We looked at how we can extend to some other external platforms such as cloud services like AWS, Microsoft Azure, and DigitalOcean before. In this section, we will focus on extending Docker to the Heroku platform. Heroku is more a little different than those cloud services; it is considered a **Platform as a Service** (**PaaS**). Instead of deploying containers to it, you can link your containers to the Heroku platform from which it is running a service, such as PHP, Java, Node.js, Python, or many others. So, you can run your rails application on Heroku and then attach your Docker container to that platform.

Heroku

The way you can use Docker and Heroku together is by creating your application on the Heroku platform. Then, in your code, you will have something similar to the following:

```
{
  "name": "Application Name",
  "description": "Application to run code in a Docker container",
  "image": "<docker_image>:<tag>",
  "addons": [ "heroku-postgresql" ]
}
```

To take a step back, we first need to install a plugin to be able to get this functionality working. To install it, we will simply run:

```
$ heroku plugins:install heroku-docker
```

Now, if you are wondering what image you can or should be using from Docker Hub, Heroku maintains a lot of images you can use in the preceding code. They are as follows:

```
heroku/nodejs
heroku/ruby
heroku/jruby
heroku/python
heroku/scala
heroku/clojure
heroku/gradle
heroku/java
heroku/go
heroku/go-gb
```

Overall security

Lastly, let's take a look at the security aspect of putting Docker into production. This is probably one of the most talked about aspects of not only Docker, but any technology out there. What security risks exist? What security advantages exist? We will take a look at both of these aspects as well as cover the best practices for your overall Docker setup.

Security best practices

These are the things to keep in mind when you are setting up your production environment:

- Whoever has access to your Docker host has access to every single Docker container that is running on that host and has the ability to stop them, delete them, or even start up new containers.
- Remember that you can run Docker containers or attach containers to Docker volumes using the read-only modes. This can be done by adding the `:ro` option to the volume:

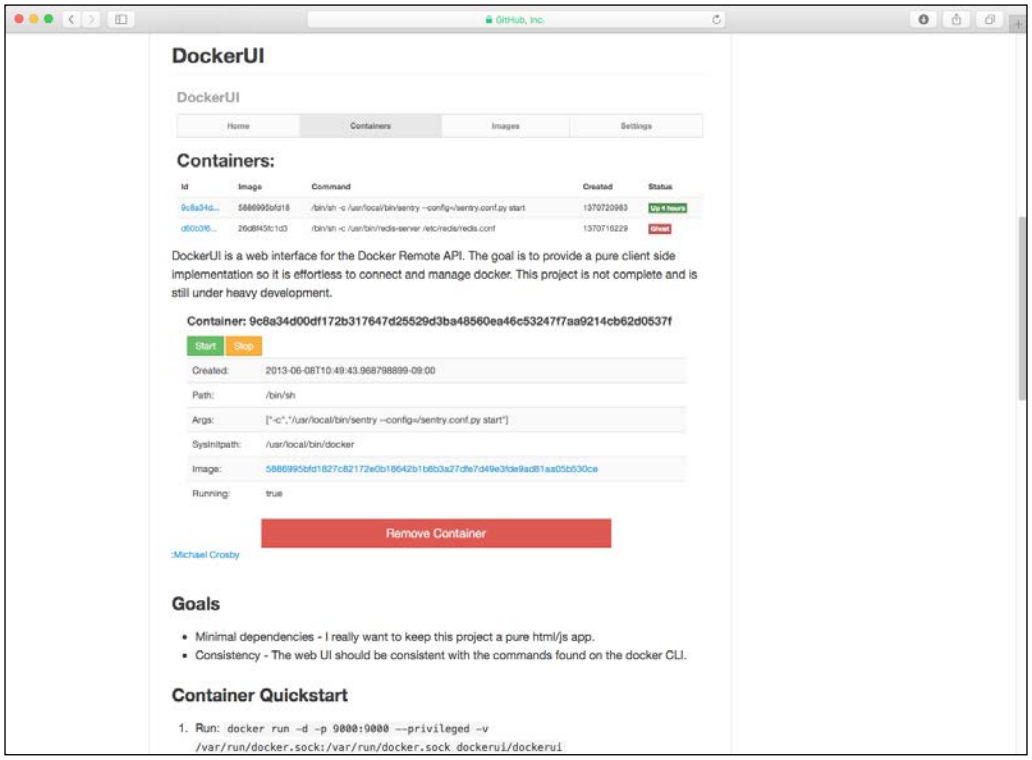
  ```
  $ docker run -d -v /opt/uploads:ro nginx
  $ docker run -d --volumes-from data:ro nginx
  ```

- Remember to utilize the Docker security benchmark application to help tune your environments (see *Chapter 5, Docker Security*, for more information).
- Utilize the Docker command-line tools to your capability to see what has changed in a particular image:

```
$ docker diff
$ docker inspect
$ docker history
```

DockerUI

DockerUI is a tool written by Michael Crosby, who at the time of writing this book worked for Docker. DockerUI is a simple way to view what is going on inside your Docker host.

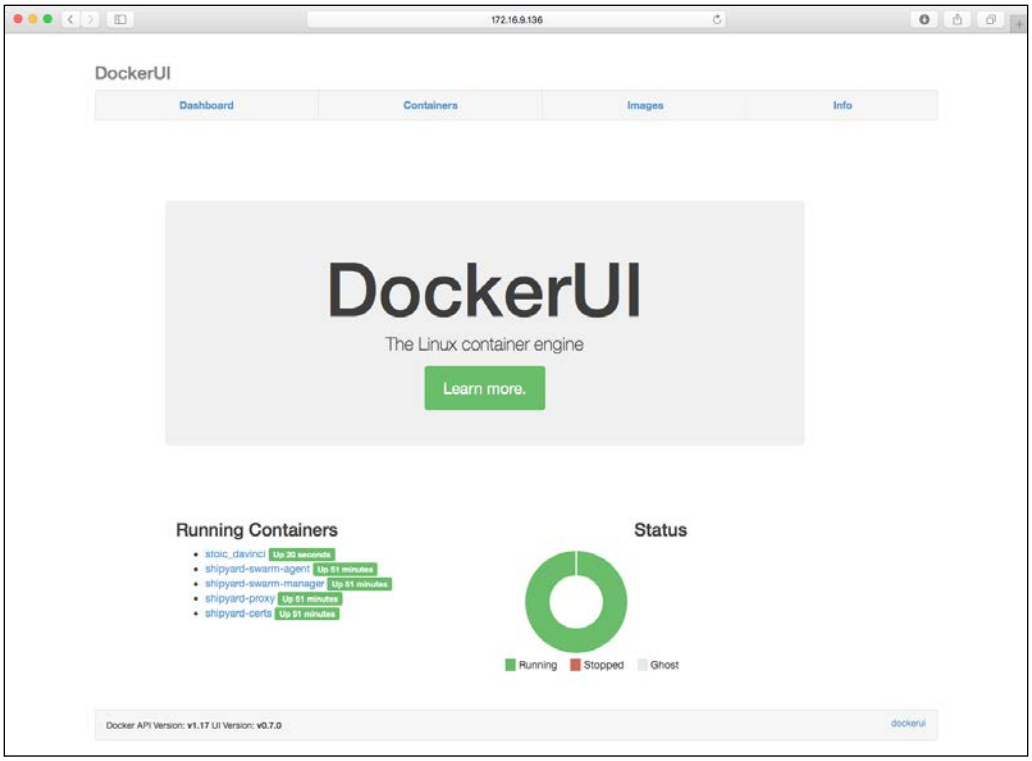


This is a screenshot of the GitHub repository, where the code for DockerUI is kept. You can view the content yourself by visiting `https://github.com/crosbymichael/dockerui`.

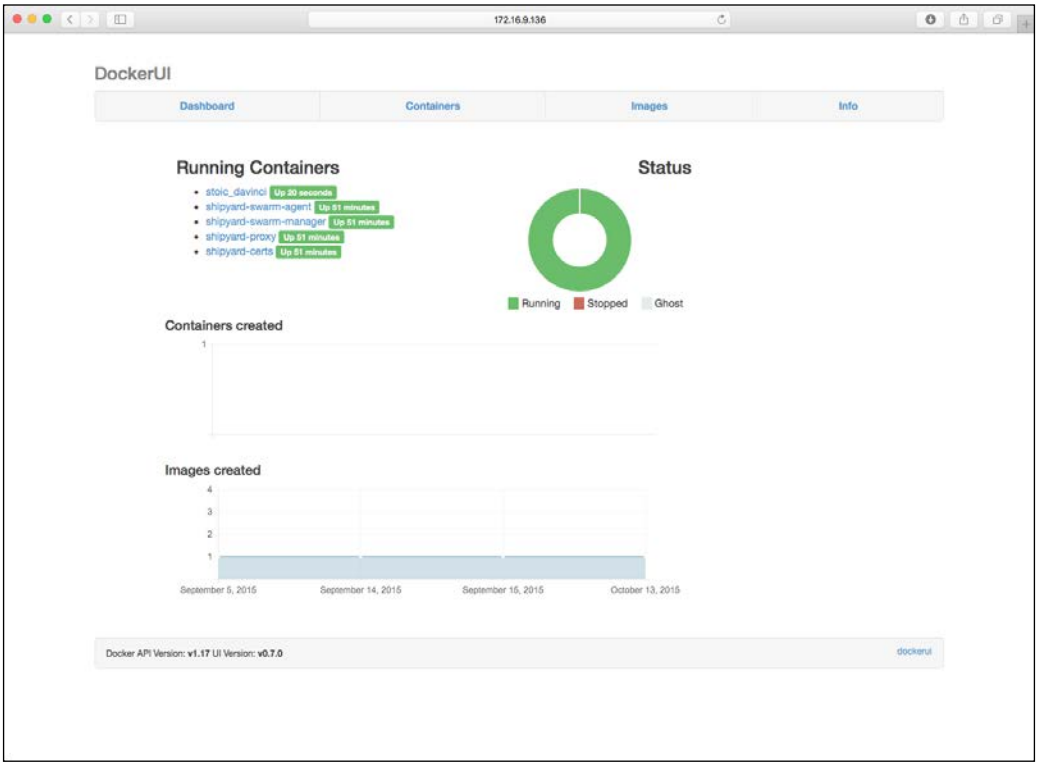
This page will include screenshots of DockerUI in action as well as the current features of DockerUI that are available. You can create pull requests against the code if you have ideas you would like to see in DockerUI and would like to help contribute to the code. You can also submit issues that you might find with DockerUI.

The installation of DockerUI is very straightforward with you just running a simple Docker run command to get started:

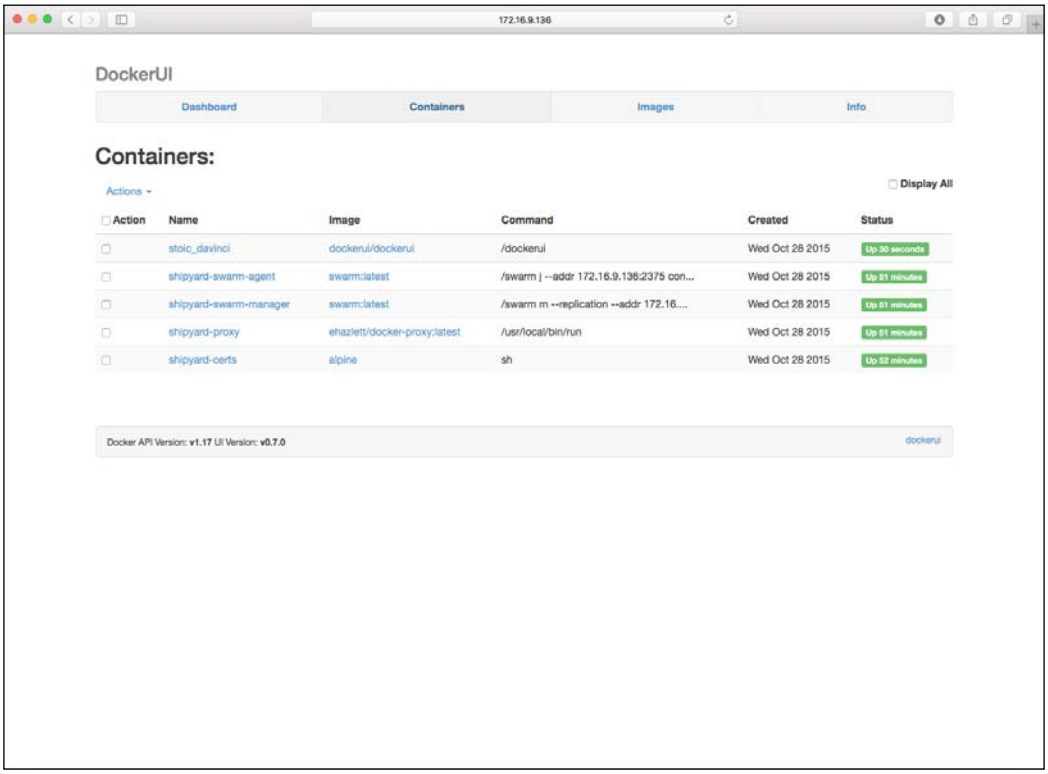
```
$ docker run -d -p 9000:9000 --privileged -v /var/run/docker.sock:/var/
run/docker.sock dockerui/dockerui
```



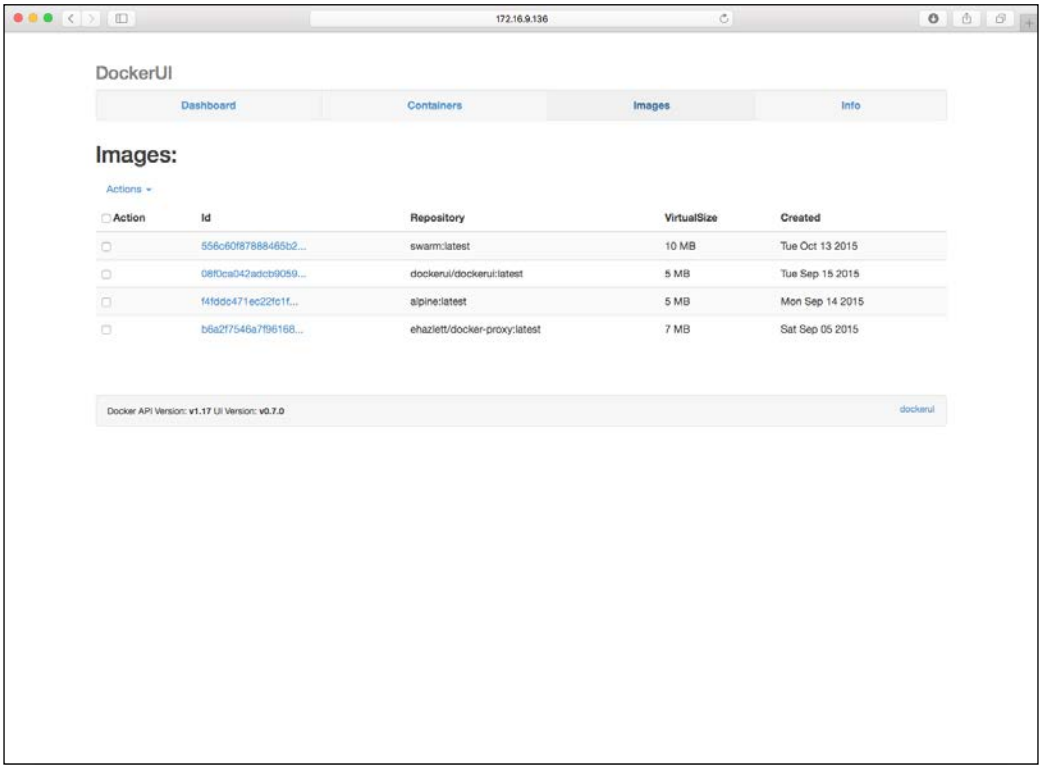
After you have run the previous command, you will be able to navigate to the DockerUI web interface. You should be able to easily break down the run command and see what it is doing and where you need to go to get to the dashboard. However, in case you are stumped, here is what the command is doing: it is running the DockerUI container on your Docker host and exposing port `9000` from the host to the container. So, simply launching a web browser and pointing to the IP address of the Docker host and then port `9000` will give you to a screen similar to the previous one. This is the DockerUI web dashboard.



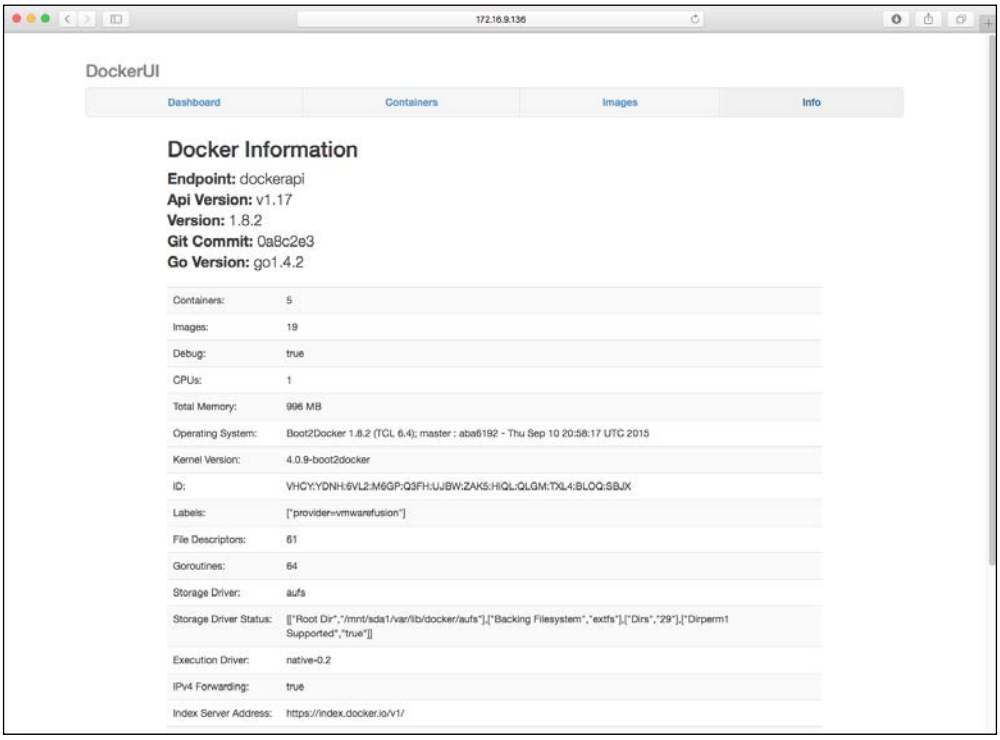
This is another view of the dashboard shortly after you have launched the container and visited the web interface. You can see information such as what containers are currently running on your Docker host and what their statuses are; some could be stopped as well. It will also show you the containers that are created and a timeline for when the images were created.



At the top of the web interface, you will see a navigation bar. When you click on the **Containers** item, you will be brought to a page that provides you information on all the containers running on your host. You will see their name, the images used to run the containers, what command is being executed inside each container, when they were created, and their statuses. You can take actions against these containers from here as well. These actions are start, stop, restart, kill, pause, unpause, and remove.



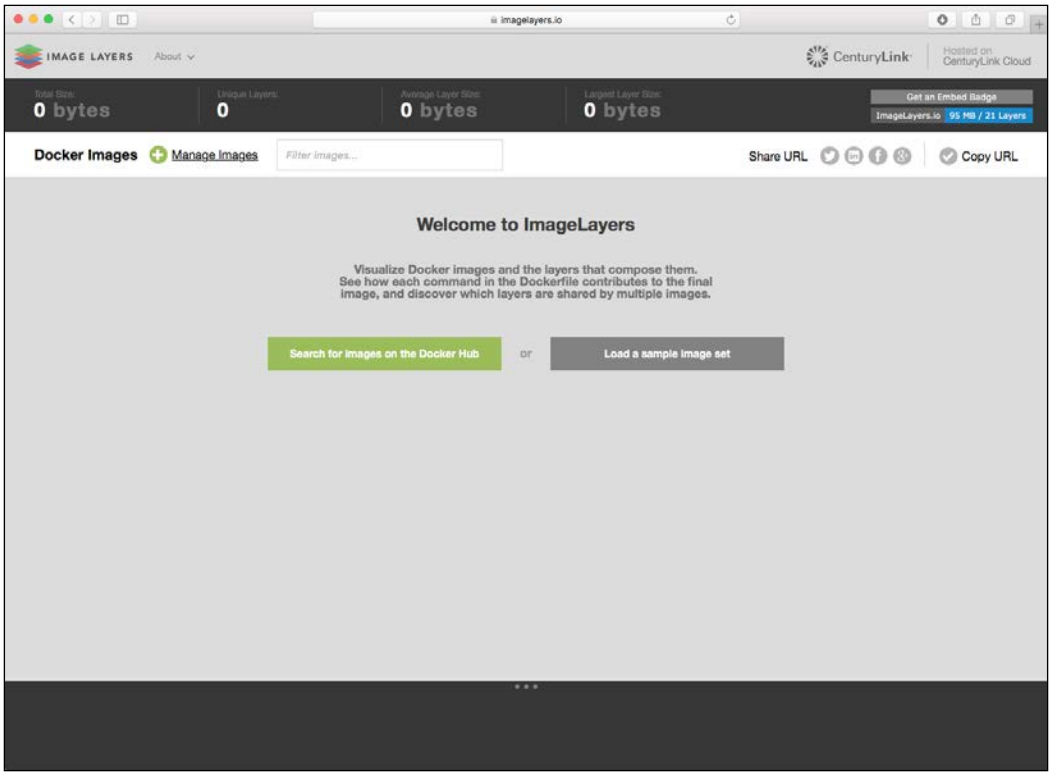
Next up in the navigation bar is **Images**. Again, like **Containers**, you can get all the information on all the images being used on your Docker host here. Information such as their IDs, what repositories they are from, their virtual sizes, and when they were created will be displayed here. Again, you can take some actions on your images. But for images, the only option you have is to remove them from your Docker host.



The last item in the navigation menu is **Info**. The **Info** section gives you a general overview of your Docker host, such as what Docker version it is running and how many containers and images are there. It also provides system information on the hardware that is available.

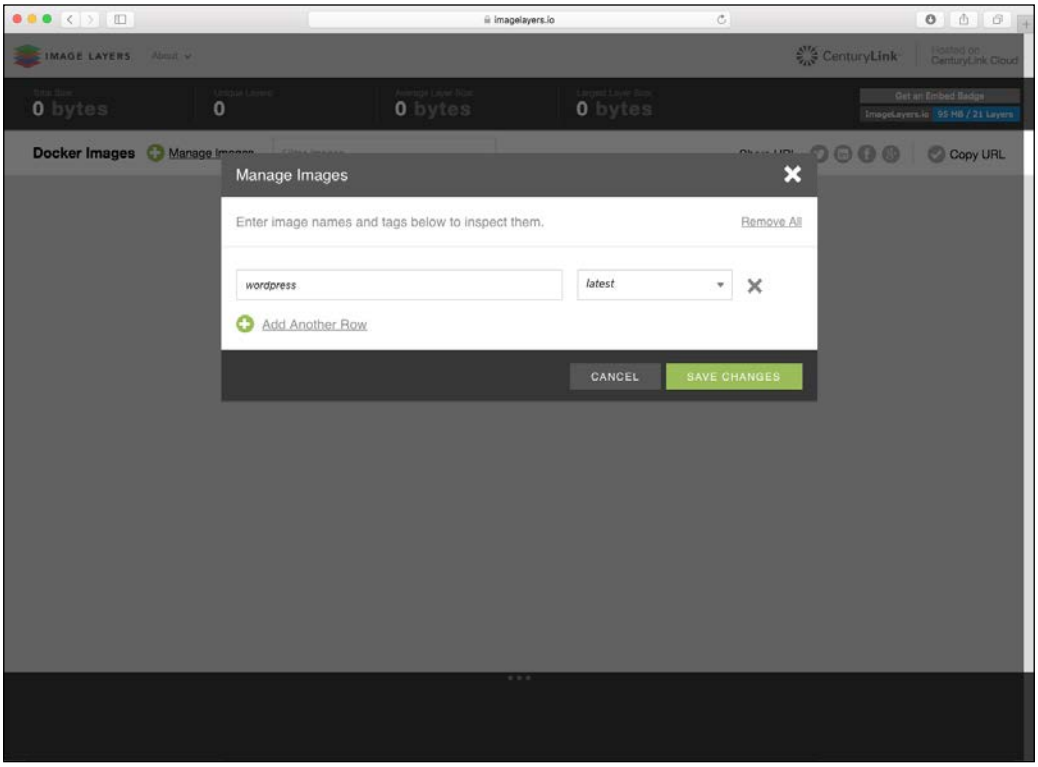
ImageLayers

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

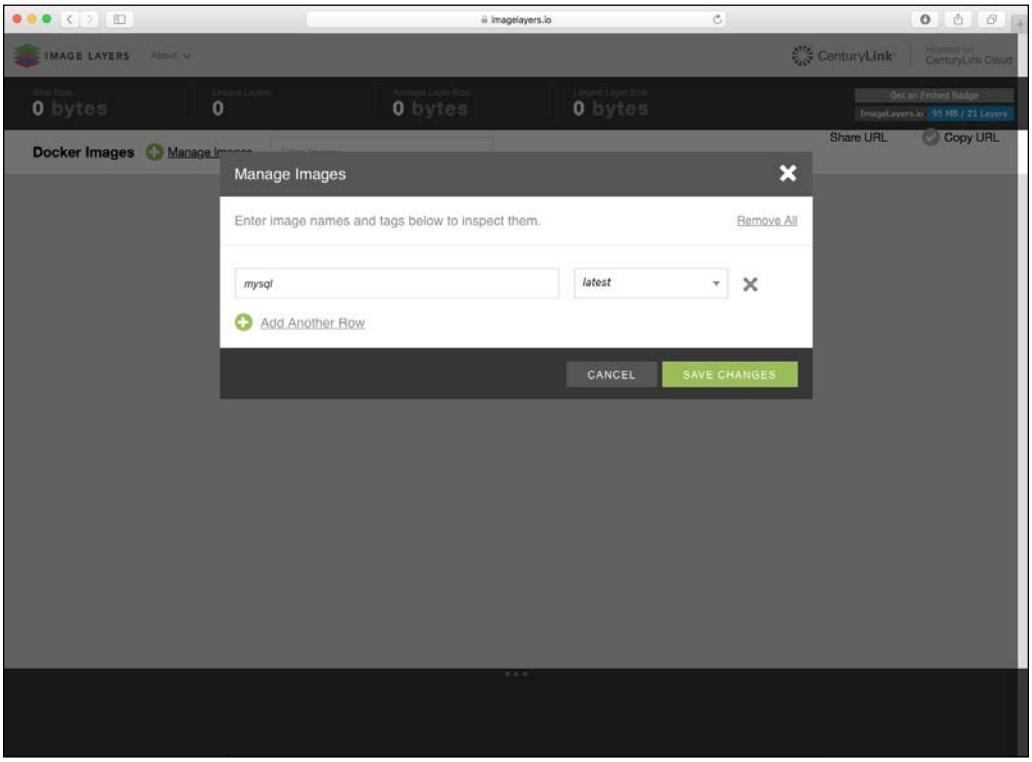


This screenshot is what you will be presented with while navigating to the ImageLayers website: `https://imagelayers.io`.

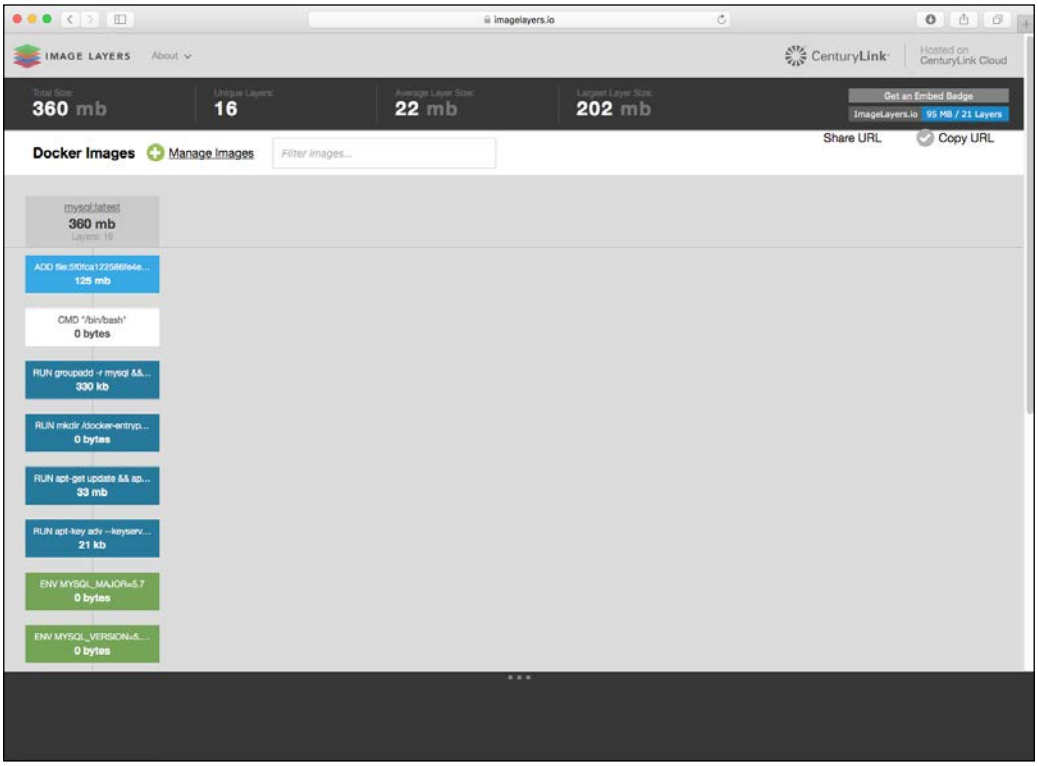
You can search for images that are on Docker Hub to have ImageLayers provide information on the image back to you. Or, you can load up a sample image set if you are looking at providing some sample sets or seeing some more complex setups.



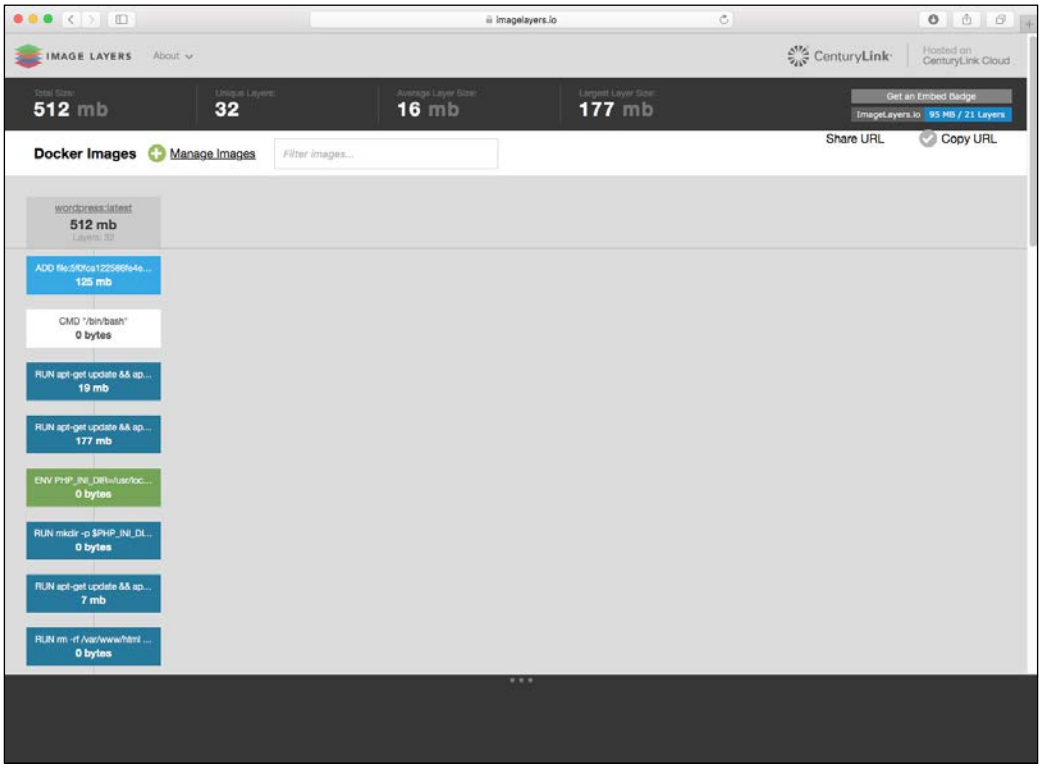
In this example, we are going to search for the `wordpress` image and select the **latest** tag. Now, you can search for any image and it will do auto-complete. Then, you can select the appropriate tag you wish to use. This could be useful if you have, say, a staging tag and are thinking of pushing a new image to your **latest** tag, but you want to see what impact it has on the size of the image.



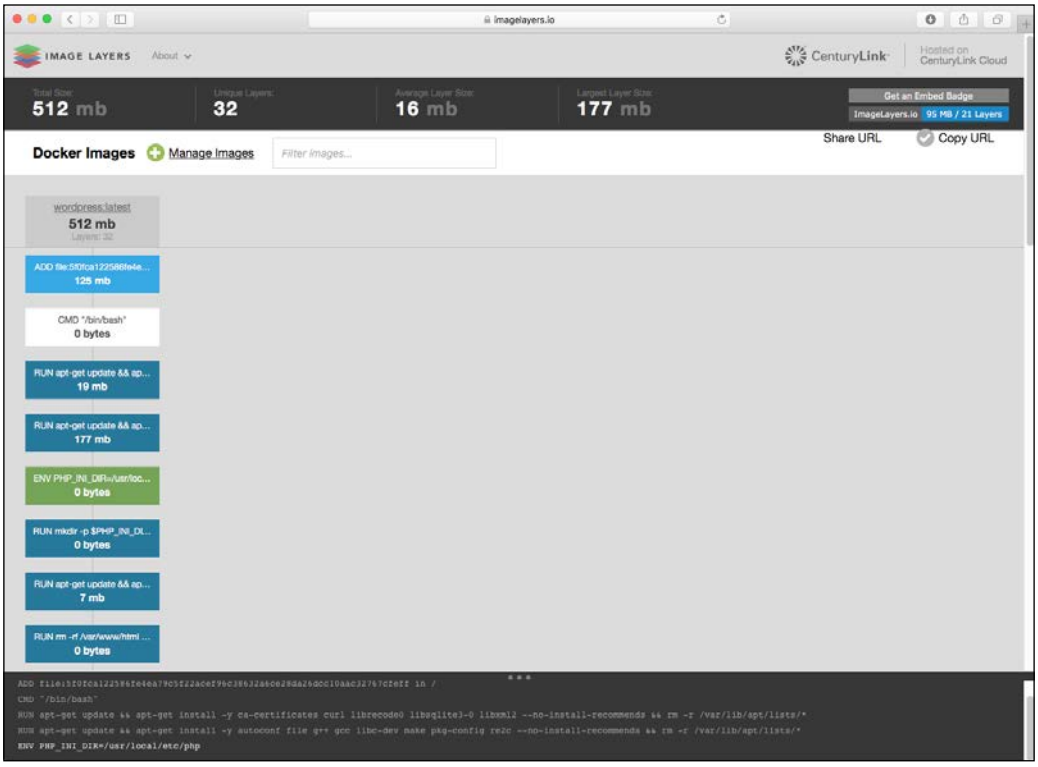
So, let's walk through an example. In this example, we are going to select a `mysql` image and the **latest** tag. We will use this since it is a common image that most people will use at some point in their Docker experience.



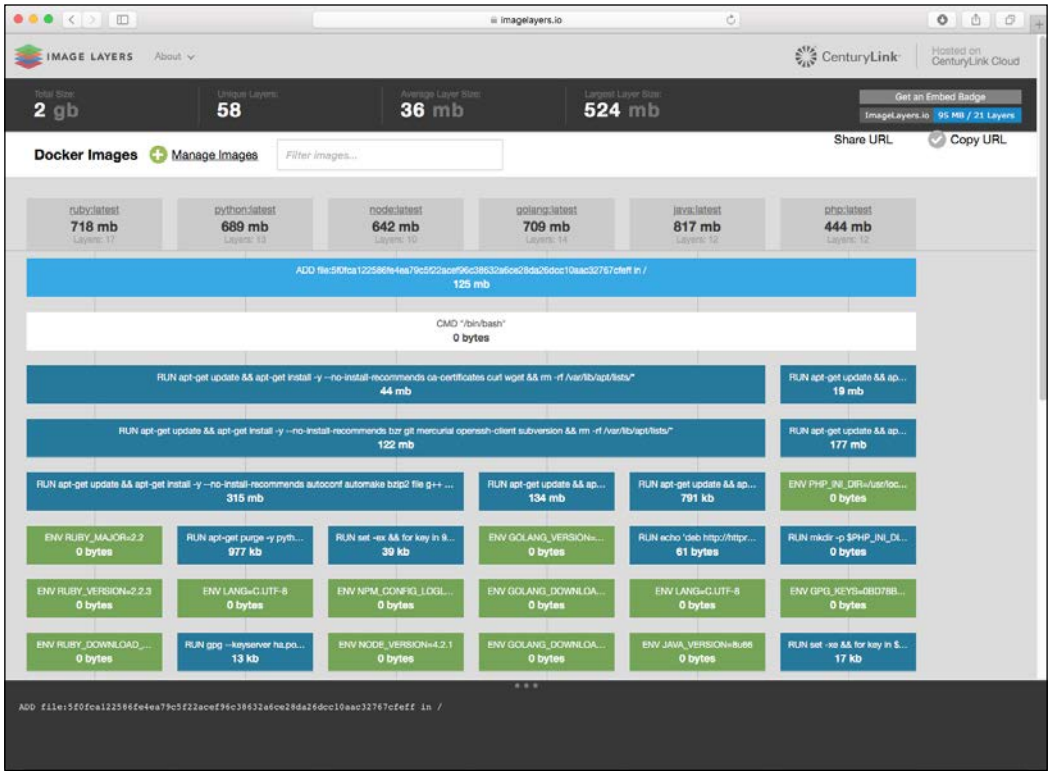
Once we click on **Save Changes** from the previous item, we will be shown something similar to the preceding screenshot (now, this will vary depending upon the image you have selected in your search). This displays some information at the top, such as the total image size, unique layers, the average layer size, and the largest layer size. This will help you hone in on a particular layer that might have grown wildly.



The layers are broken down on the left-hand side of the previous screenshot. We can see what action is being done at each level as the size that it adds to the overall image per layer.



Upon hovering on a particular layer, you will be given information on it at the bottom of the screen in a black box. This will show how each action is layered one after the other so as to help see the command structure of the image.



The preceding screenshot is an example of what you might see if you were to click on the sample image set from the main screen. As you can see, this one is quite complex; not only does it have a lot of layers, but it also has a lot of images that are being used. This could be something you would see while adding multiple images to see your desired output.

Summary

In this chapter, you have learned how to use Docker in a production environment as well as the key considerations to keep an eye on during the times of and before implementation.

In the next three chapters, we are going to be taking a look at some GUI applications that you can utilize to manage your Docker hosts, containers, and images. They are some very powerful tools and choosing one can be difficult, so let's cover all three!

10
Shipyard

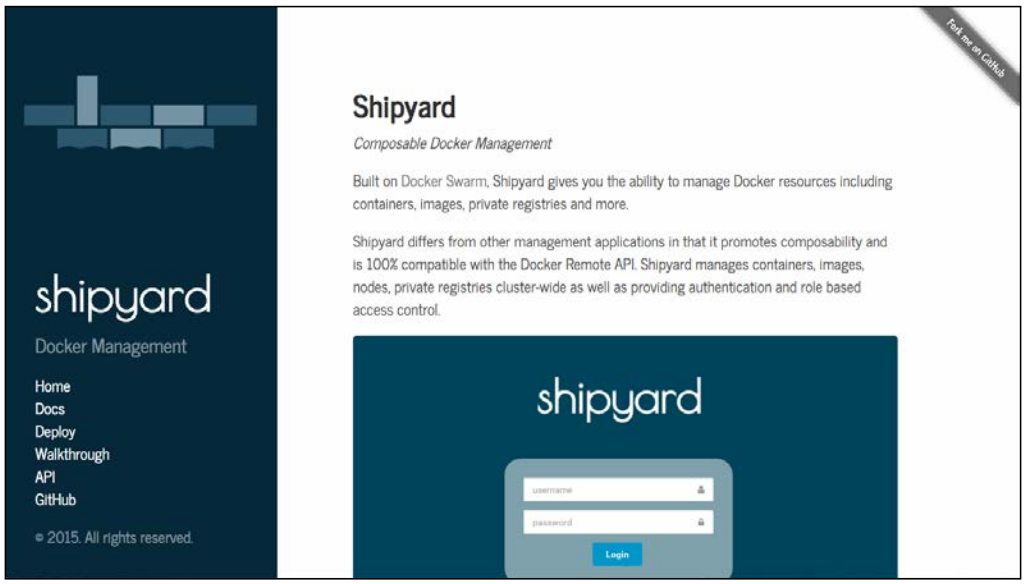
In this chapter, we will take a look at Shipyard. Shipyard is a tool that allows you to manage Docker resources from a web UI or a GUI interface.

The topics that will be covered are:

- Starting Shipyard
- The components of Shipyard

Up and running

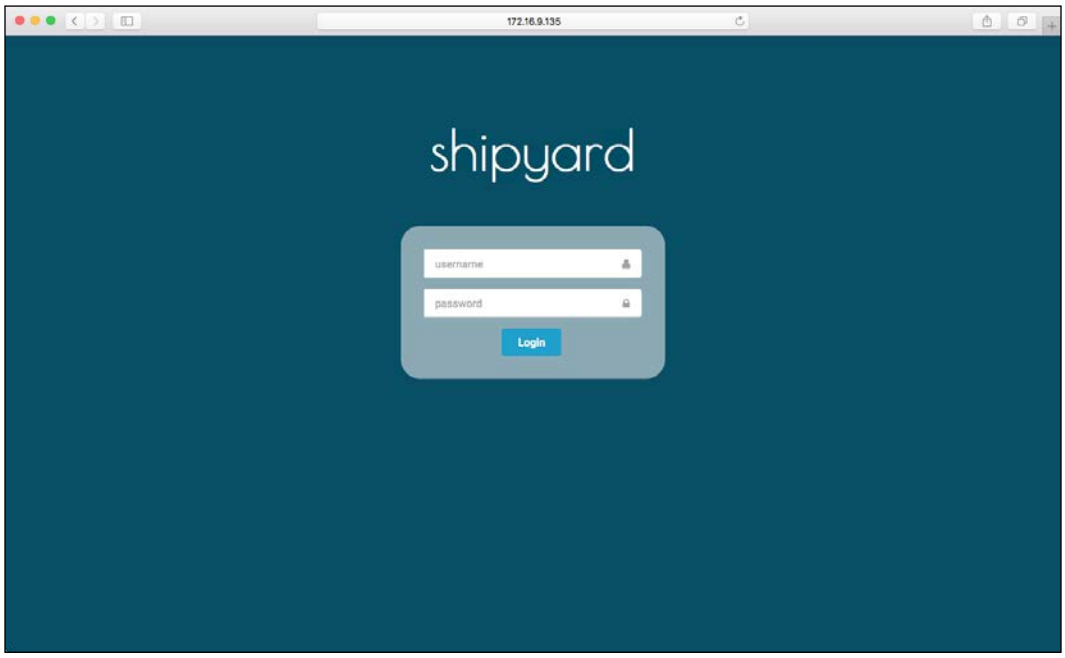
You will see a screen similar to the following screenshot while navigating your browser to the Shipyard website at `https://shipyard-project.com`:



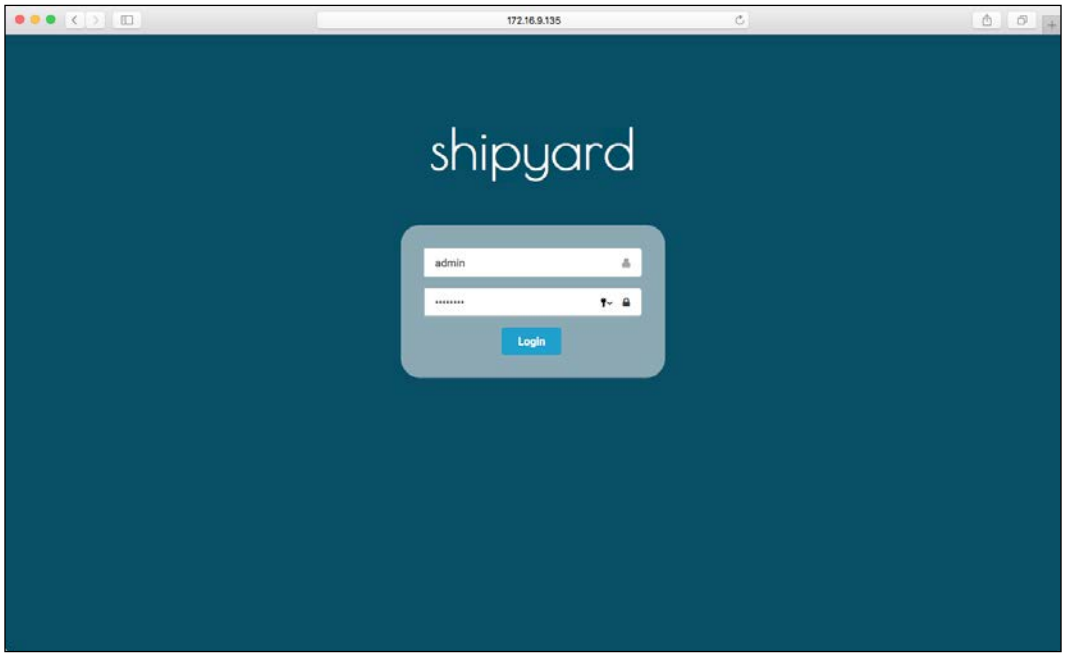
First, we need to get Shipyard up and running. To do this, we will execute the following commands:

```
$ docker-machine create --driver vmwarefusion ship1
$ docker-machine env ship1
$ eval "$(docker-machine env ship1)"

$ curl -sSL https://raw.githubusercontent.com/scottpgallagher/shipyard/
master/deploy | bash -s

$ docker-machine create --driver vmwarefusion ship2
$ docker-machine env ship2
$ eval "$(docker-machine env ship2)"

$ curl -sSL https://raw.githubusercontent.com/scottpgallagher/shipyard/
master/deploy | ACTION=node DISCOVERY=consul://<IP_ADDRESS_of_SHIP1>:8500
bash -s
```

You will see the following login screen when you first navigate to the `shipyard` web instance:



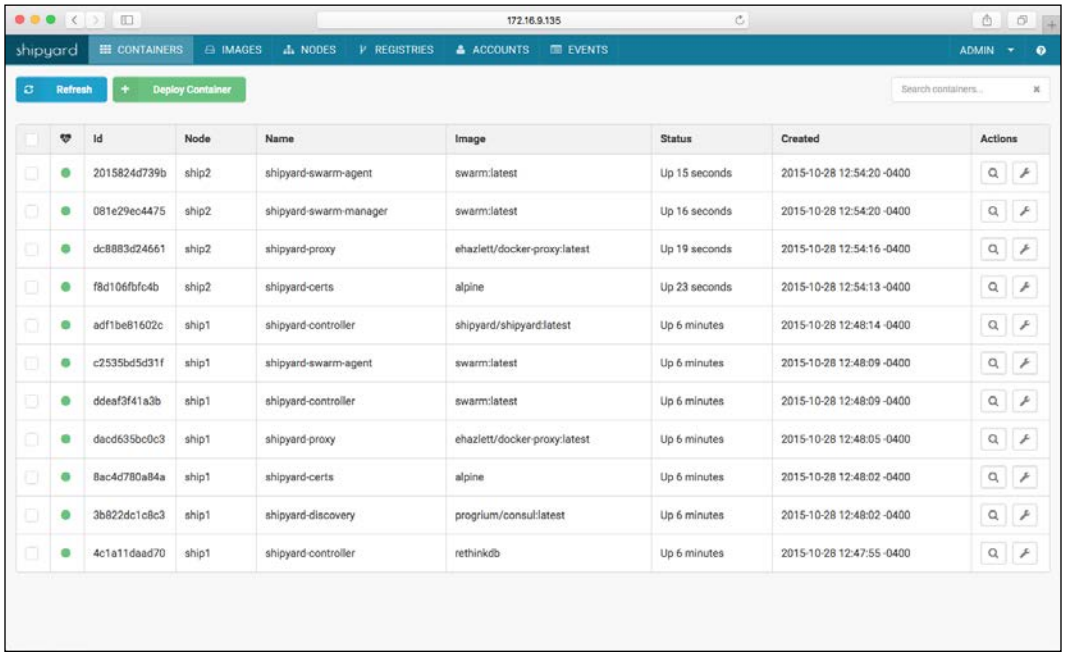
The URL is always the IP address of your Docker host. It runs on port `8080` (that is, `172.16.9.135:8080`).



The default username is `admin`. The default password is `shipyard`. Enter these details and click on **Login**.

Containers

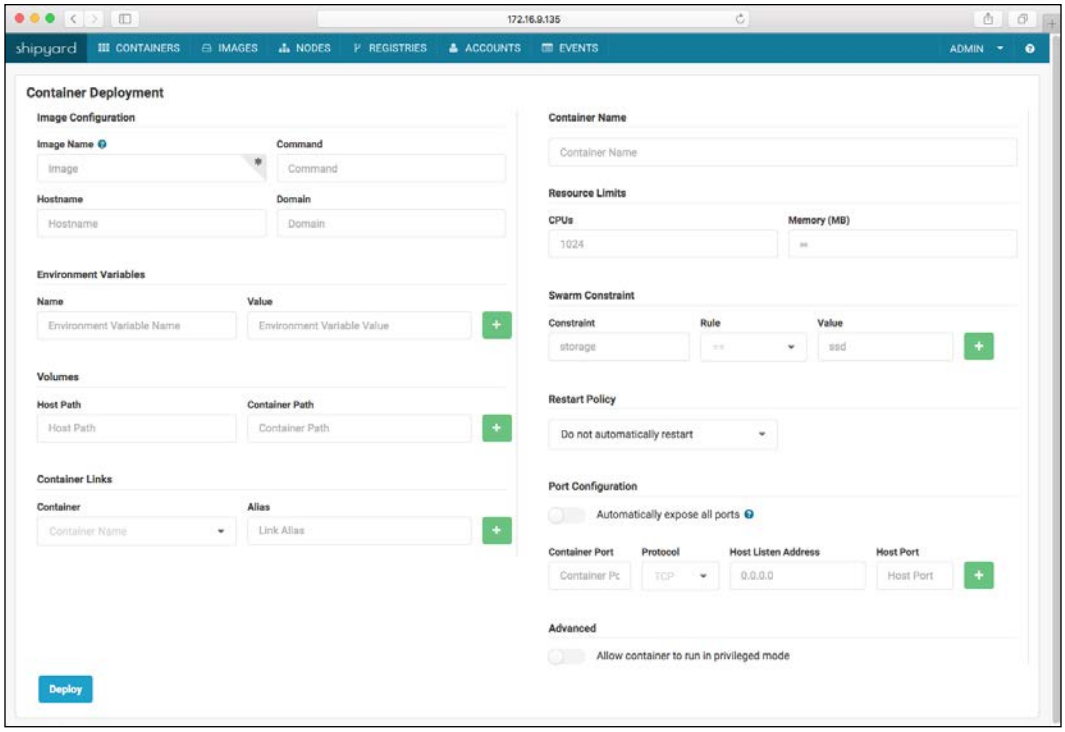
After logging in, you will be taken to the main dashboard or the **CONTAINERS** section as follows:



There is a lot you can do in this section. We will cover all of it step by step in the following and the *Back to CONTAINERS* section.

Deploying a container

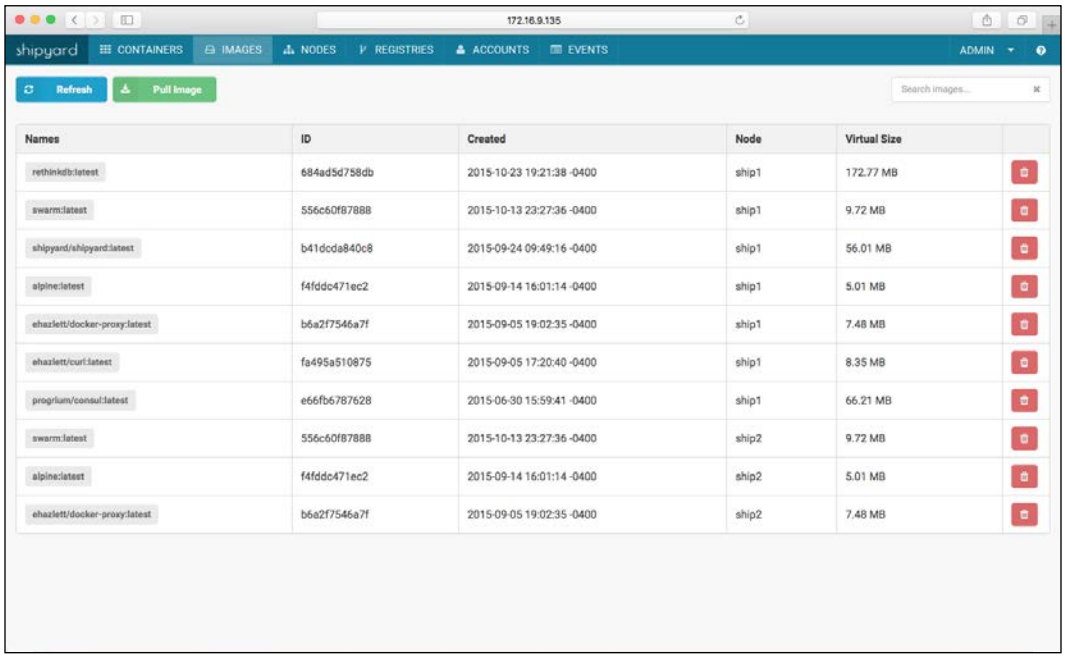
The first thing we will tackle on this page is the **Deploy Container** button.



There is a lot of information to digest here. But at the same time, this is the information you are used to providing either in your Dockerfile or your `docker-compose.yml` file. Once you type in all your information, you're ready to deploy. So, go ahead and click on the **Deploy** button.

IMAGES

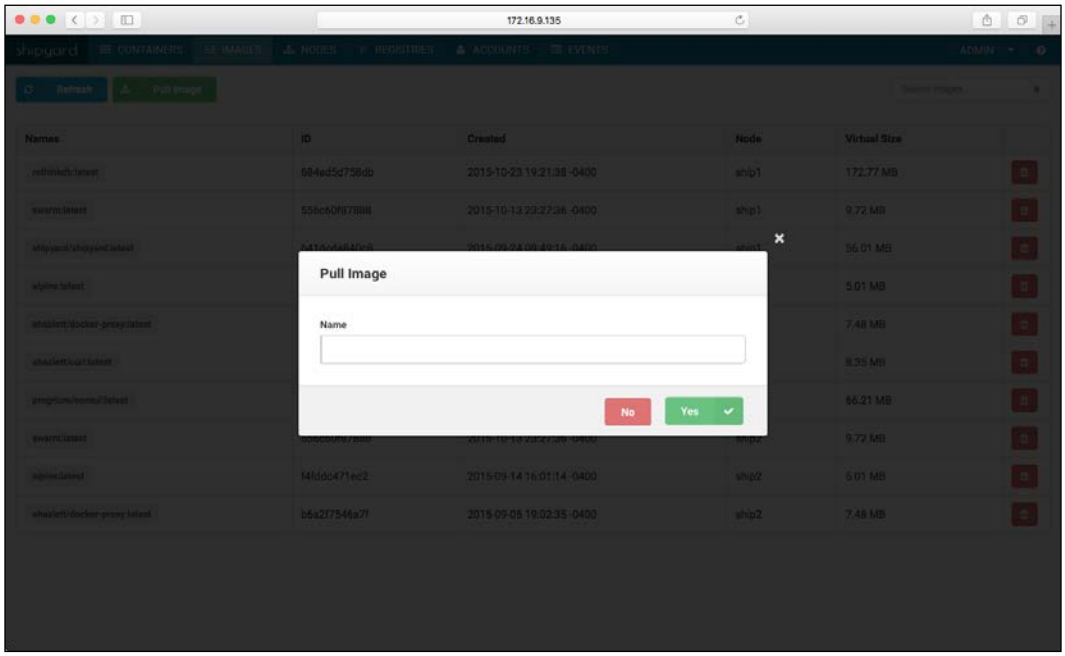
At the top of the screen, we can see a blue navigation bar. Moving on from the **CONTAINERS** section (for now), we will now cover the **IMAGES** section. In the **IMAGES** section, we can see all the images that are being used across our hosts.



We can see information such as the name of the image, its ID, when it was created, what node or Docker host it's running on, and its virtual size. We also have the option to delete the images by using the red trash can icon.

Pulling an image

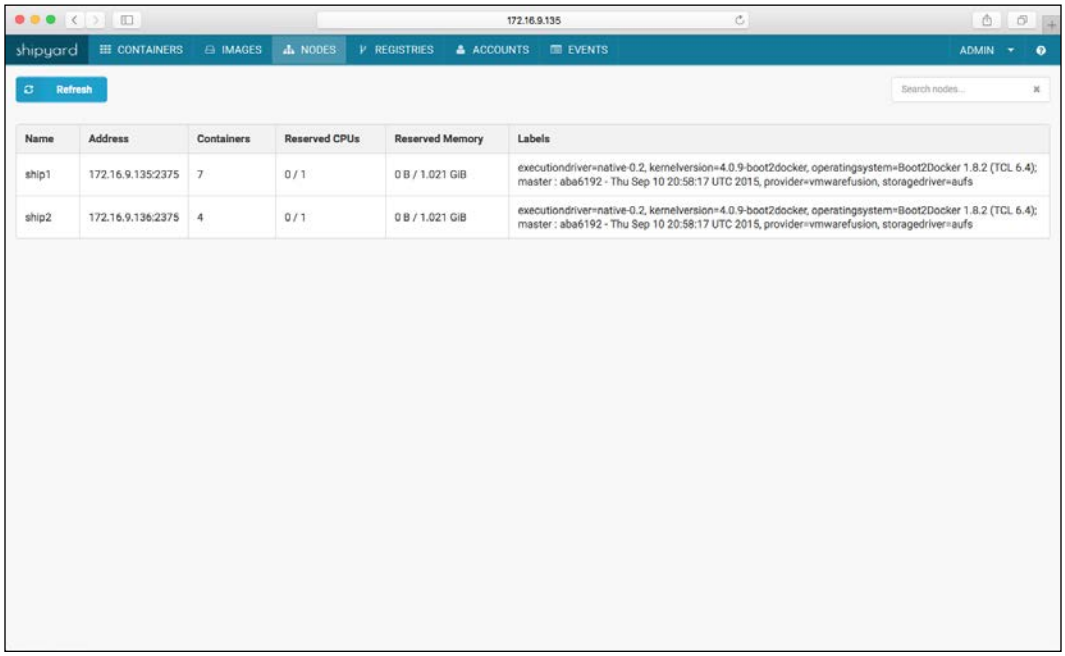
Now, one thing that we didn't cover was the **Pull Image** button. By clicking on this, you will be presented with the following screen:



On this screen, you can enter an image name as well as its tag and have it pulled. You could then go back to the **CONTAINERS** page and deploy that image. Now, this will work not only with Docker Hub, but with any other repository you add later to Shipyard.

NODES

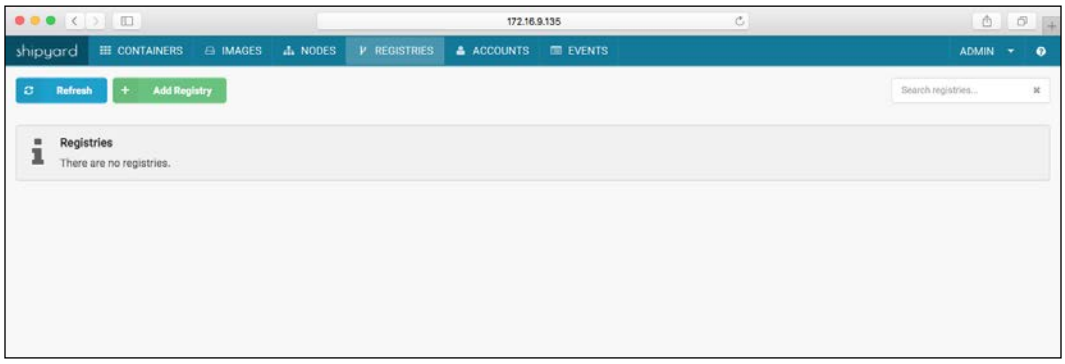
Next up is the **NODES** section. This section shows information on what nodes or Docker hosts you have connected to Shipyard.



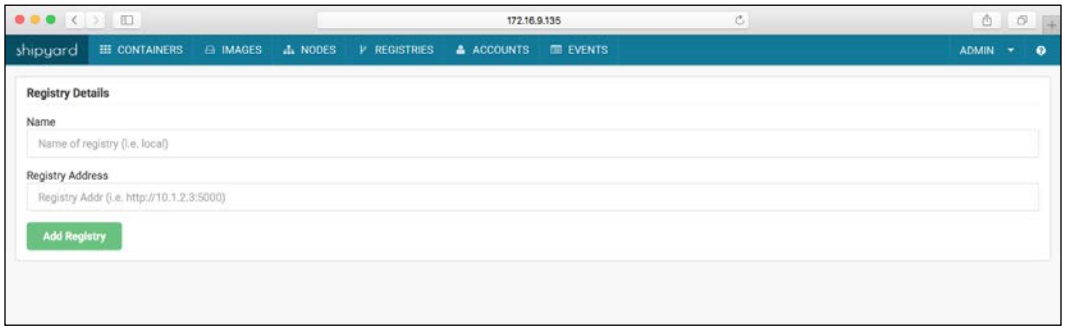
It will give you information such as the name of the node, its IP address, the number of reserved CPUs and memory, as well as the labels that provide information such as what version of the Linux kernel or Docker is being used.

REGISTRIES

Next up is the **REGISTRIES** tab. This is where you can add registries beyond Docker Hub.



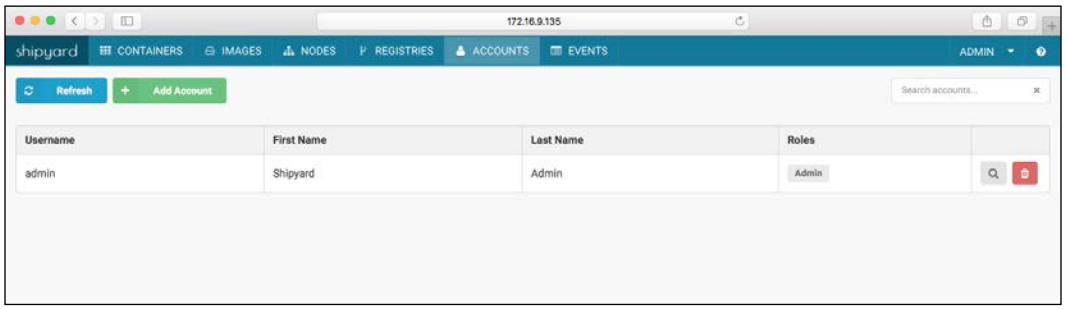
On clicking the **Add Registry** button, you will be taken to the following screen:



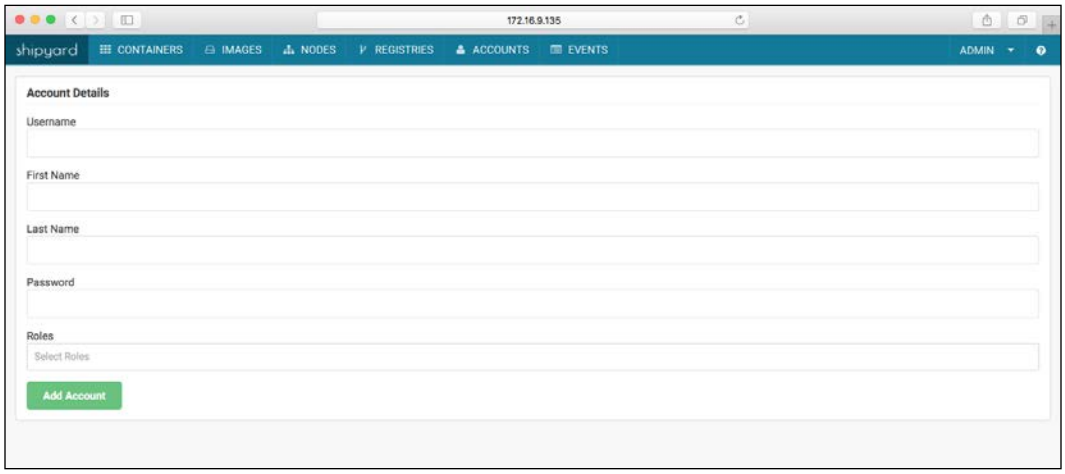
This will allow you to enter information about the registry such as its name and registry address, which would include the IP address or the DNS name and the port it is running on.

ACCOUNTS

Next up is the **ACCOUNTS** tab where—you guessed it—you can add or remove accounts.



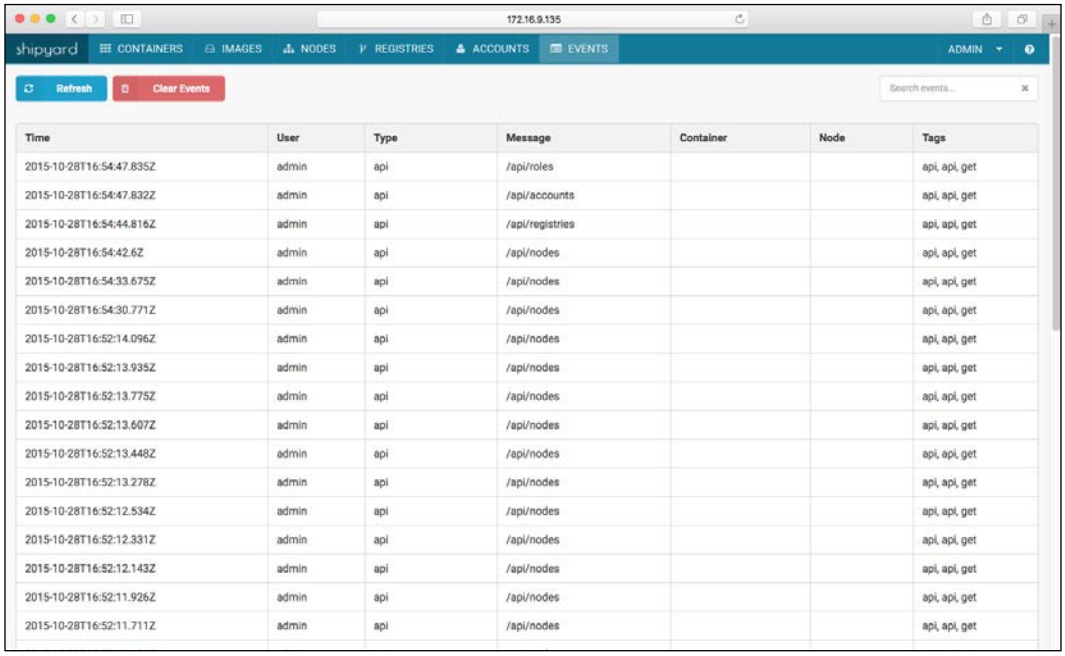
In the following screenshot, you can see what information is needed when you add a new account:



Information such as the username you want to use, your first and last names, the password you want to assign to it, and lastly your assigned role.

EVENTS

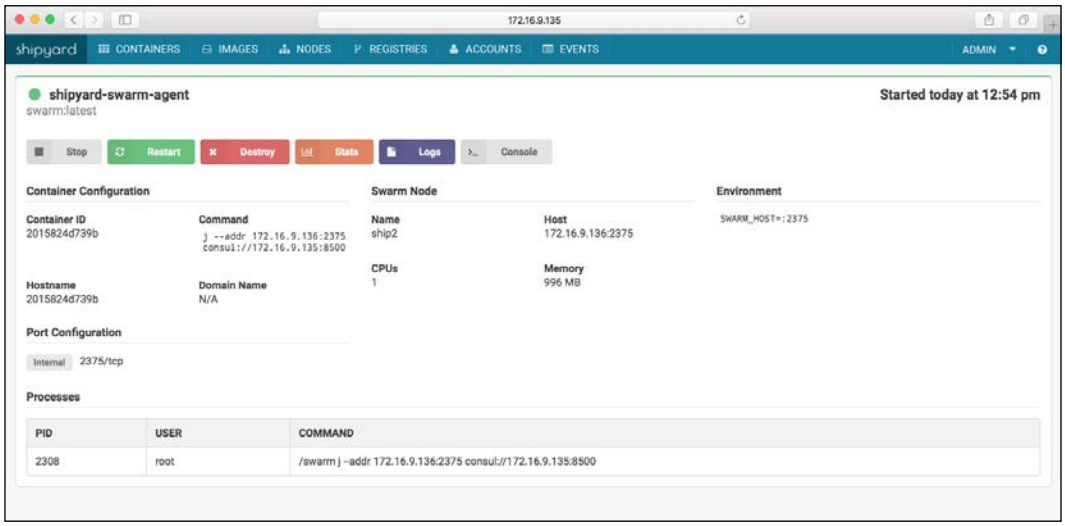
Okay, last up is the **EVENTS** tab that will display the following screen:



This tab will show you all the events that have occurred and what user accounts they were initiated from. Information such as the message, container, node, and tags are also displayed.

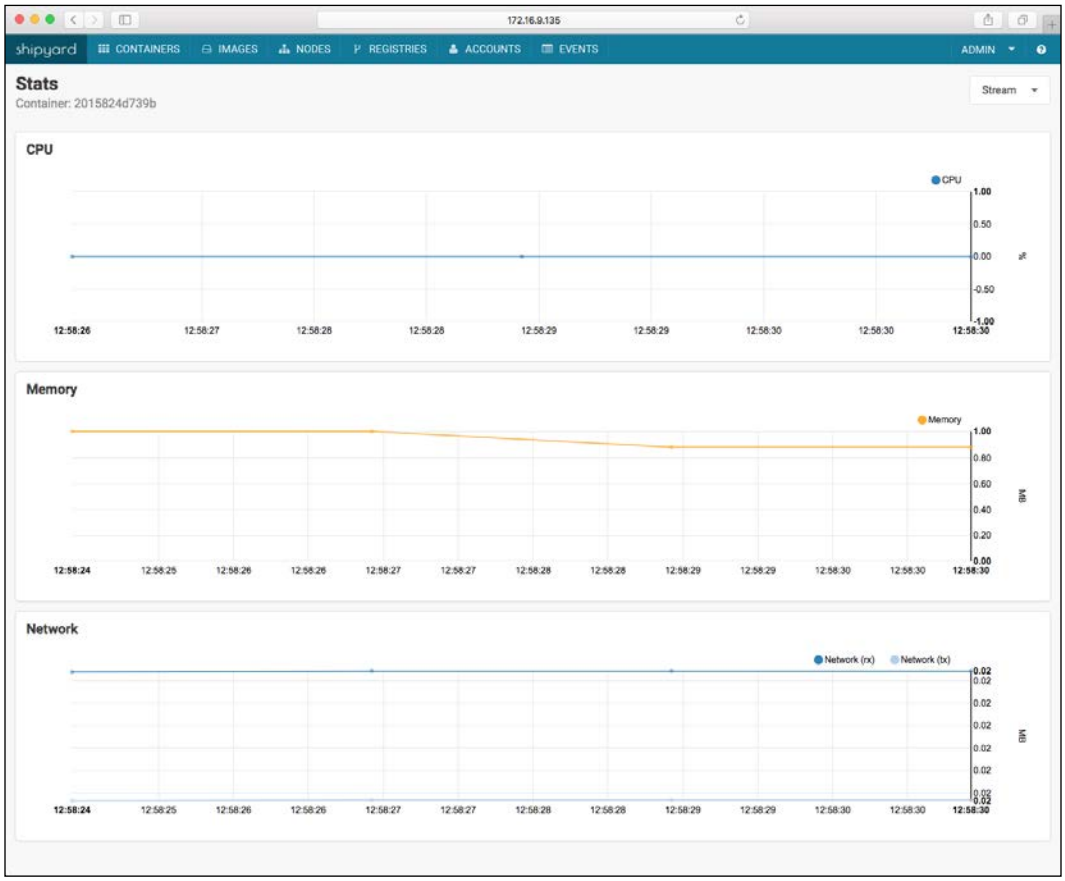
Back to CONTAINERS

We jump back to the **CONTAINERS** section where we saw all our containers. We can also click on the magnifying glass on the right-hand side of each container to get pulled to the following screen:

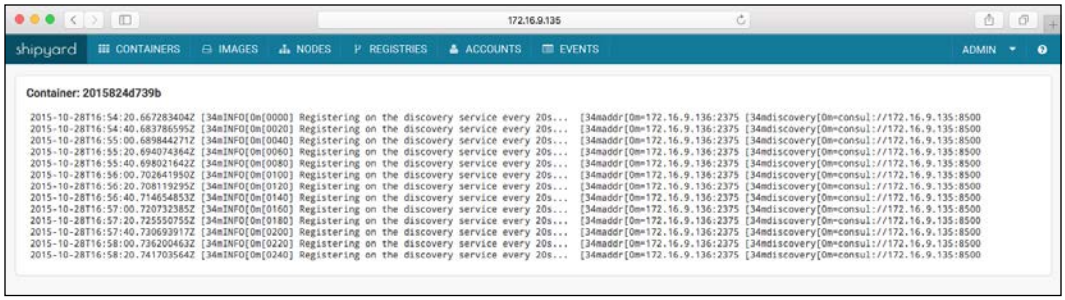


We can then get information on that running container and manipulate it. We can stop, restart, or destroy (or remove) it. We can also see information on it such as the command that it's running, its port, its IP address, and its node name.

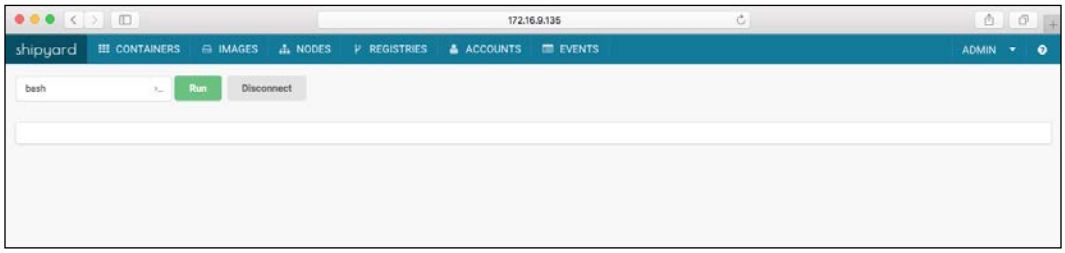
Clicking on the **Stats** button, we can see information pertaining to the running container such as the CPU, memory, and network information.



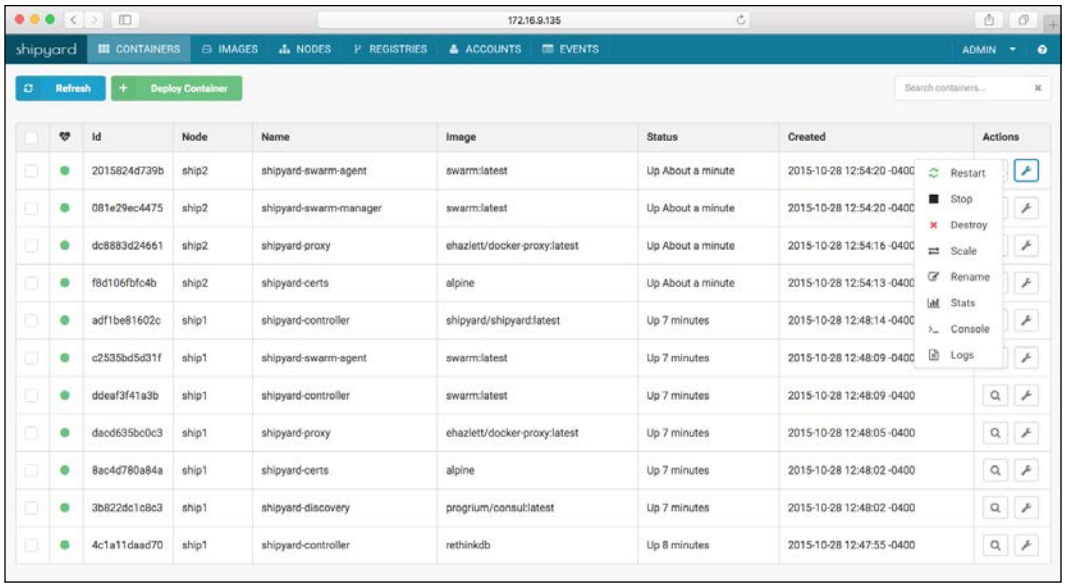
Clicking on the **Logs** button will show you everything that is going on with the container. In this case, the container is polling consult for new information ever so often.



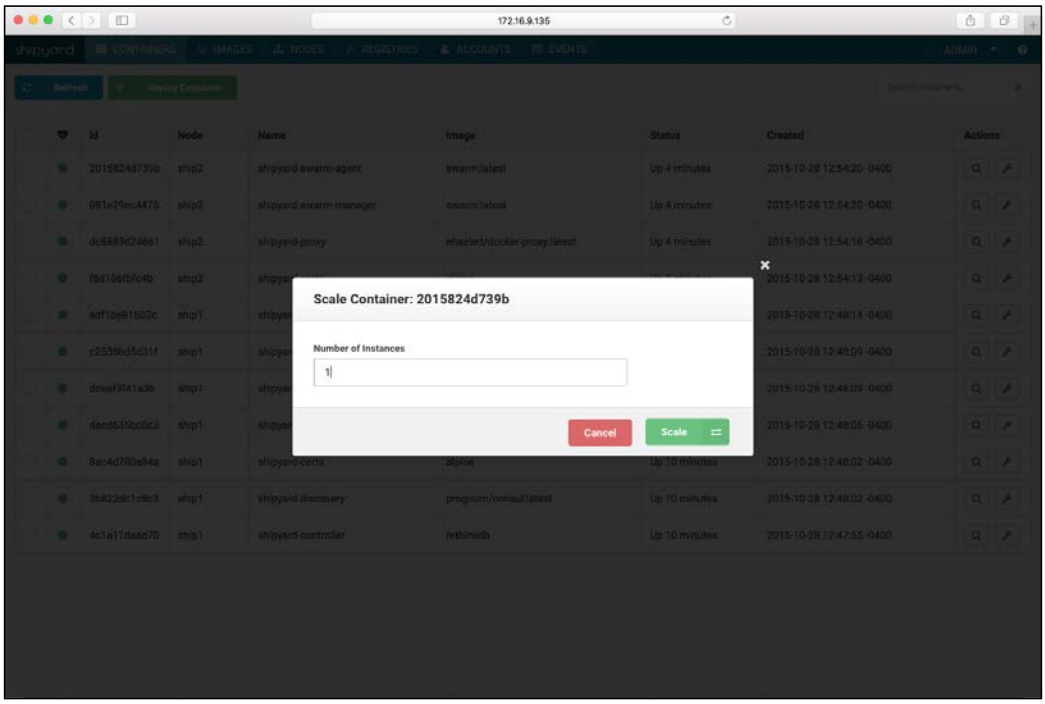
Now, the **Console** button is interesting. It will allow you to actually run a command against the container and provide the output from that command.



There are other ways to manipulate these containers as well. We will go back to the **CONTAINERS** page, where we can see a list of all our containers and their status. We have some controls here to restart, stop, and destroy the container.

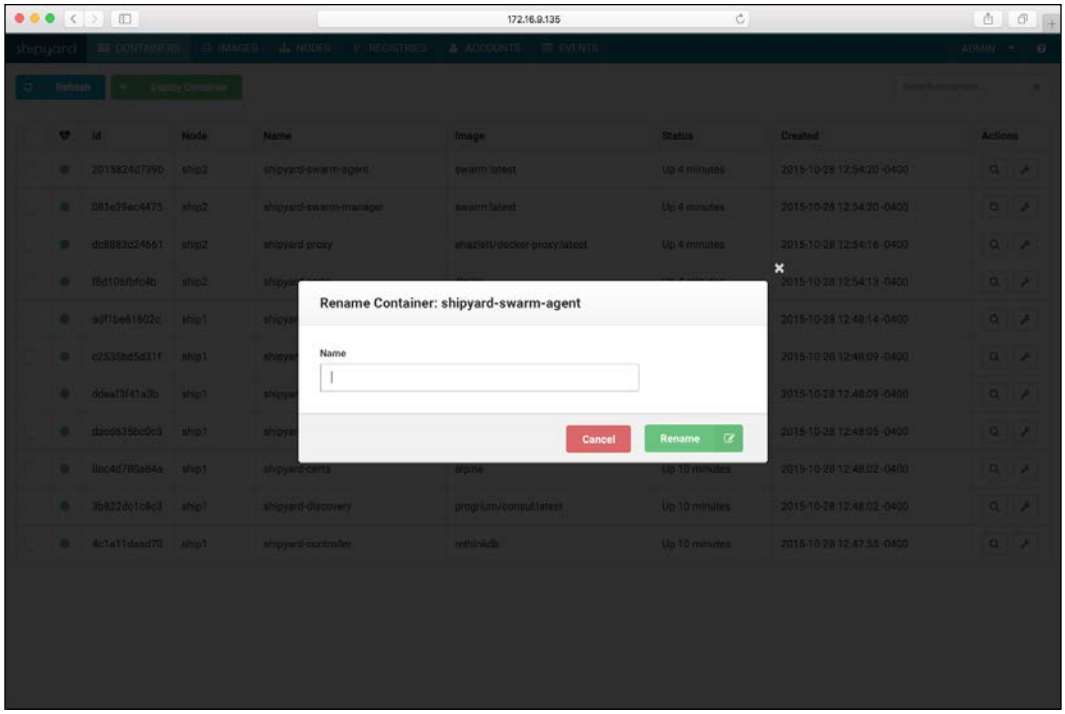


We can also scale or rename the container and get to the other areas we saw earlier such as **Stats**, **Console**, or **Logs**.



You will be taken to this section if you click on the **Scale** option. This will allow you to enter a numerical value and scale the instance up as far as you like.

You can also click on the **Rename** option to rename the container to anything you wish.



Do be careful; use a name that helps you identify the container.

Summary

As you can see, Shipyard is very powerful and will only continue to grow and integrate more of the Docker ecosystem. With Shipyard, you can do a lot of manipulation with not only your hosts, but also the containers running on the hosts.

In the next chapter, we will take look at another GUI tool to manage your Docker hosts, containers, and images, and that is **Panamax**.

11
Panamax

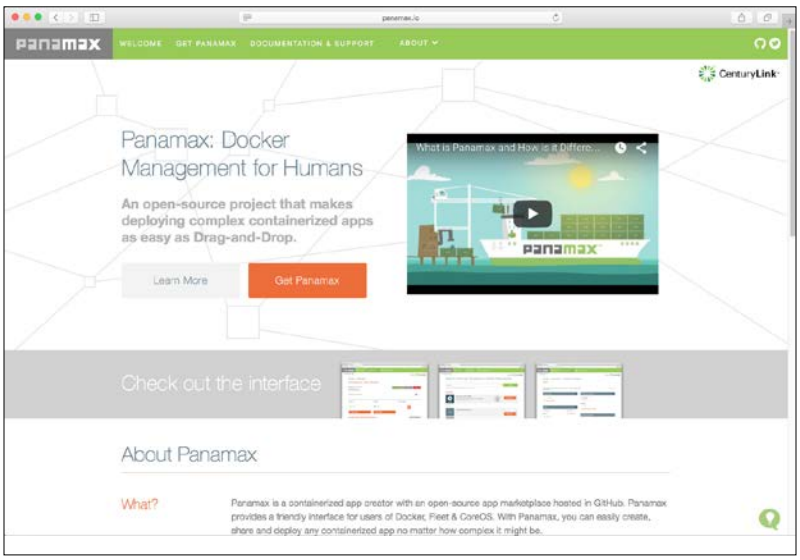
Panamax is another open source project that helps with deploying Docker environments by using a GUI interface to allow you to control just about everything that you can with the CLI.

In this chapter, we will cover:

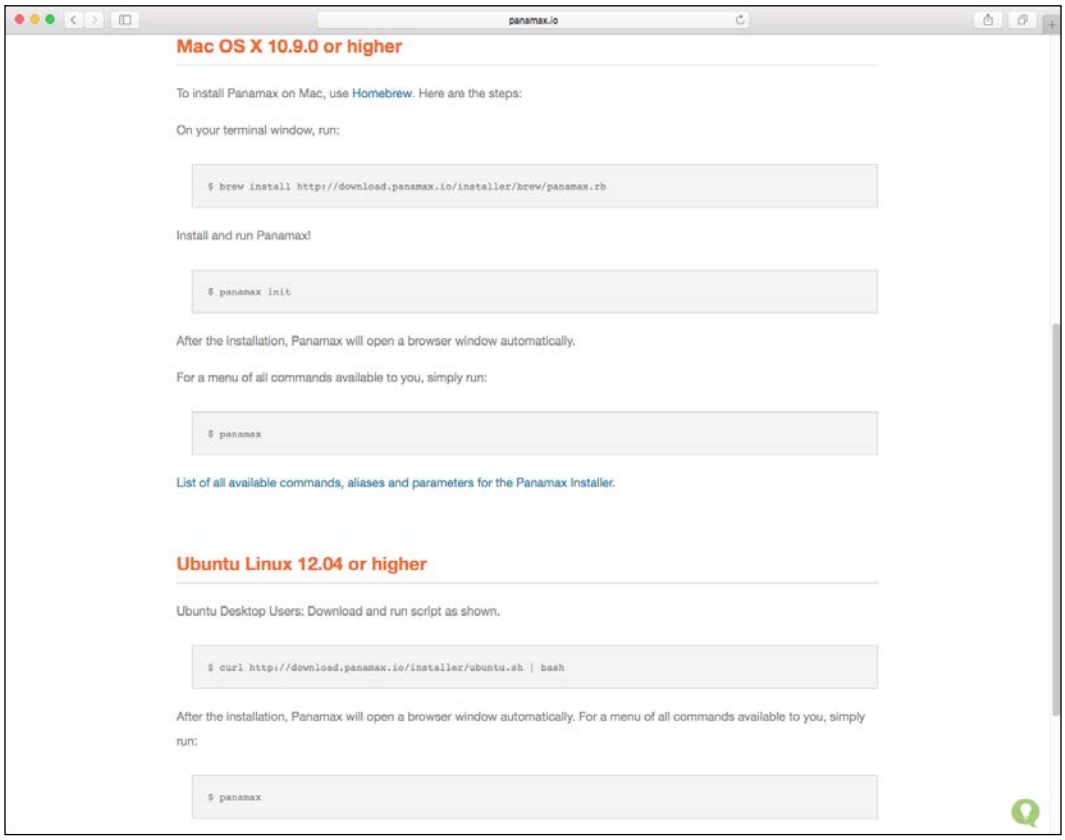
- Installing Panamax
- What after installing?

Installing Panamax

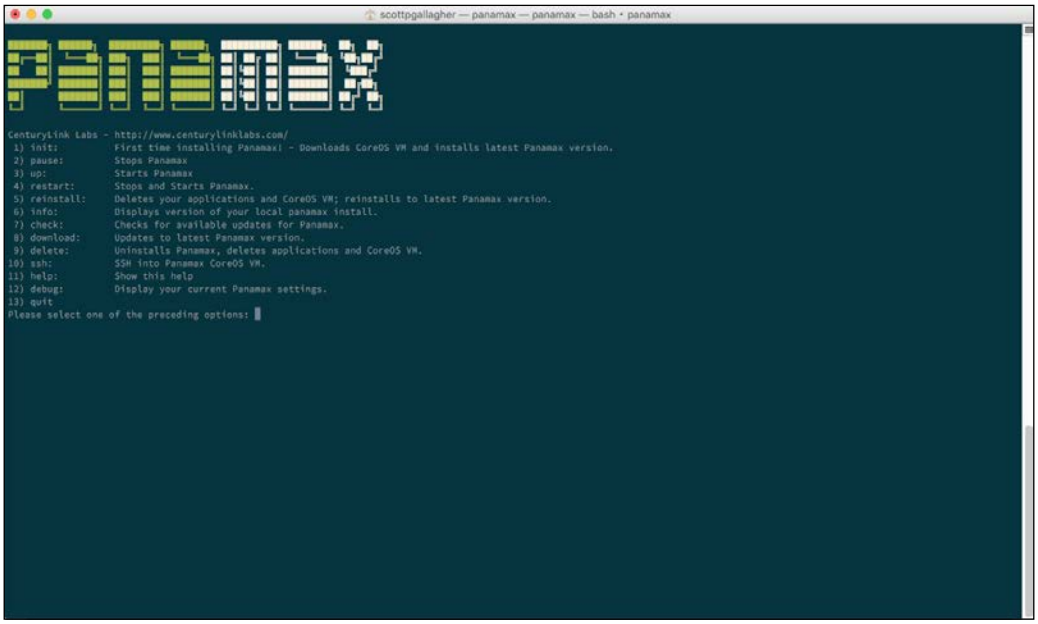
You will see the following page while navigating to the Panamax website at `http://panamax.io/`:



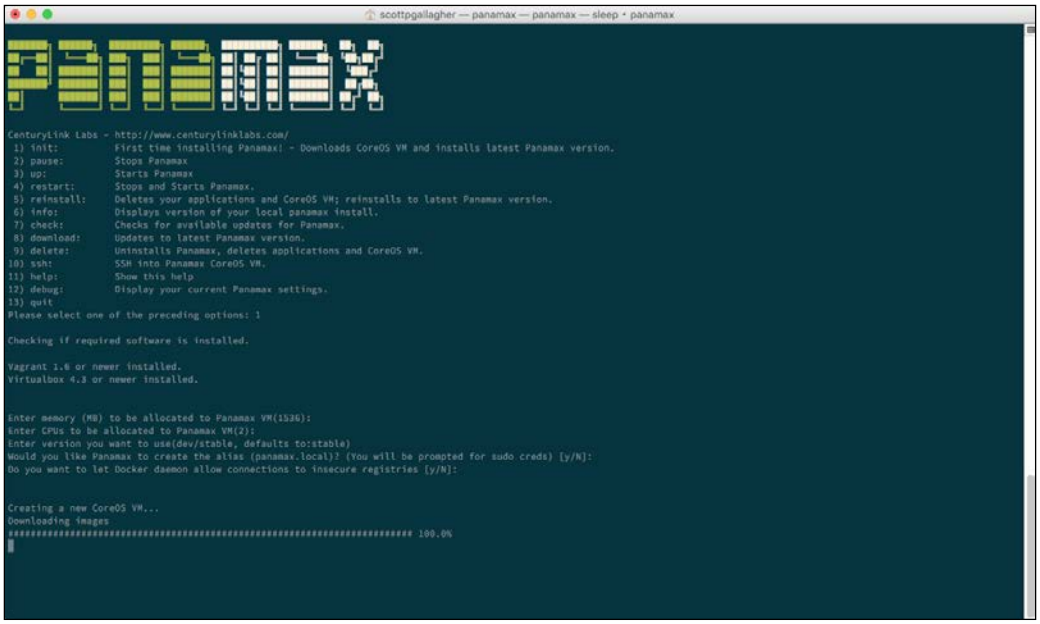
Next, you will see the instructions to install Panamax on both Mac OS X and Ubuntu:



After running the `panamax init` command and then the `panamax` command, you will see the following options:

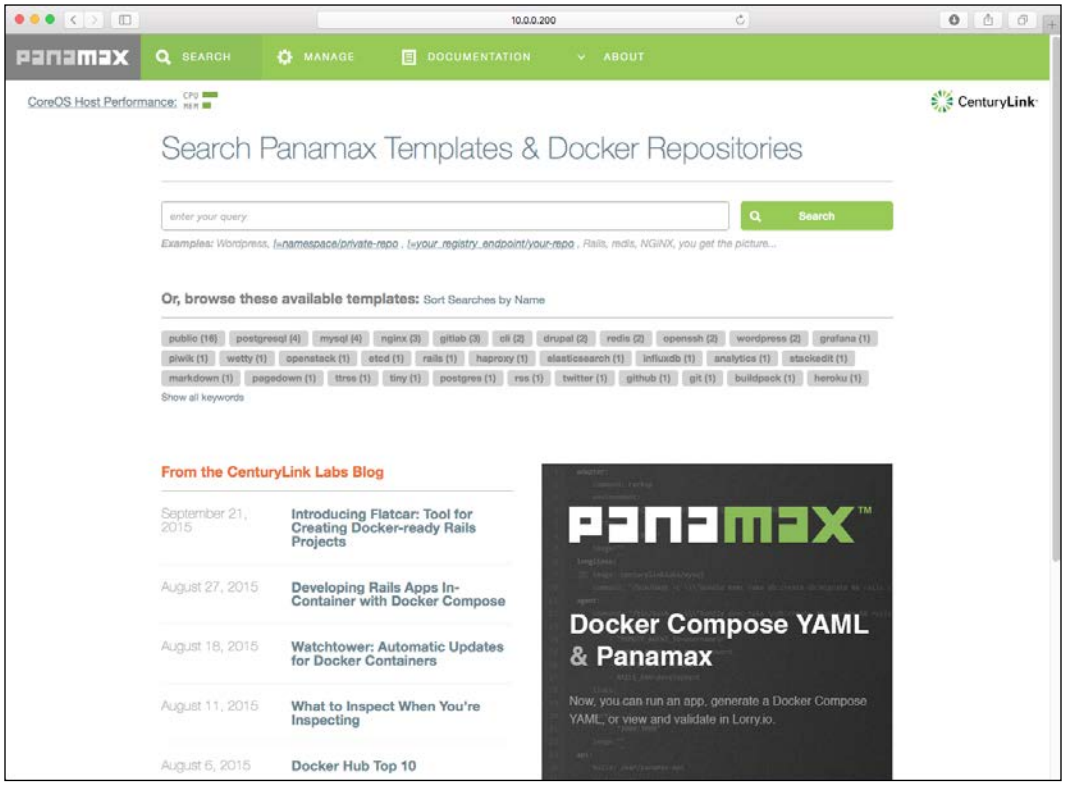


Upon selecting the first selection `init`, all the magic starts to happen.



Once all the magic is complete, you will be taken to the Panamax dashboard.

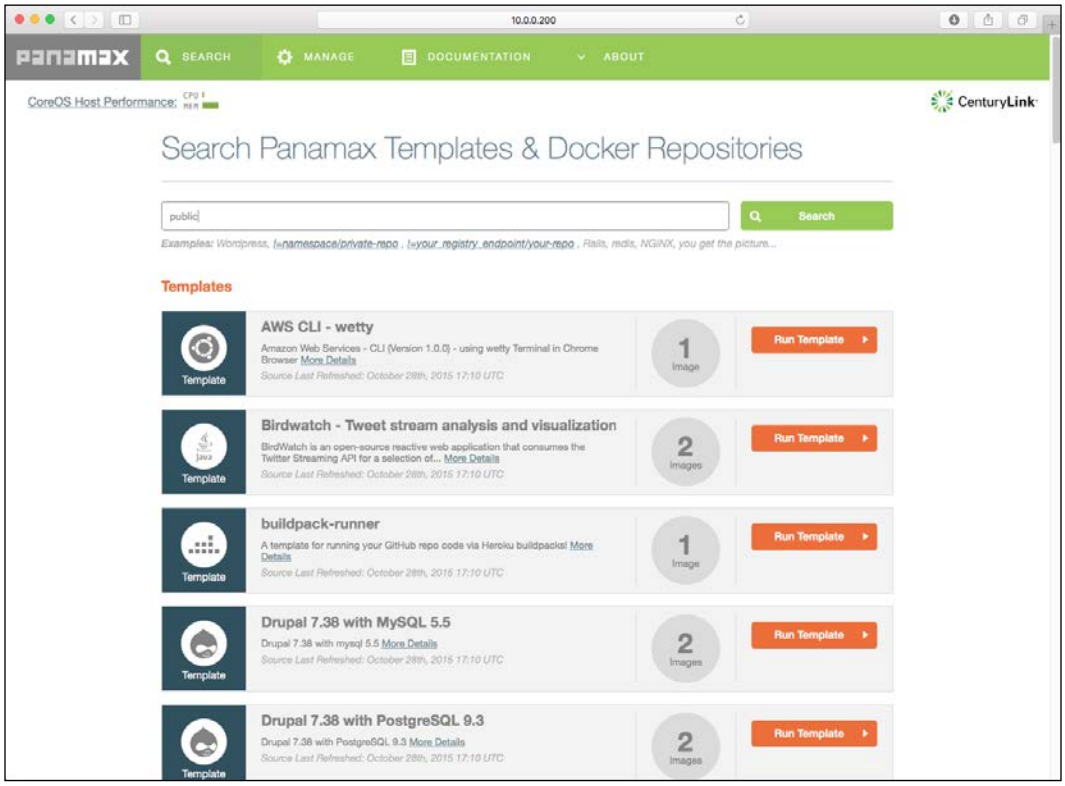
The following screenshot shows you what you will see once the installation has been completed and the browser page has been loaded for you:



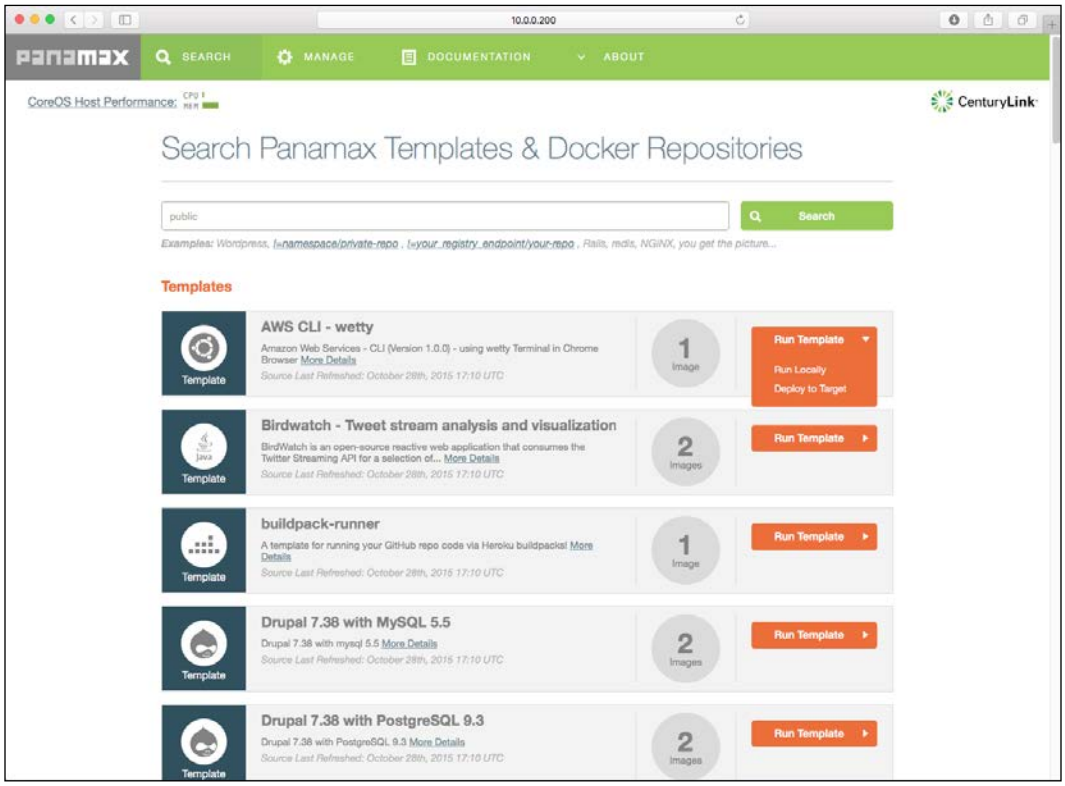
On this page, you can search for images that are on Docker Hub or browse the available templates that Panamax has to offer. You can also see the performance of the host that is running Panamax at the top with information such as the CPU and memory usage.

An example

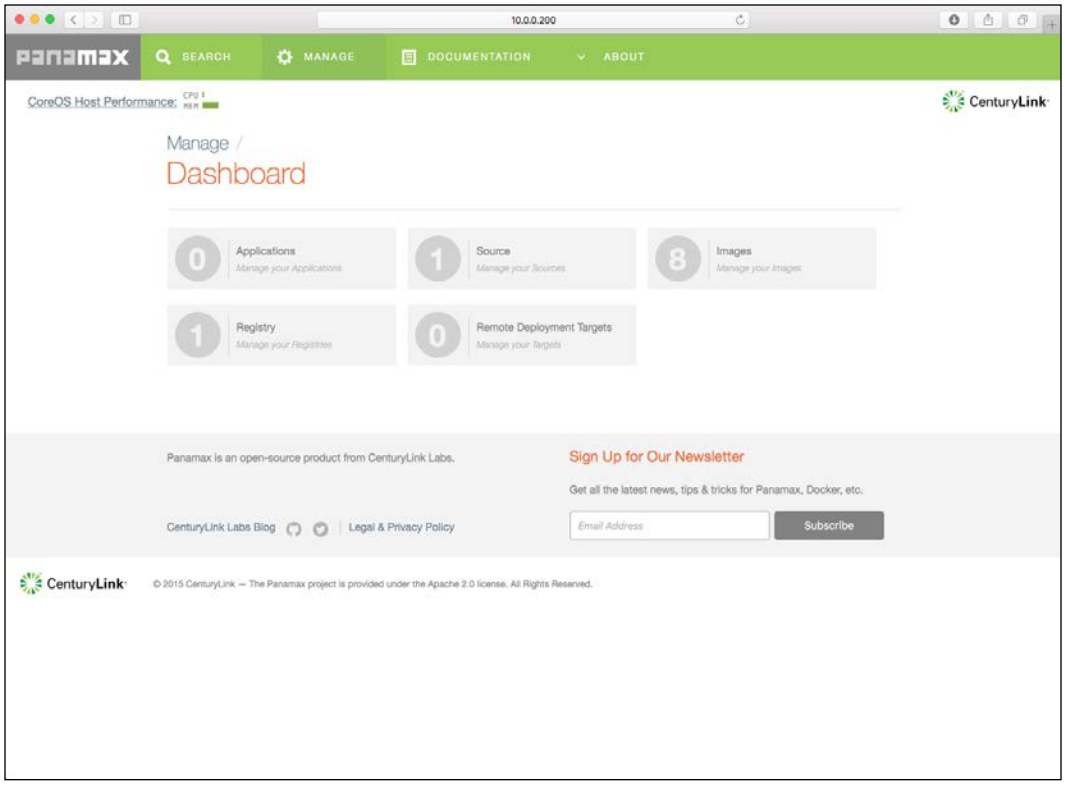
For this example, we select `public` from its available templates and use the `AWS CLI - wetty` image to run.



You can see information such as the image name, the description, how many images it will contain, and the option to run the template.



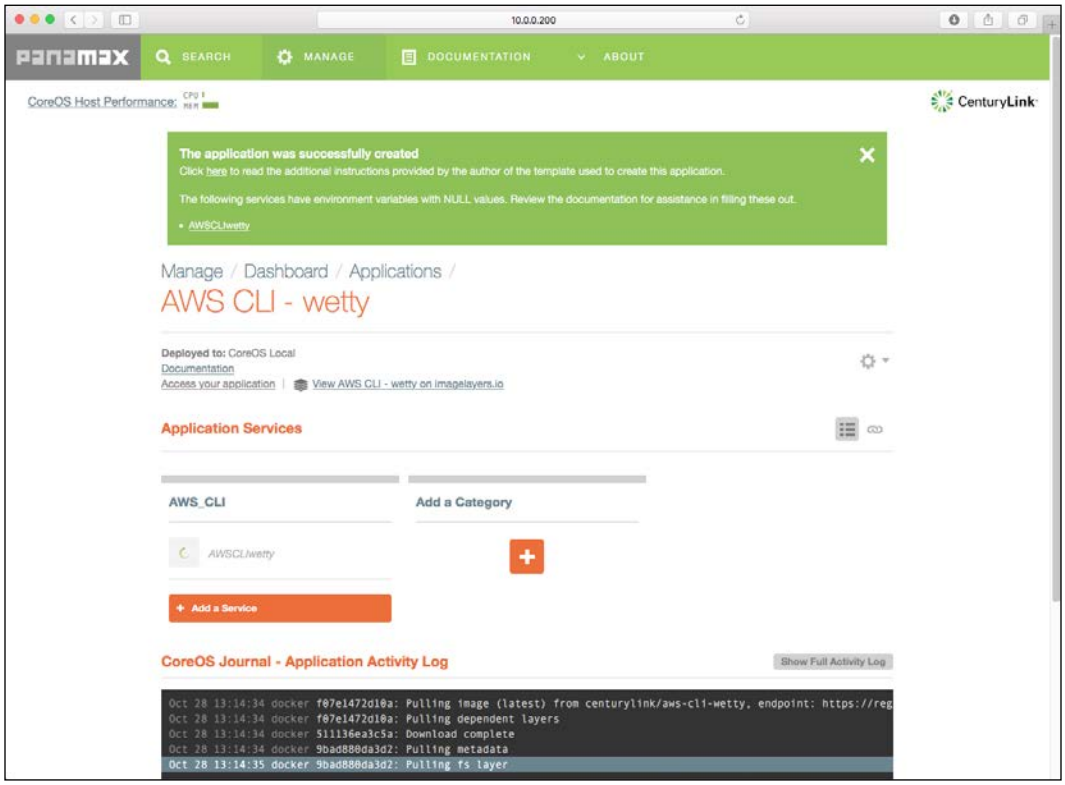
Upon clicking the **Run Template** button, you will get two options. You can run it locally or deploy it to a target, such as the cloud. For this example, we will choose to run it locally.



After you choose to run it locally, you will want to navigate to the **Manage** section. In this section, there are multiple subsections that you can then navigate to such as **Applications**, **Sources**, **Images**, **Registry**, and **Remote Deployment Targets**. It will show you how many of these each subsection has in it. We will take a look at each of these next.

Applications

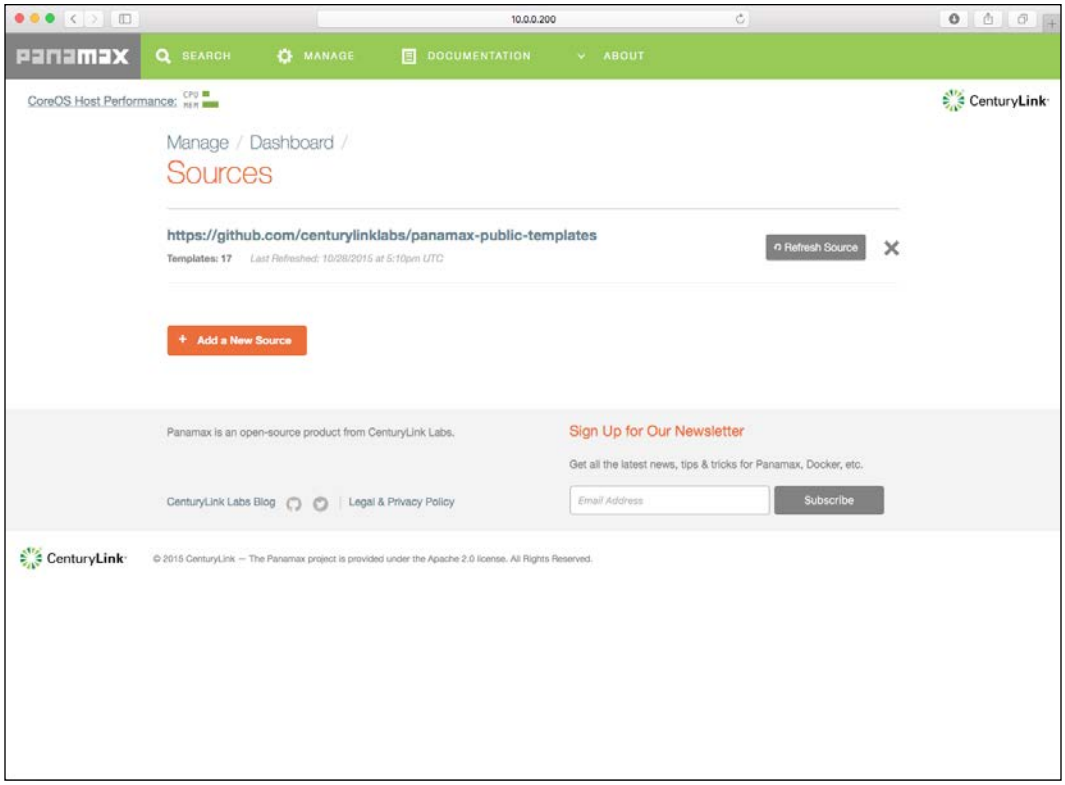
First up is the **Applications** section. Upon entering this one, we can see the application we launched earlier is now in here.



We can see information about this running instance such as where it is deployed to (in this case, locally), the application services that it is running, and the application activity log.

Sources

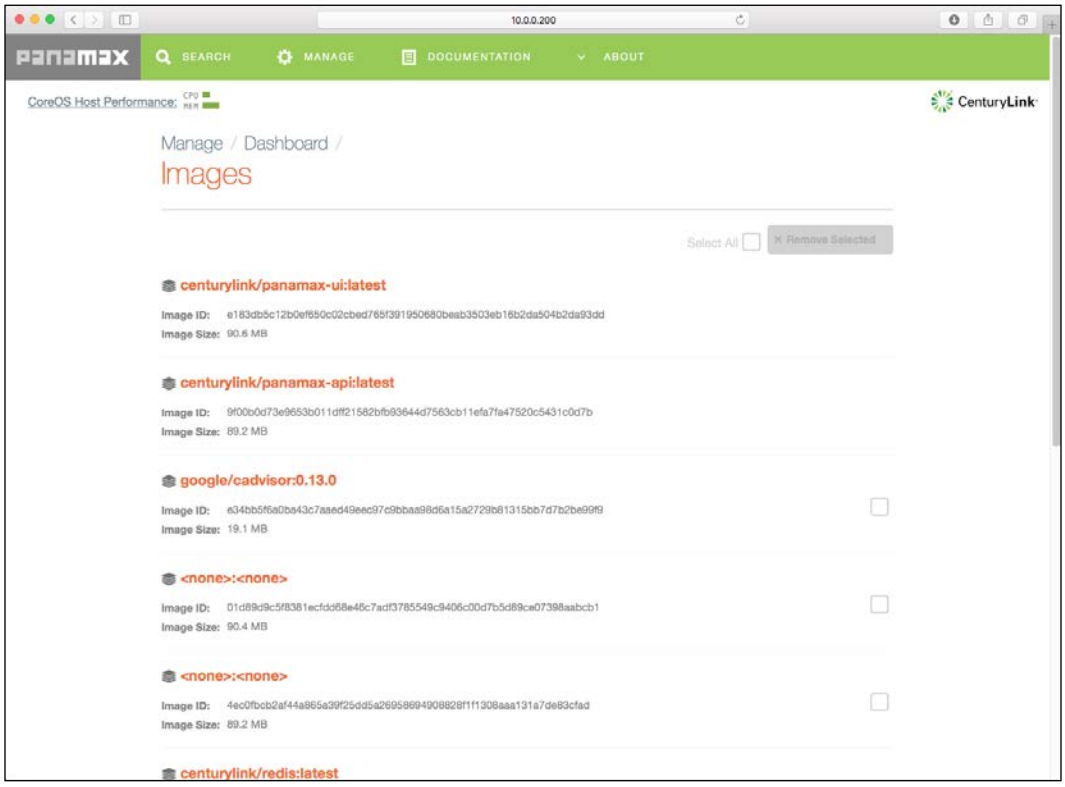
The **Sources** section shows you what resources are currently loaded into the system.



In our case, we can see that the public templates for the Panamax public sources are available. On this screen, you can add additional resources as needed.

Images

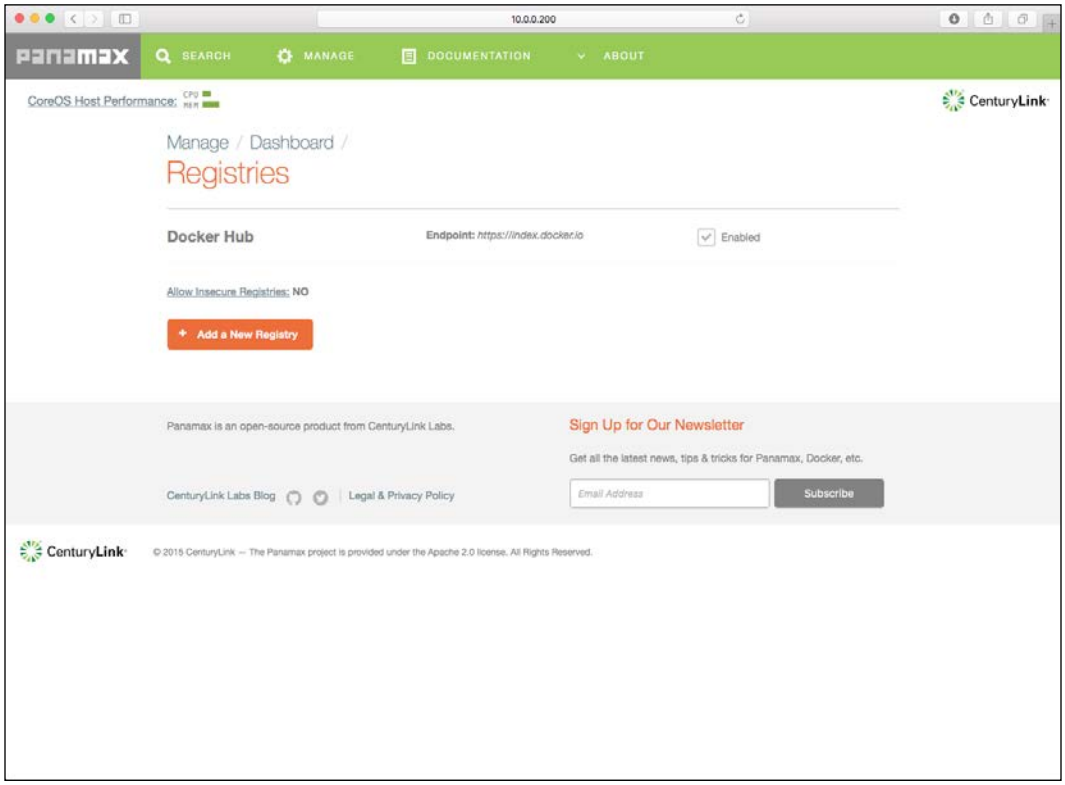
In the next section, the **Images** section, you can see all the images that are currently being used.



Your options on this screen are to remove whatever images you would like to by selecting the checkbox next to them and then selecting the **Remove Selected** button.

Registries

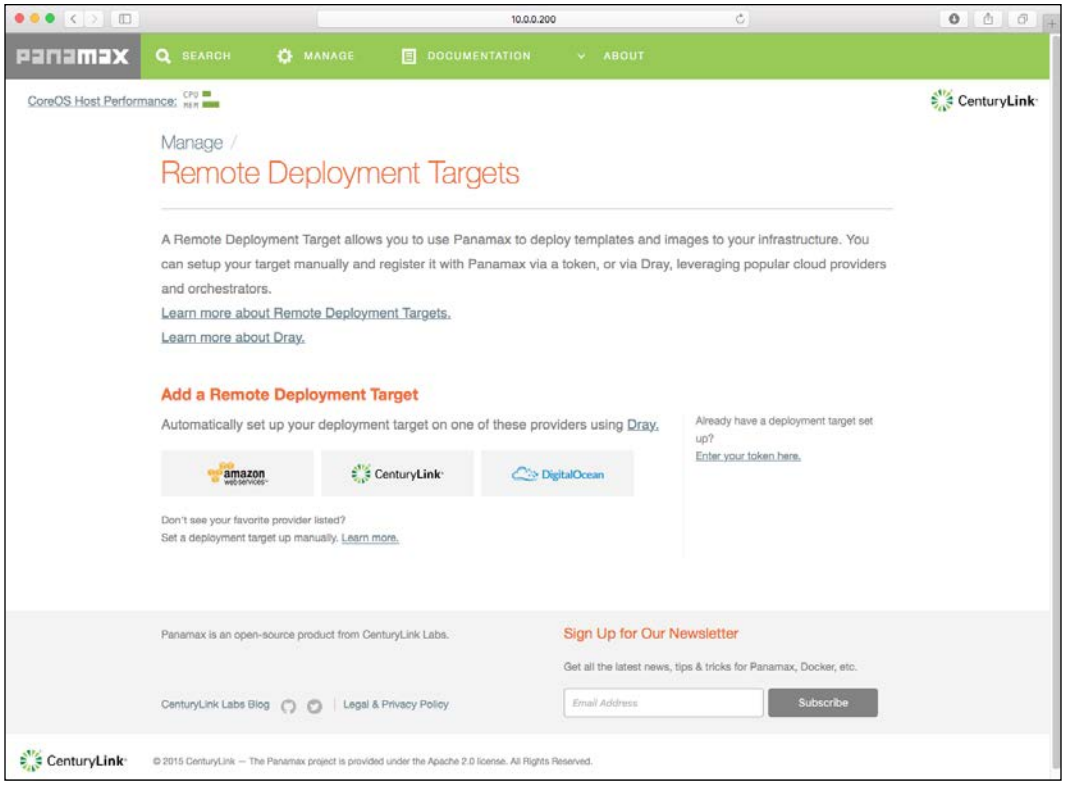
The next section deals with the registries that you can search for templates and images. By default, it searches Docker Hub and includes insecure registries along with secure registries.



You can change that to only search the secure registries if you desire so. You can also add additional registries such as the registries that you may have deployed in your own environment.

Remote Deployment Targets

The last section is **Remote Deployment Targets**.

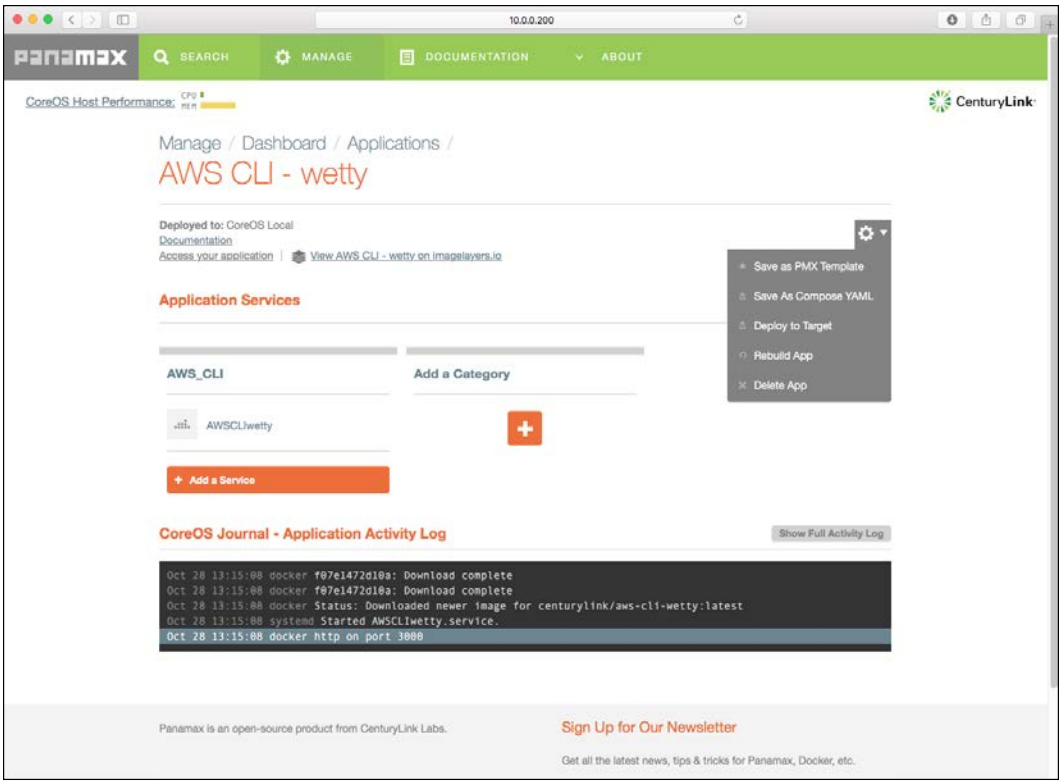


These are items such as cloud hosts that may include AWS, CenturyLink, and DigitalOcean.

Now that we have covered all the sections, let's go back to the application that we deployed and see what all we can do with it.

Back to Applications

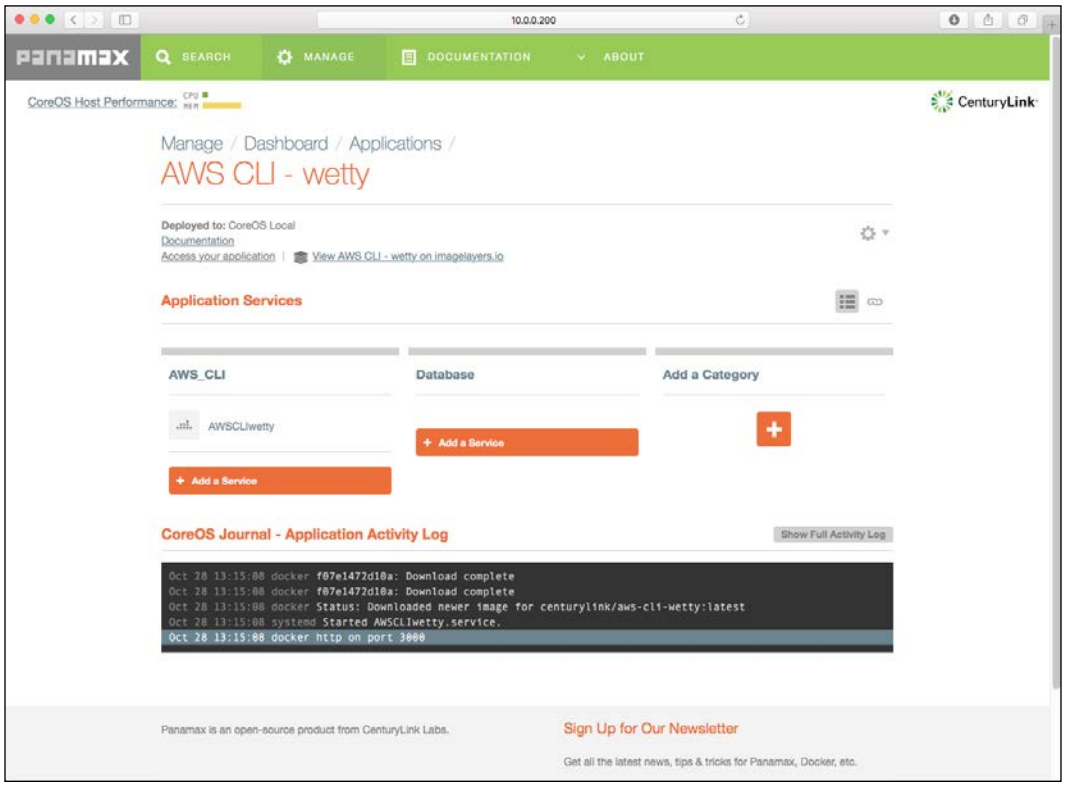
Back in our **Applications** section under the application that we deployed earlier, the `AWS CLI - wetty` image, we can click on the gear icon on the right-hand side of the screen. Given some options such as saving as a PMX template that will allow you to share it with others that are using Panamax, you can also save it as a Compose YAML file that can be used in Docker Compose. Other options include deploying to a target and rebuilding and deleting the app.

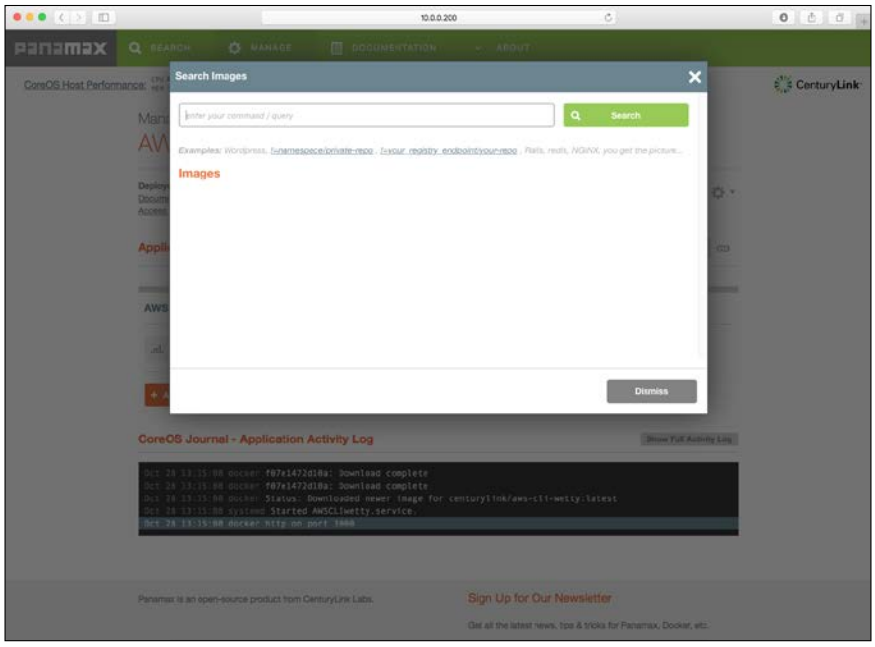


Adding a service

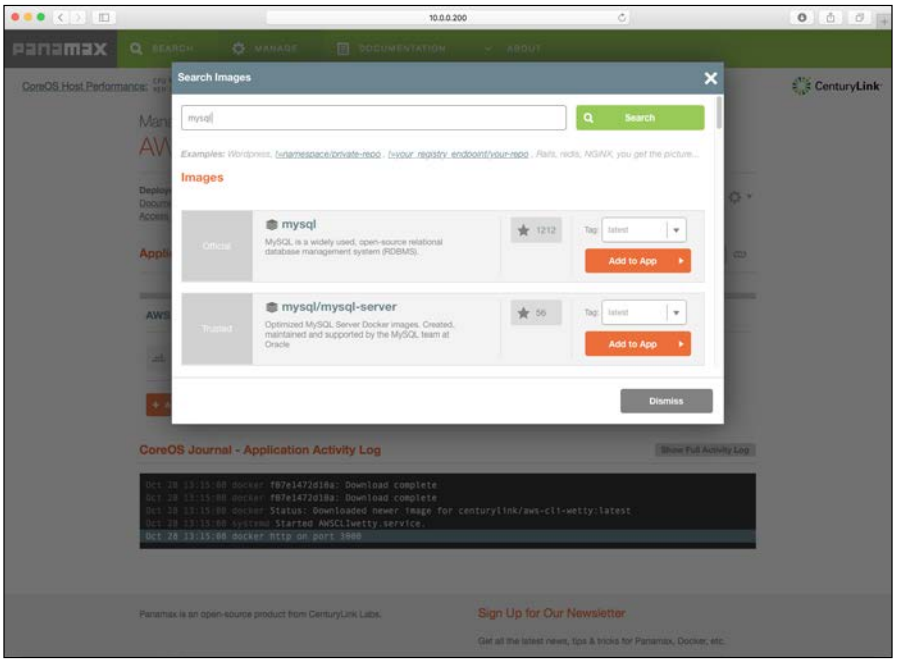
Next, we are going to add a service to our application. To do so, we will click on the **+** button and then give it a name.

In our case, we are going to add a database, so we will name this section `Database`.





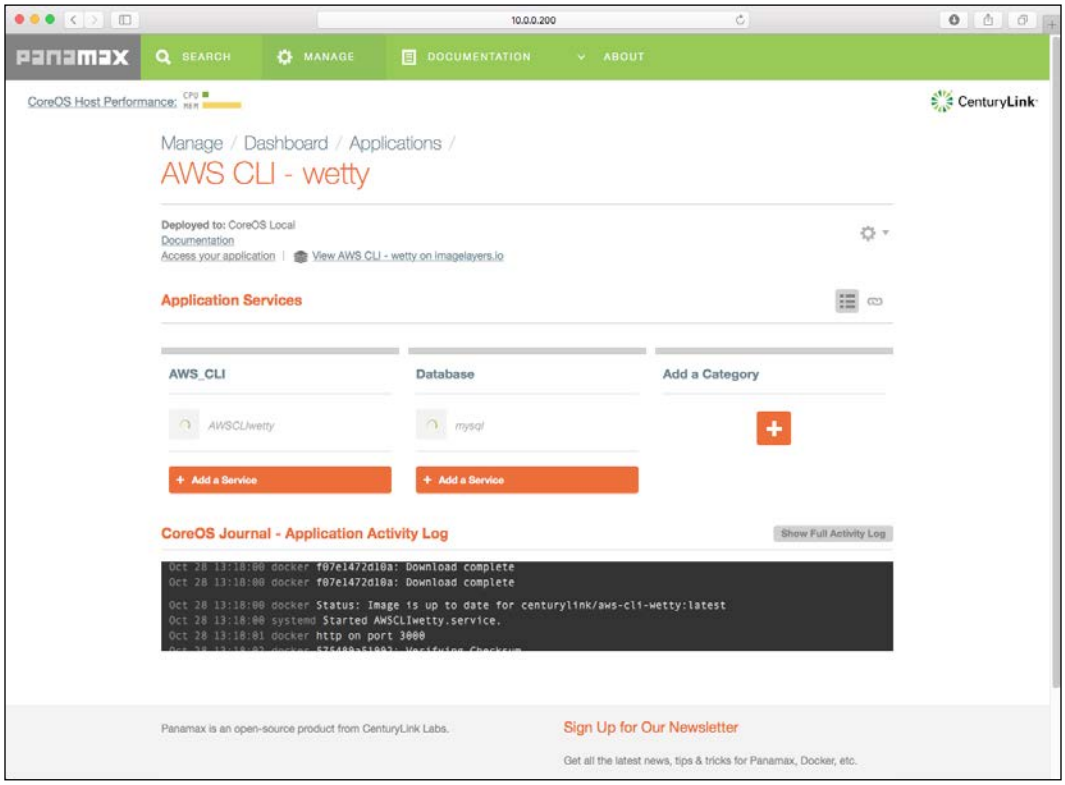
After this, we will click on **+ Add a Service** to the database's application services and will need to search for an image that we want to use.



Since this is a database application and MySQL is known by almost everyone, we will search for it and add it to the app.

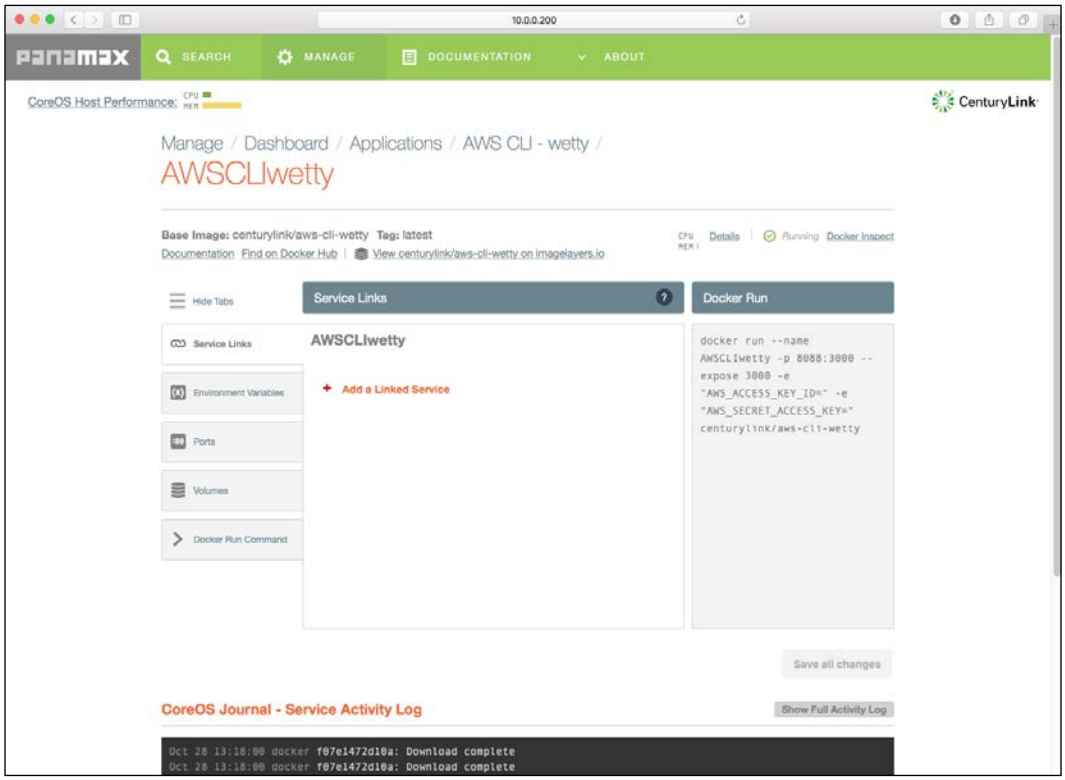
Configuring the application

After we have added it to the app, Panamax will start to configure it for our usage, so we can tie the application services we are running together.



Service links

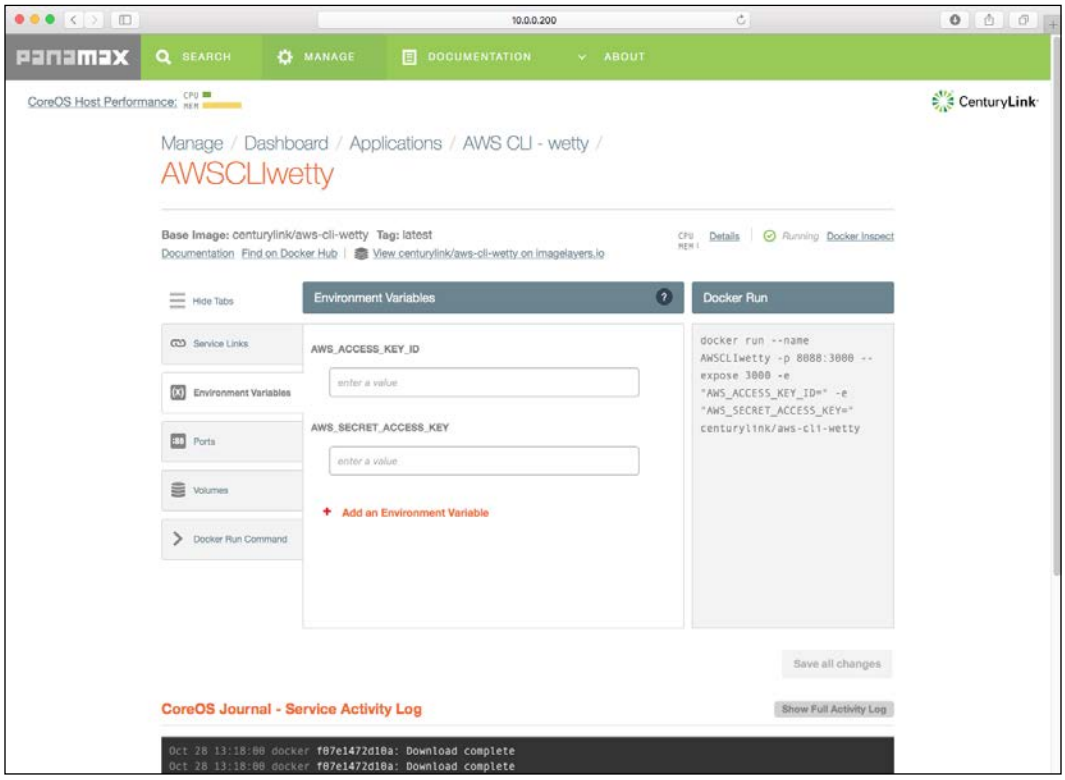
If you want to configure each application service, you can click on it and you will be taken into a submenu.



For this example, we will look at what items we can configure in the `AWSCLIwetty` application. The first item we can configure is the service links. We can also see the `docker run` command that will be used once we populate our environmental variables.

Environmental variables

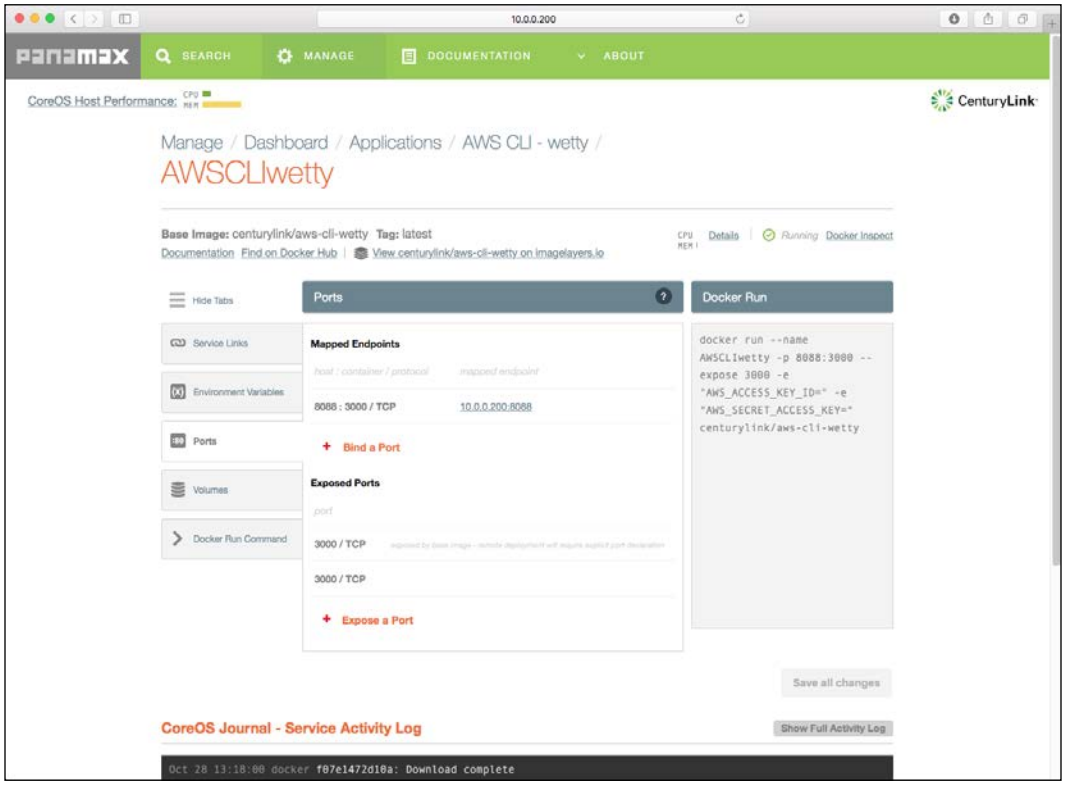
Next are the environmental variables. For this image, it would ask us to supply our AWS access key ID and our AWS secret access key.



These are two items that are required to be able to use the AWS CLI to execute commands against your AWS environment. You can add additional environmental variables too.

Ports

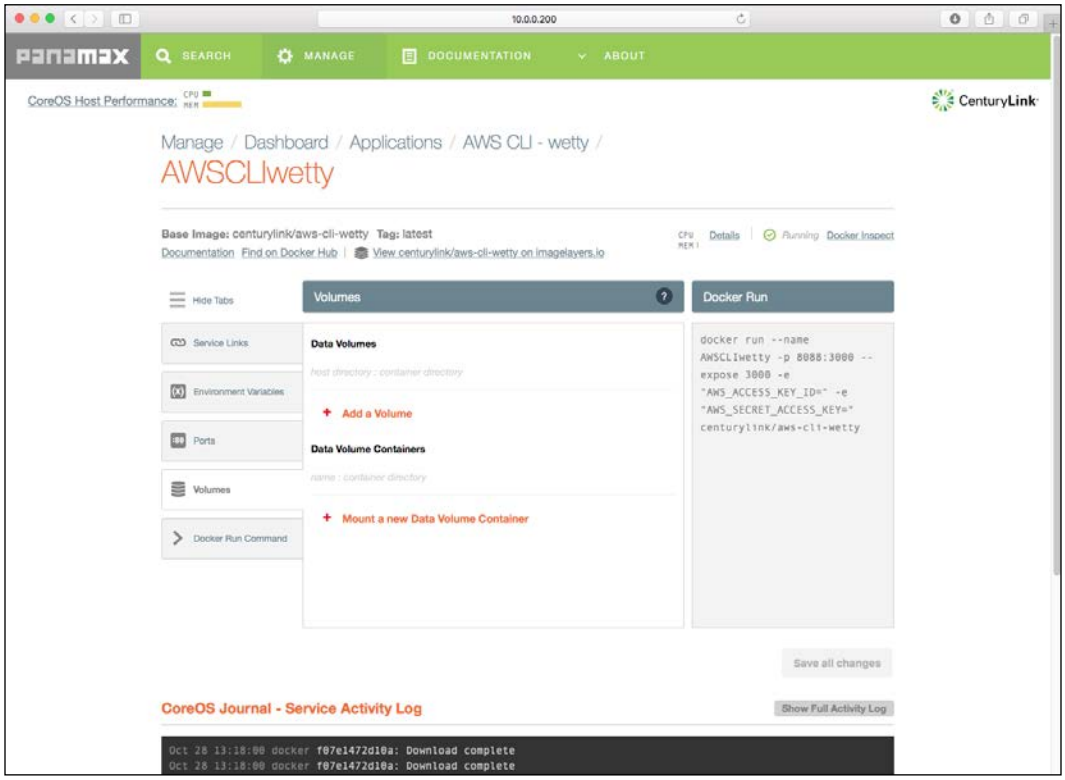
Next, you can view or configure the port configuration that each service uses.



For this service, we can see that it is exposing port `8088` on the host to port `3000` on the container using the TCP protocol. We can see the exposed ports at the bottom and, for this service, it is just port `3000`. We can also add additional ports for each service.

Volumes

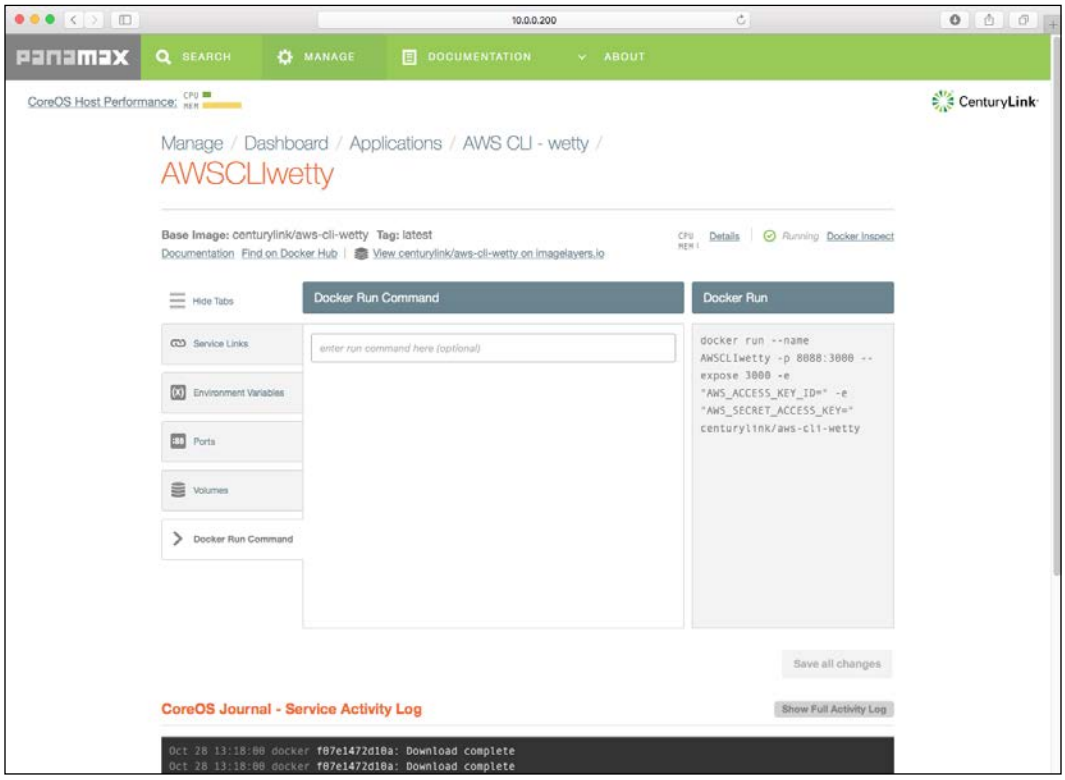
Next, we can see the volume configuration for each service.



This service doesn't utilize any; but if we want to add one, we can do it from this screen. We can remove one if there was one.

Docker Run Command

Last is the **Docker Run Command** section. In this section, you can execute commands against the container that is running the service.



This would be similar to using the `docker exec` command.

Summary

We have now taken a look at two very powerful GUIs that can be used to control your hosts, containers, and images, and they both do very well. If you only had more choices! Well, let's dive into the next chapter and introduce another!

In the next chapter, we will take a look at another GUI tool to manage your Docker hosts, containers, and images, and that is **Tutum**, which was recently purchased by Docker.

12
Tutum

Tutum is a company that was just recently purchased by Docker and has joined its ranks. The goal of Tutum is to help you run your containers on the cloud. Tutum is another feature that makes Docker easy to use.

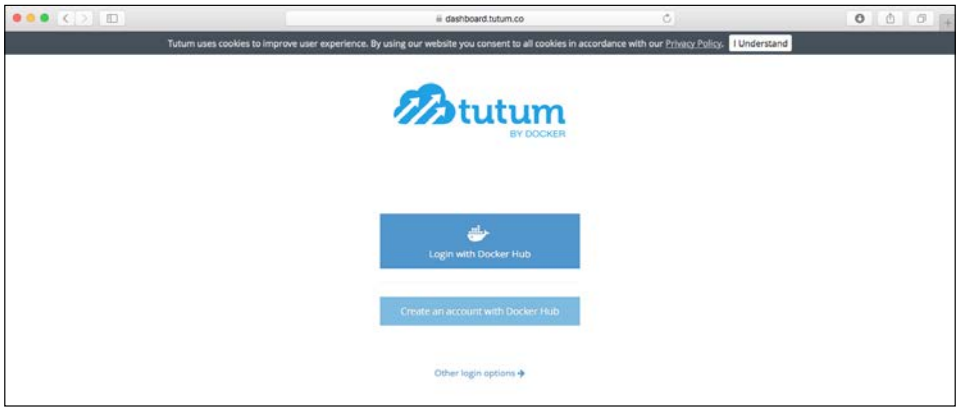
In this chapter, we will cover how to:

- Start with Tutum
- Add your node
- Create a stack

Getting started

You will see a screen similar to the following screenshot when you access the Tutum website at `https://www.tutum.co`.

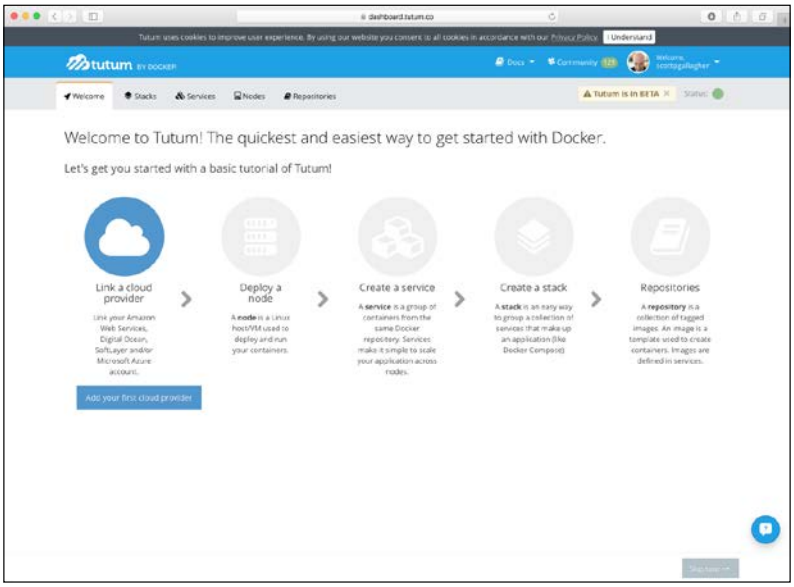
Upon clicking **Get started for free!** or the **Login** link, you will be presented with the following screen:



Now, given that Docker has recently scooped them up, this could change in the future. But you will be presented with a login screen to use your Docker Hub, current Tutum, or GitHub credentials.

The tutorial page

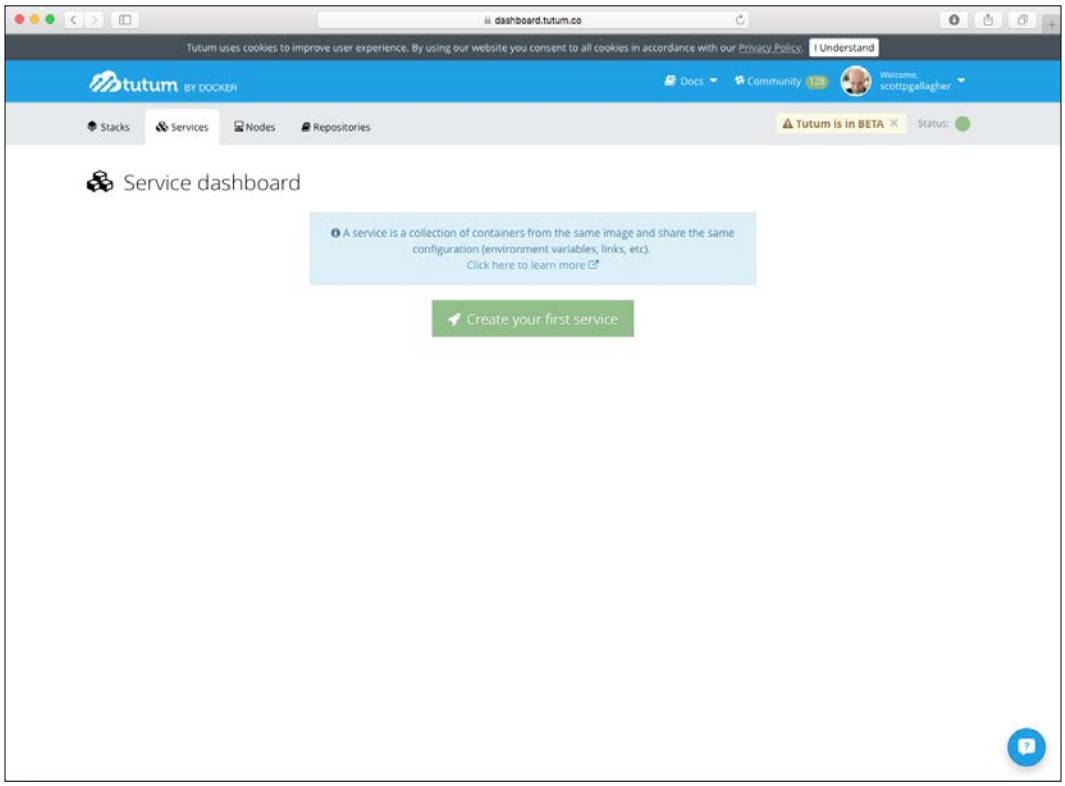
You will be presented with the tutorial page that will provide a tour of Tutum if you wish.



You can also skip the tour by clicking on the button in the bottom-right corner of the screen, which we will do to get you started.

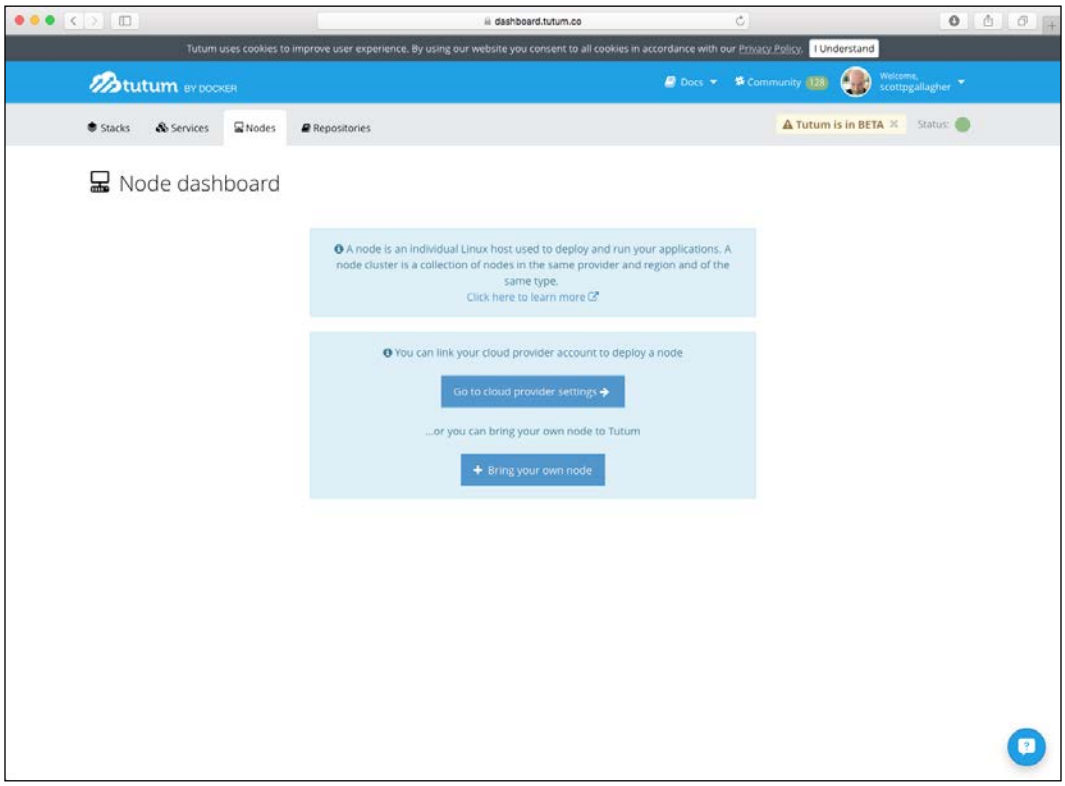
The Service dashboard

You will be taken to **Service dashboard**, where you can create your first service. But before we do that, we need to do some other work. So, let's get our nodes added first.



The Nodes section

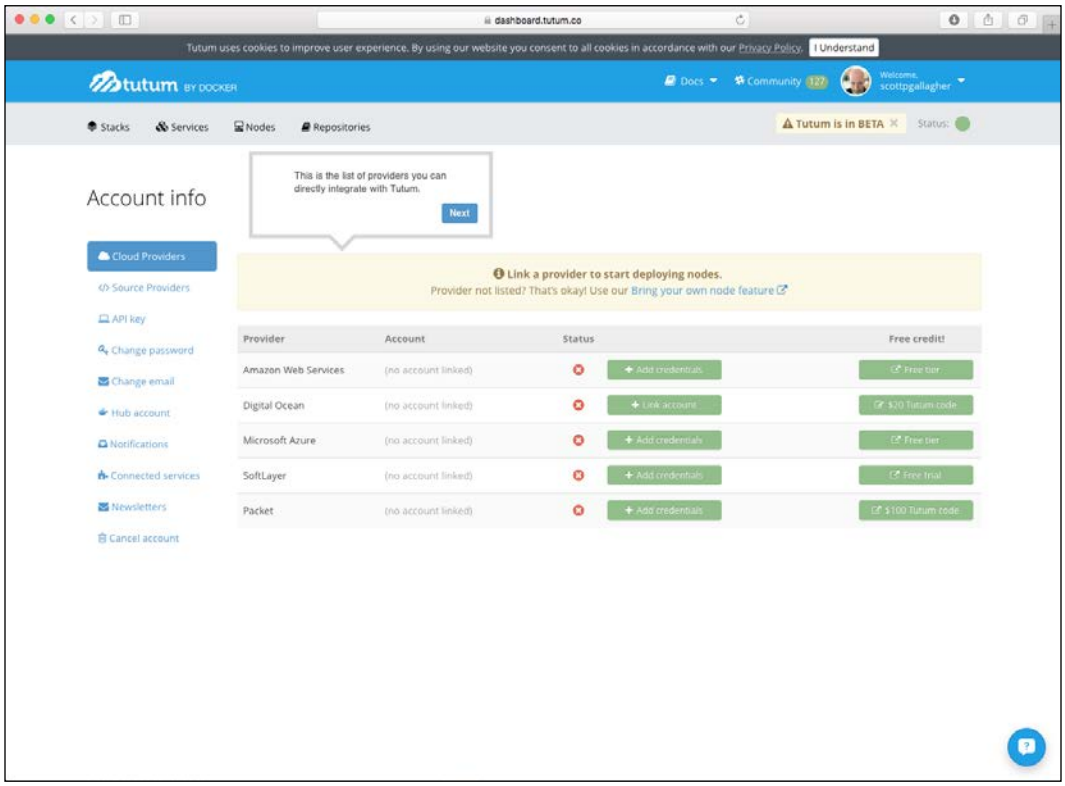
If you click on the **Nodes** section in the navigation bar, you can start adding your cloud provider or you can bring your own node.



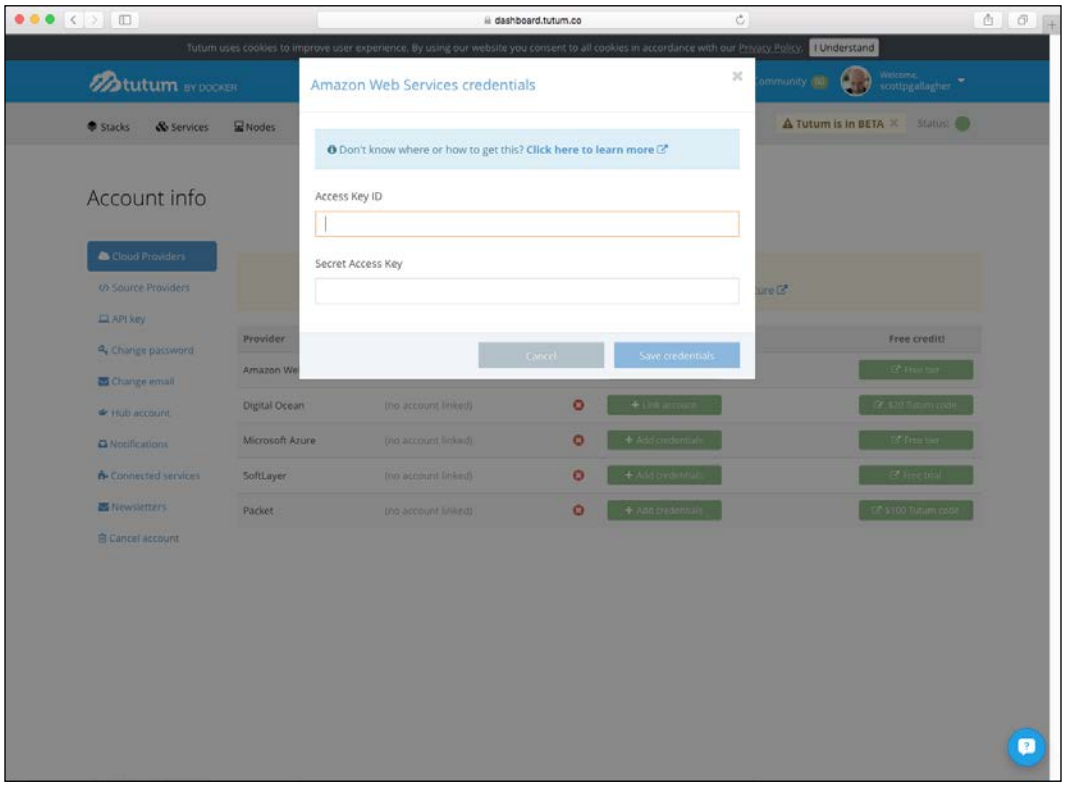
If you wish to bring your own node, you will need to install a client that Tutum uses to communicate with your node. For this example, we are going to stick with using a cloud provider: AWS in this case.

Cloud Providers

In the **Cloud Providers** section, you will get a list of cloud services that you can link to. Again, we are going to use AWS. But you could use DigitalOcean, Microsoft Azure, SoftLayer, or Packet. We will click on the **+ Add credentials** button for AWS:

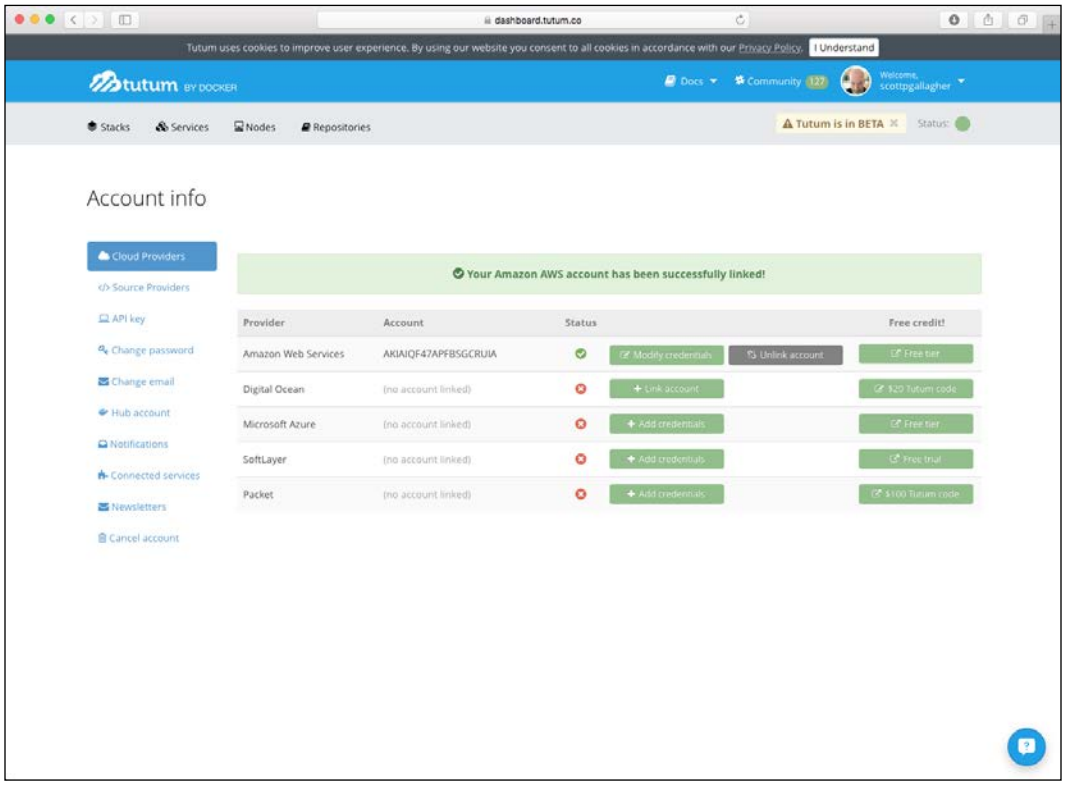


Here we would provide our AWS Access Key ID as well as our Secret Access Key:

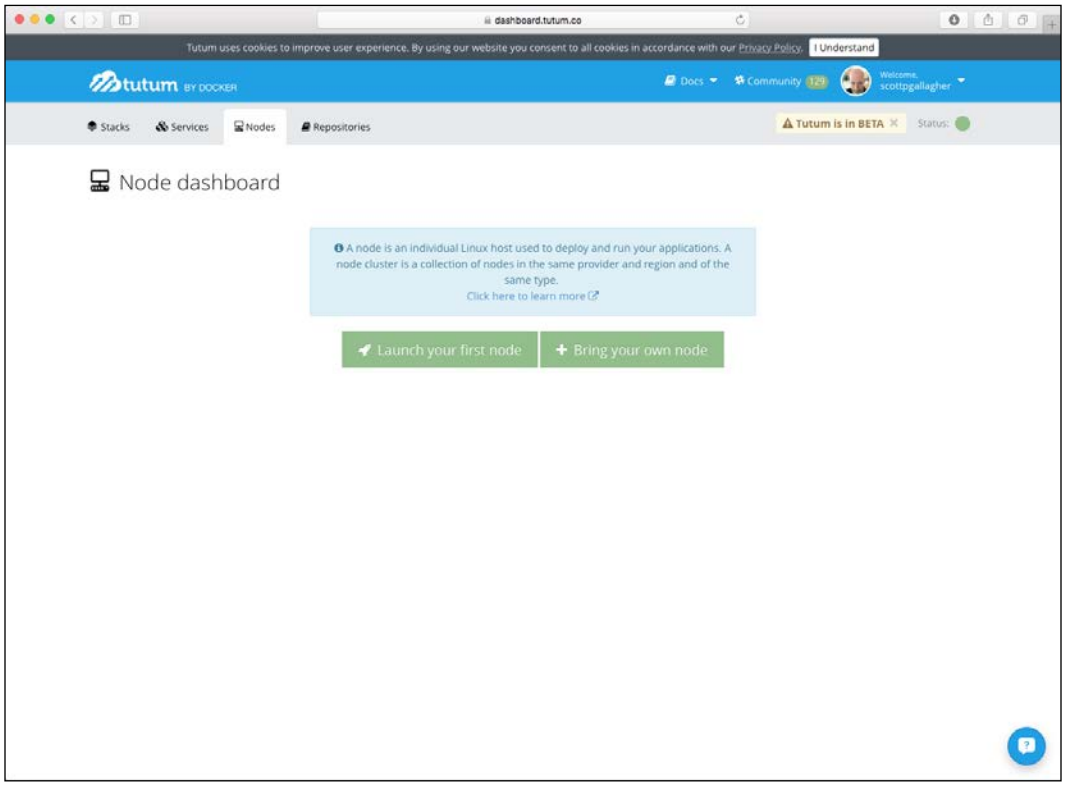


AWS uses your access key ID as well as your secret access key to authenticate against AWS. You can enter these details and then click on the **Save credentials** button.

You will then see that you have linked your AWS account, can modify the credentials if they ever change, or unlink the account all together if you need to.



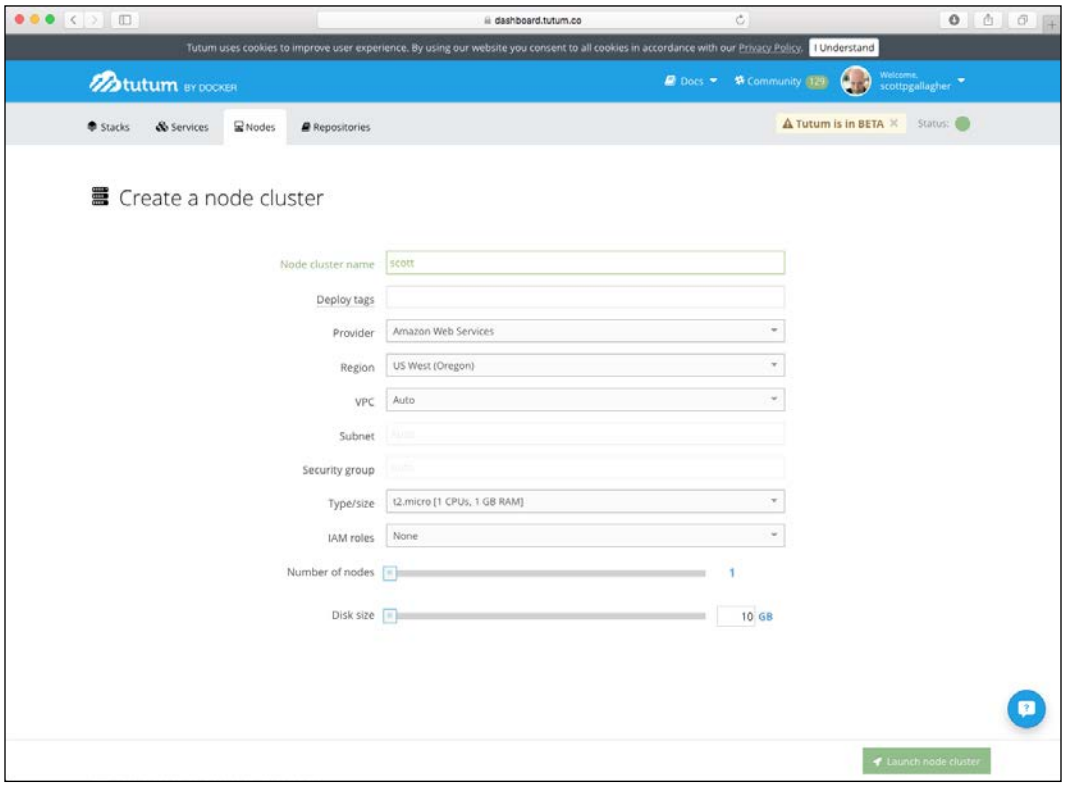
Now that we have a cloud provider to run our service on, we can launch our first node on the cloud now by clicking on the **Launch your first node** button:



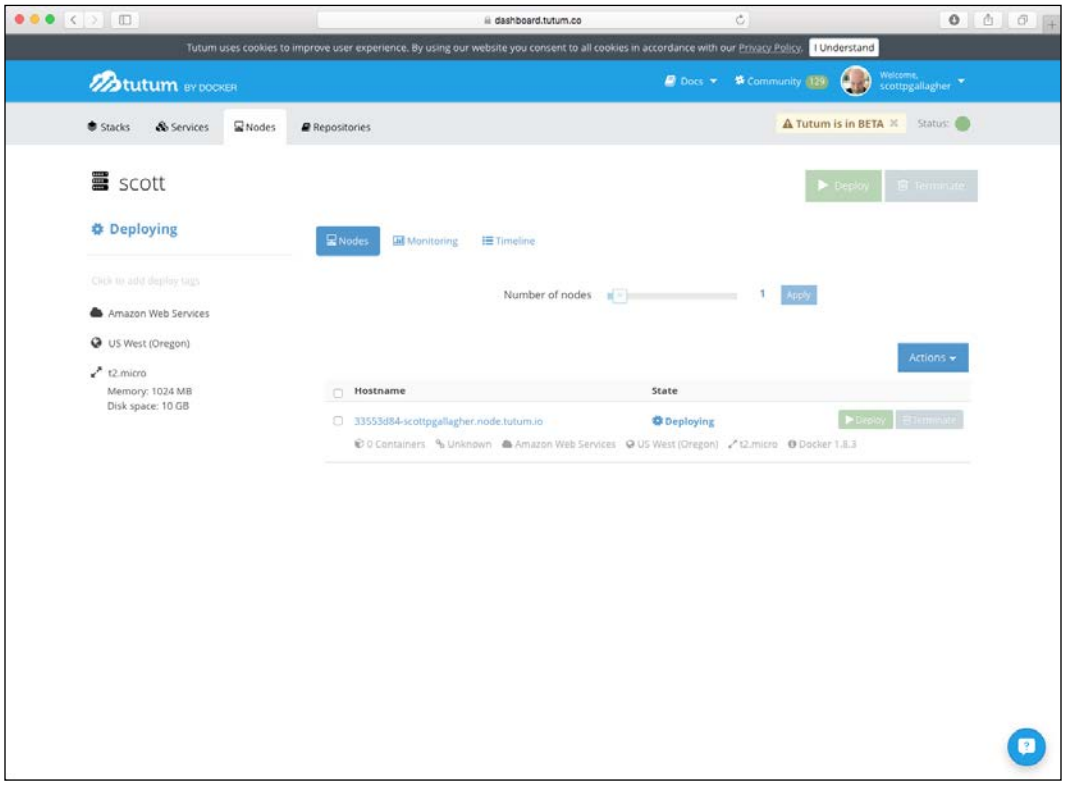
We will navigate back to the **Nodes** screen.

Back to Nodes

After clicking on **Launch your first node**, we will need to provide some additional information such as what region we want to deploy our node to, if we have a custom VPC we have created that we want to deploy our node to, what size we want the node to be, any IAM roles we want to assign to the node, the number of nodes we want, and the disk size of each node.

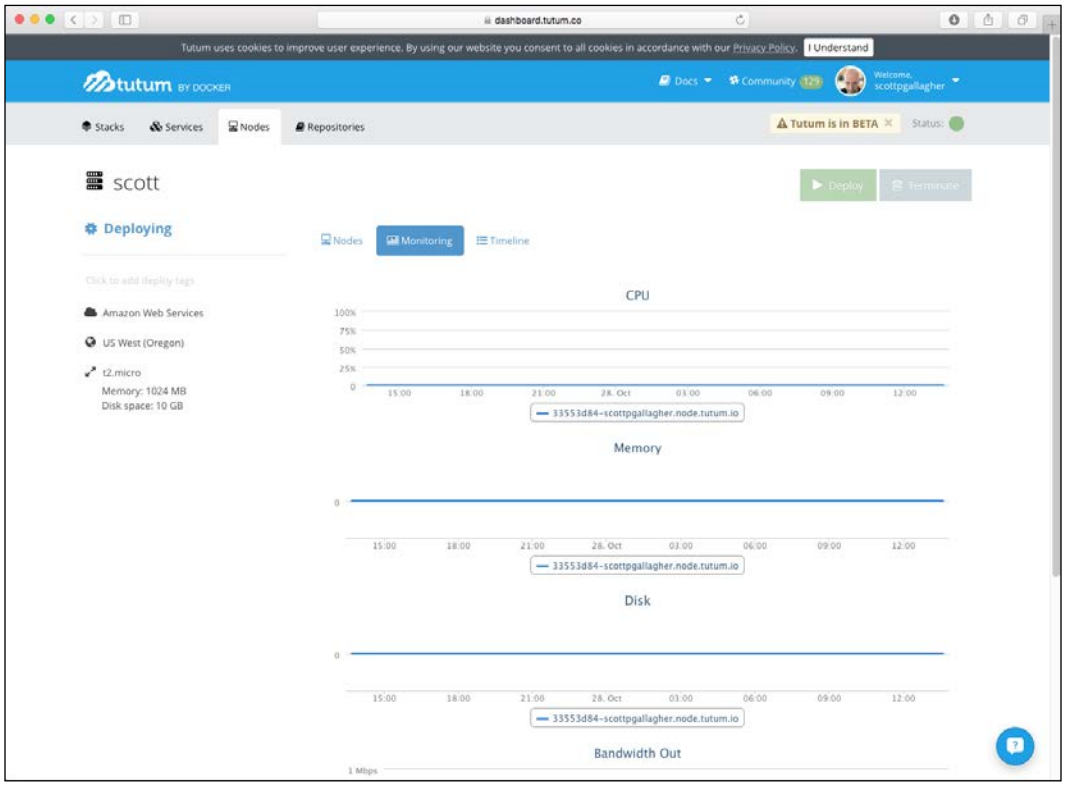


For our example, we mainly kept the default, only lowering the disk size to the minimum size of 10 GB.

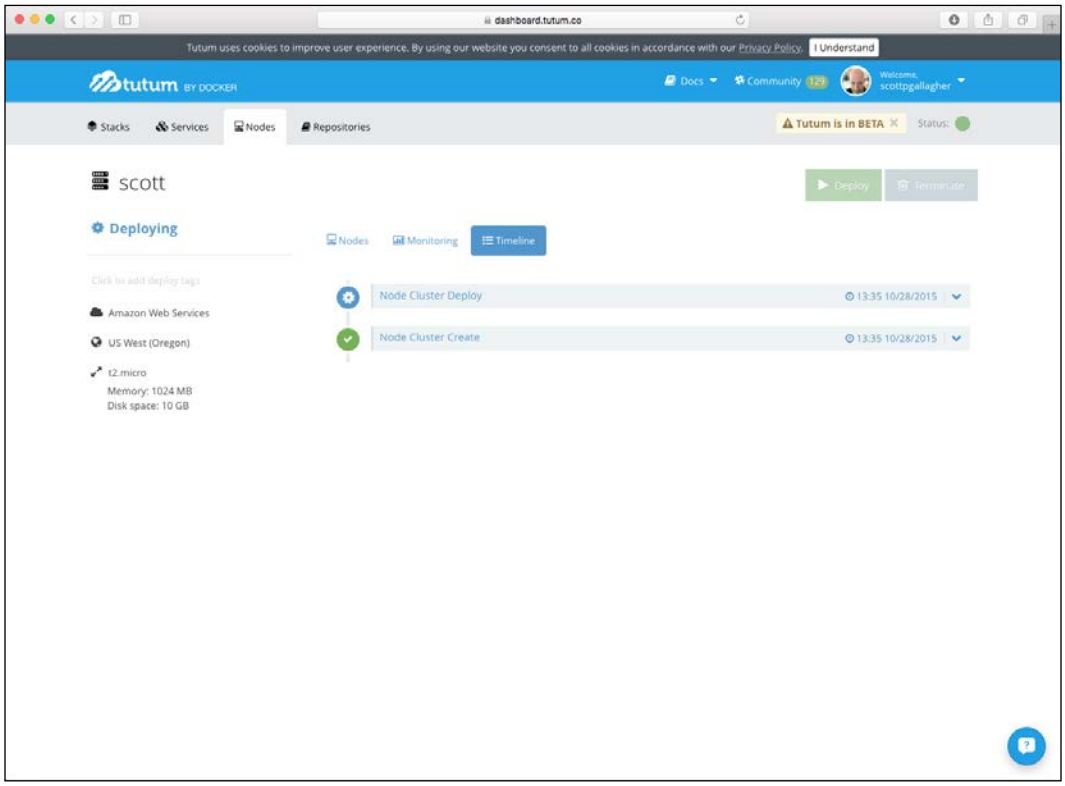


Once you have clicked on the **Launch node cluster** button, you will see the status of the node; in this case, it's **Deploying**. We also have some other items we can check out while it's deploying.

We can view the **Monitoring** tab and see information pertaining to the node such as **CPU**, **Memory**, **Disk**, and **Bandwidth Out**.

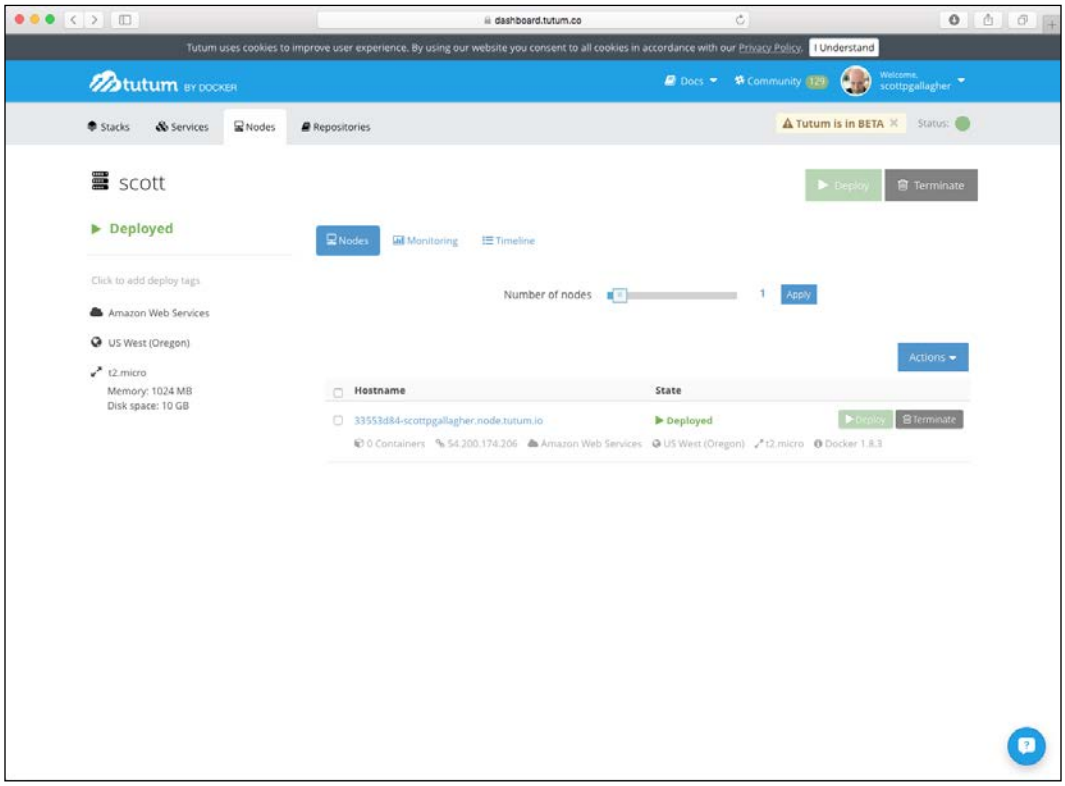


We can also view the timeline of our node. Now, at first, this will be very short as it's just showing us that we created the node and are deploying it.



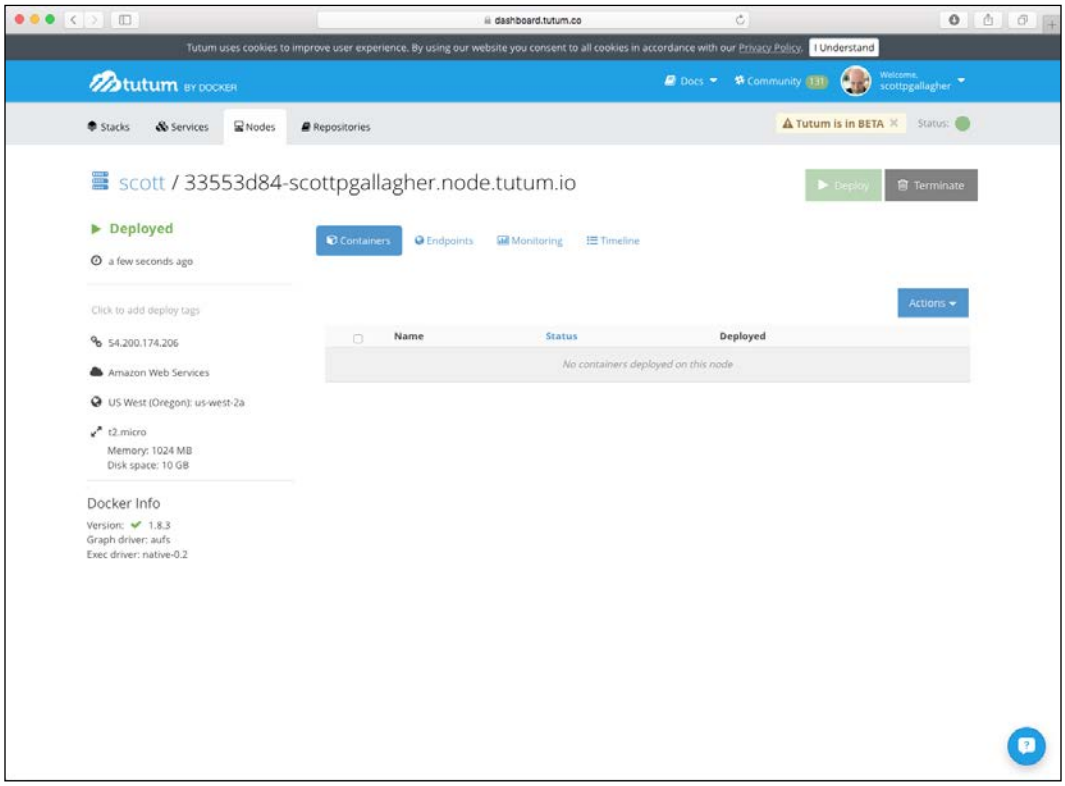
Over time, this timeline will grow and show you the progress of your node.

Our node should be deployed by now. So, we can click back on the **Nodes** link and see that it has in fact been deployed and is running.



We can get some information on the left-hand side, such as it is running on AWS in the US West (Oregon) region, and is a **t2.micro** instance with 1 GB of memory and 10 GB of disk space. We can also see that it currently has no containers running on this particular node, what IP address has been assigned, and what version of Docker it is running. We can terminate our node as well when we no longer need it or scale the number of nodes with the slider at the top if we want to increase the number of nodes.

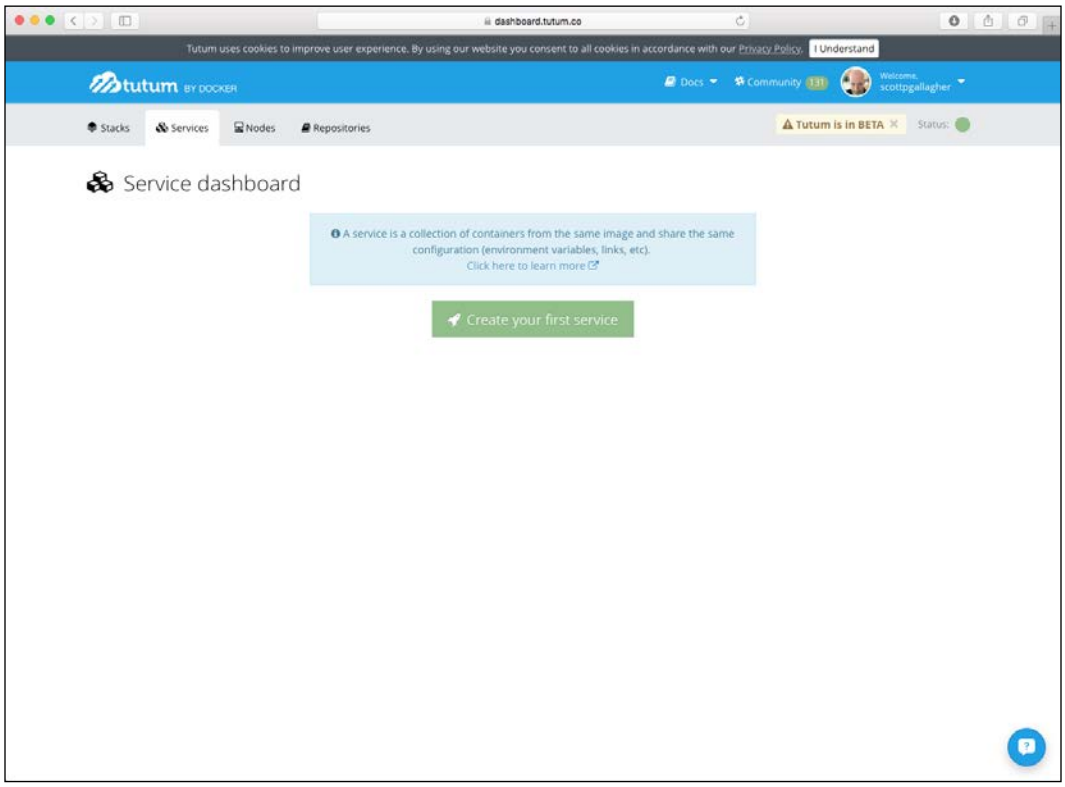
If we drill down into the node itself by clicking on its hostname, we can see some more information provided to us.



It includes what, if any, containers are running on this node, what endpoints or ports are exposed, the monitoring of the node (as we saw earlier), as well as the timeline that we saw before. Now, all of this pertains to the node itself, not the containers that will be running on the node.

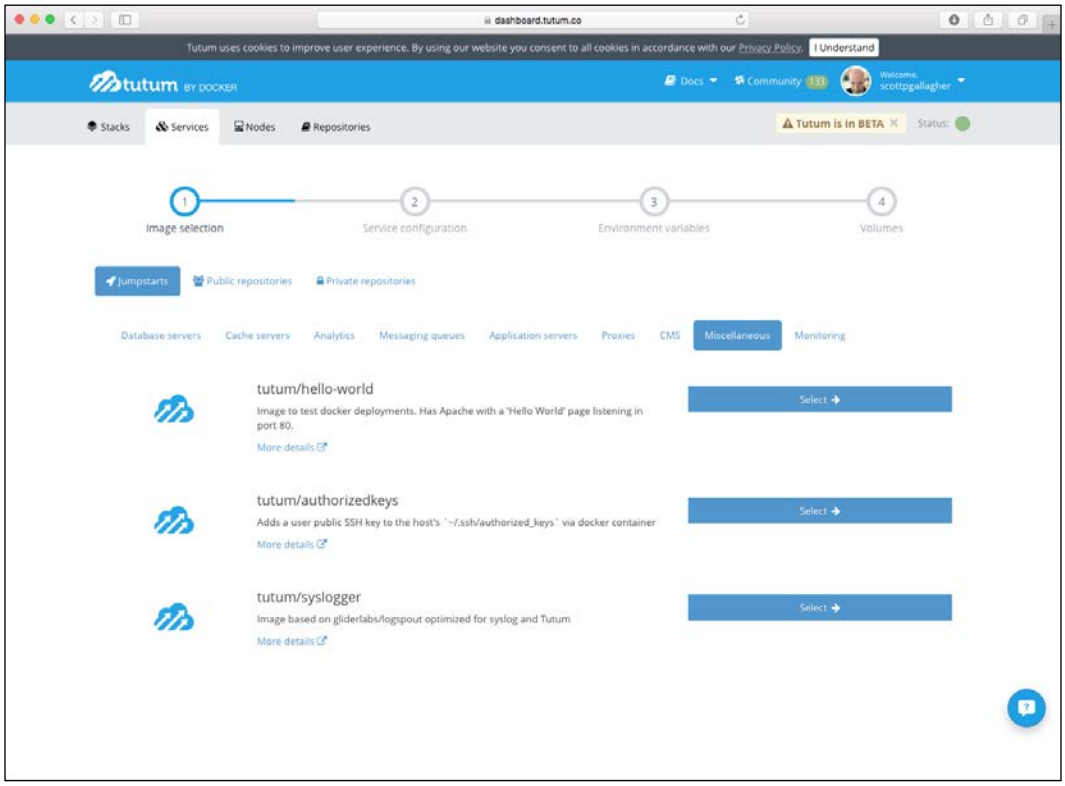
Back to the Services section

Now, it's time for us to launch a service and get some containers running on this node.

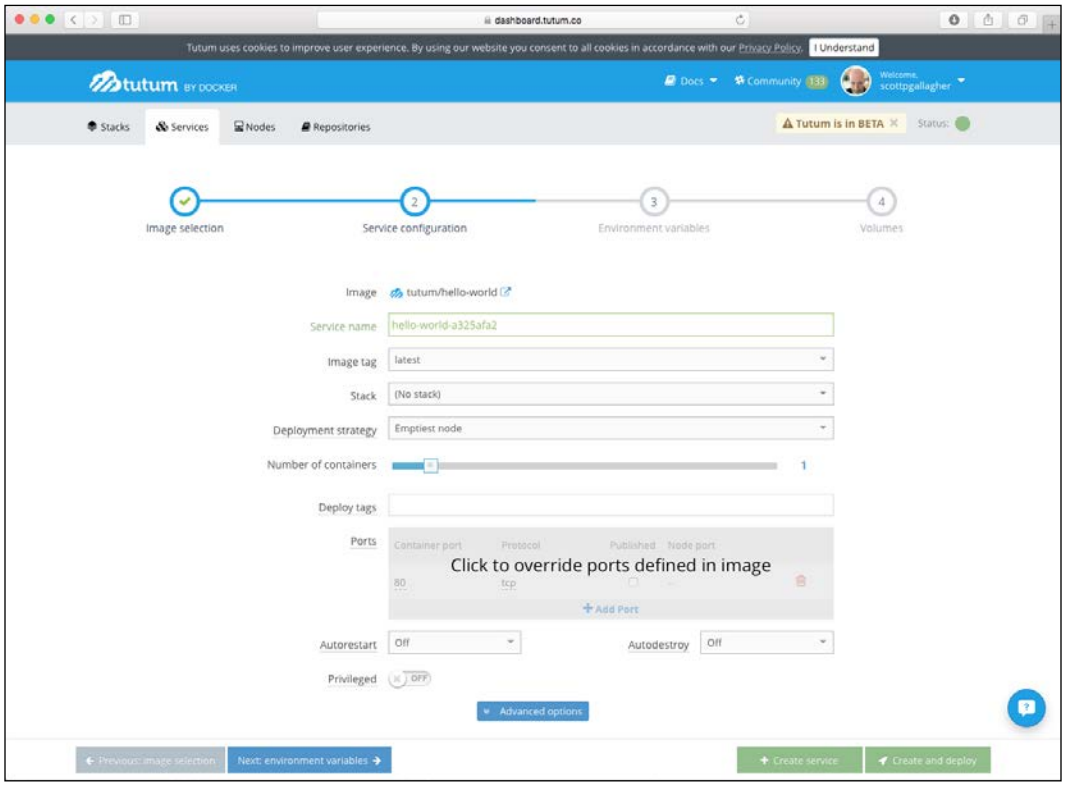


By clicking on the **Services** tab, we will be taken to the previous screen, where we can deploy a service.

Now, Tutum offers up three areas to search for the images you might want to use: jumpstarts or collections that they have categorized for you; public repositories on Docker Hub; or private repositories that you have set as private on your Docker Hub account. For our example, we are going to select the `tutum/hello-world` example due to its small size.

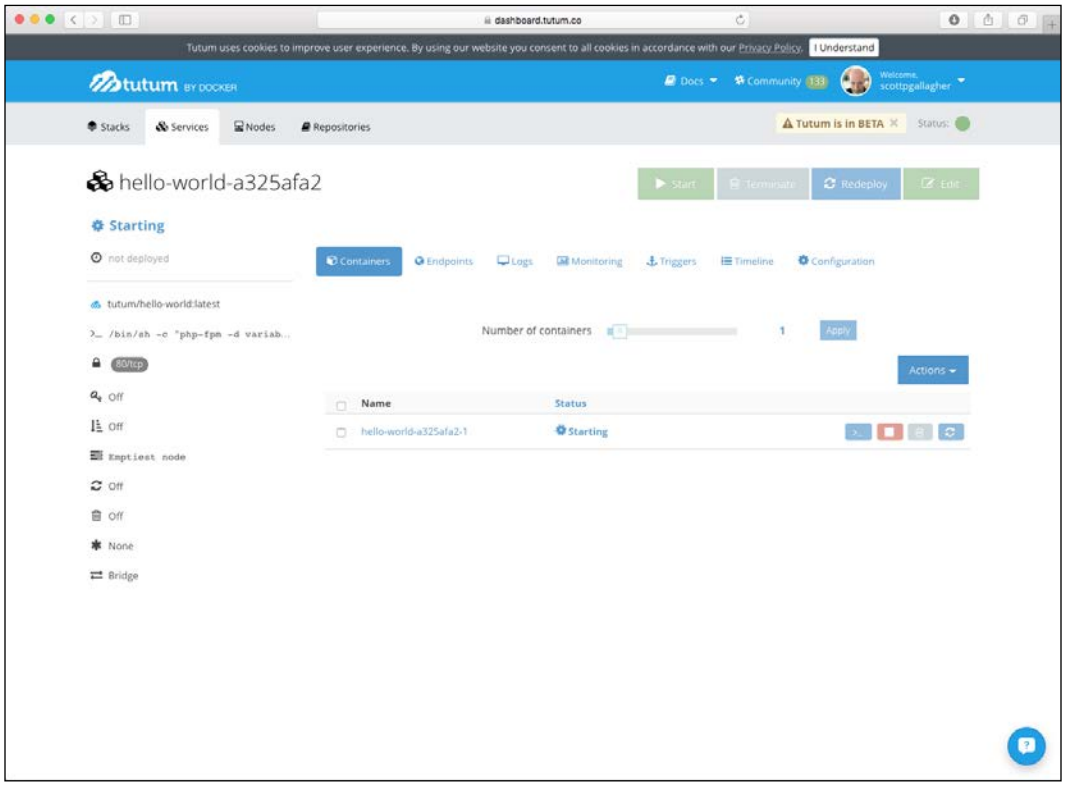


After clicking the **Select** button for it, we are taken to a screen similar to the following one; yours will vary depending upon what image you have selected.



Now, you can give the service a name or use the generated one for you. You can also select what tag to use for the image, what your deployment strategy is (if you are using multiple nodes), how many containers to deploy, any tags you wish to add to the containers that will be deployed, custom port settings (if any), and whether it should autorestart in the event of a failure. This should seem familiar as some of these items, such as deployment strategy, were covered in the book, mainly in *Chapter 8*, *Docker Swarm*, with regards to Docker Swarm. So, once you have everything kosher, go ahead and click on the **Create and deploy** button and prepare for a blast off!

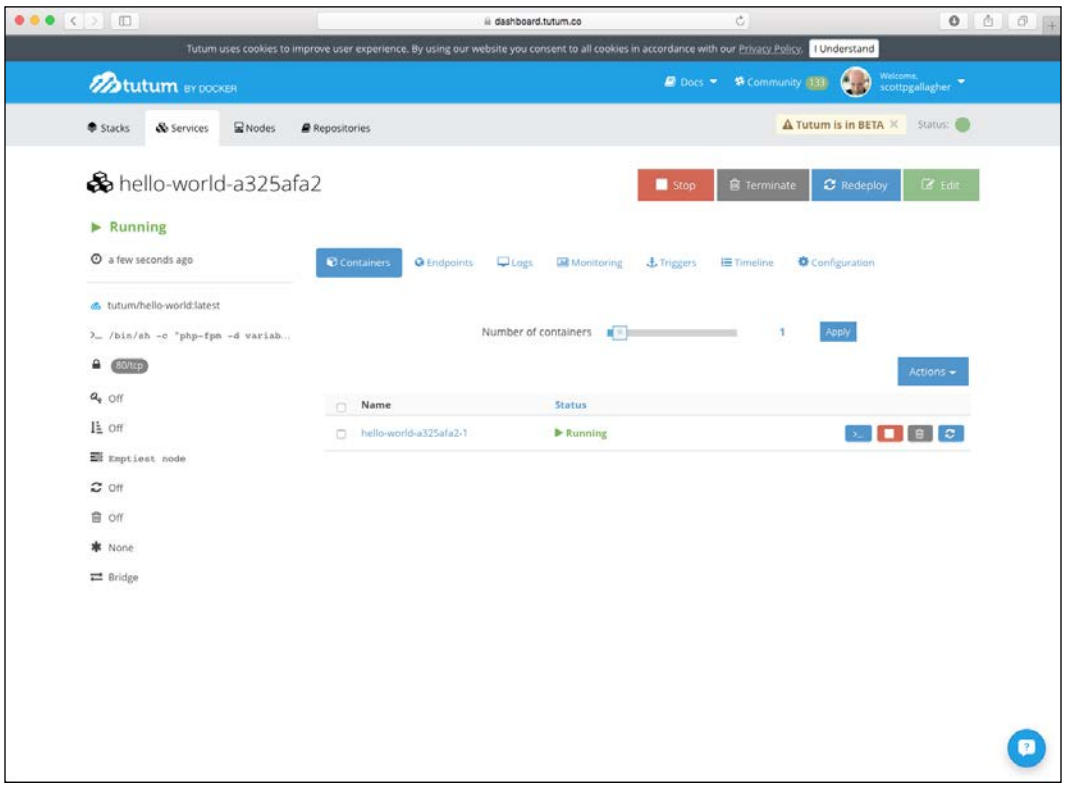
After we click on the button, we are taken to a screen similar to the one we saw when we were deploying our host node.



We can see information on the left-hand side, such as what command the container is running, what ports are exposed, and other settings as well pertaining to the container. We can see that it's in the **Starting** state and should be running shortly.

Containers

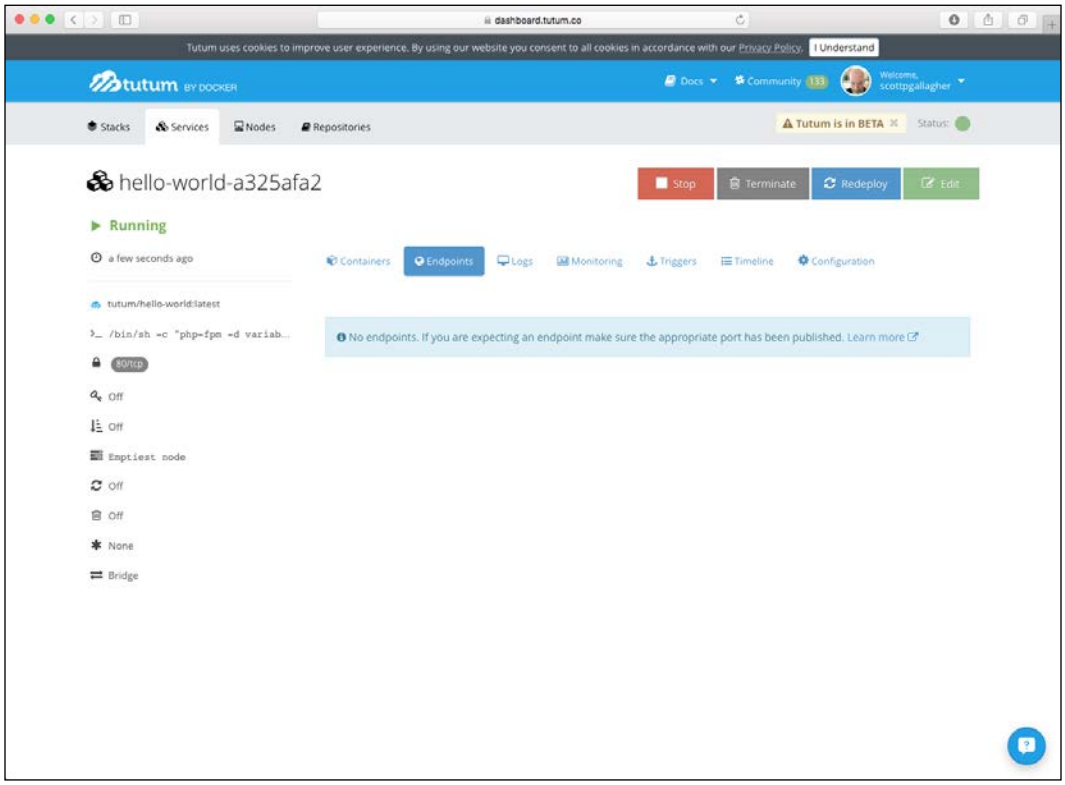
Once it has finished starting and is now in the running state, we can manipulate the container and do things such as stop, terminate, redeploy, or even edit the configuration of the container, and expand the number of containers that are running.



Now, let's take a look at the navigation menu for containers.

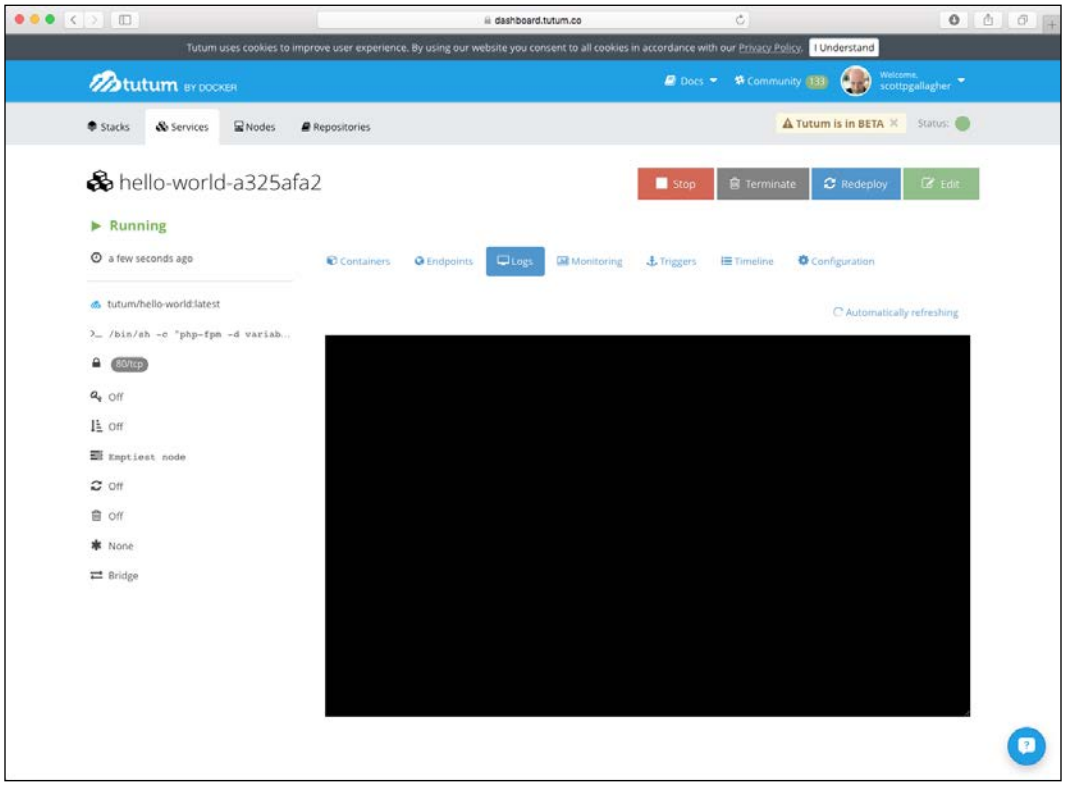
Endpoints

Again, the **Endpoints** screenshot will show us any port information pertaining to the running container.



Logs

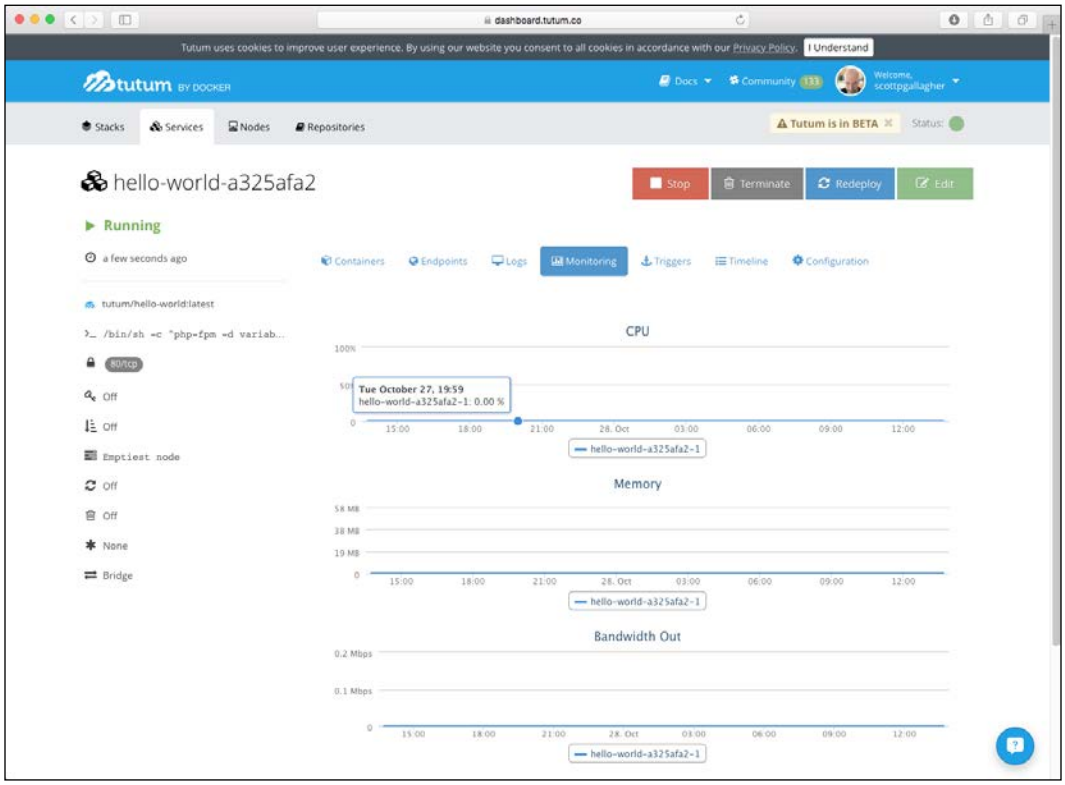
The **Logs** section will show us a running log of the screen output the container would have.



Since this container just started, we don't have anything yet; but this section can be helpful in the event you need to troubleshoot a running container.

Monitoring

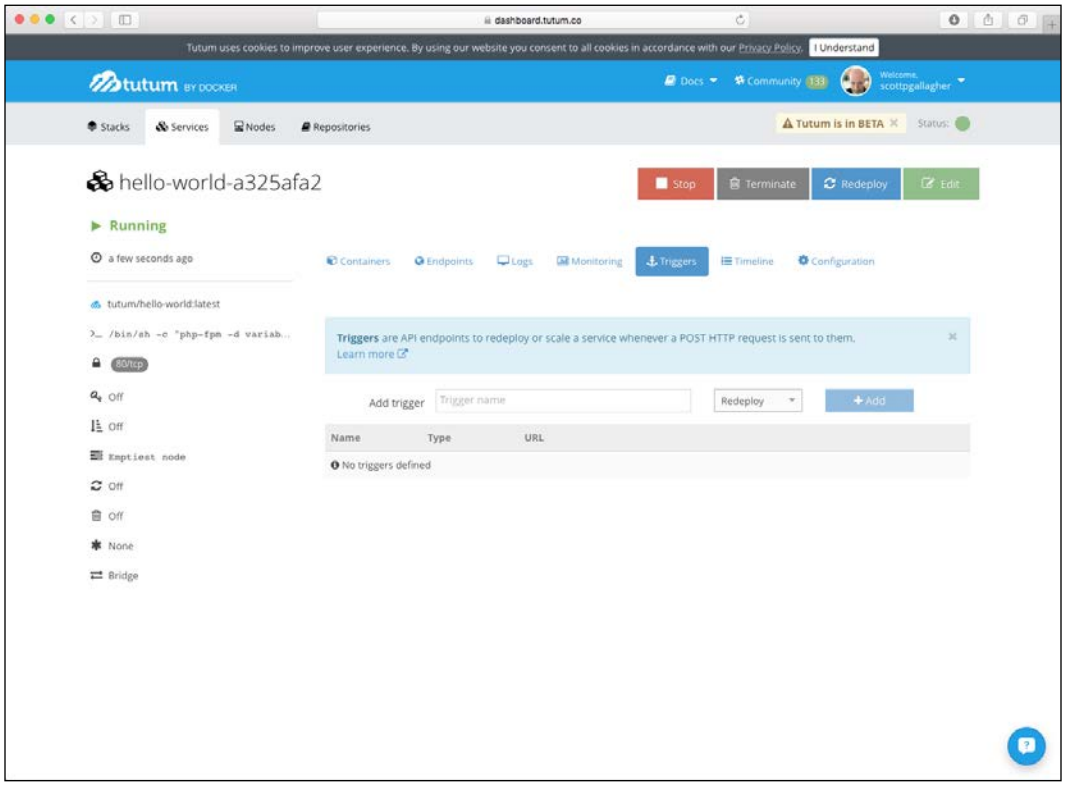
Next, we have the monitoring section that can show us the information we saw before in the **Nodes** section.



Items such as **CPU**, **Memory**, and **Bandwidth Out** can tell how much our container is being used for the service that it is running.

Triggers

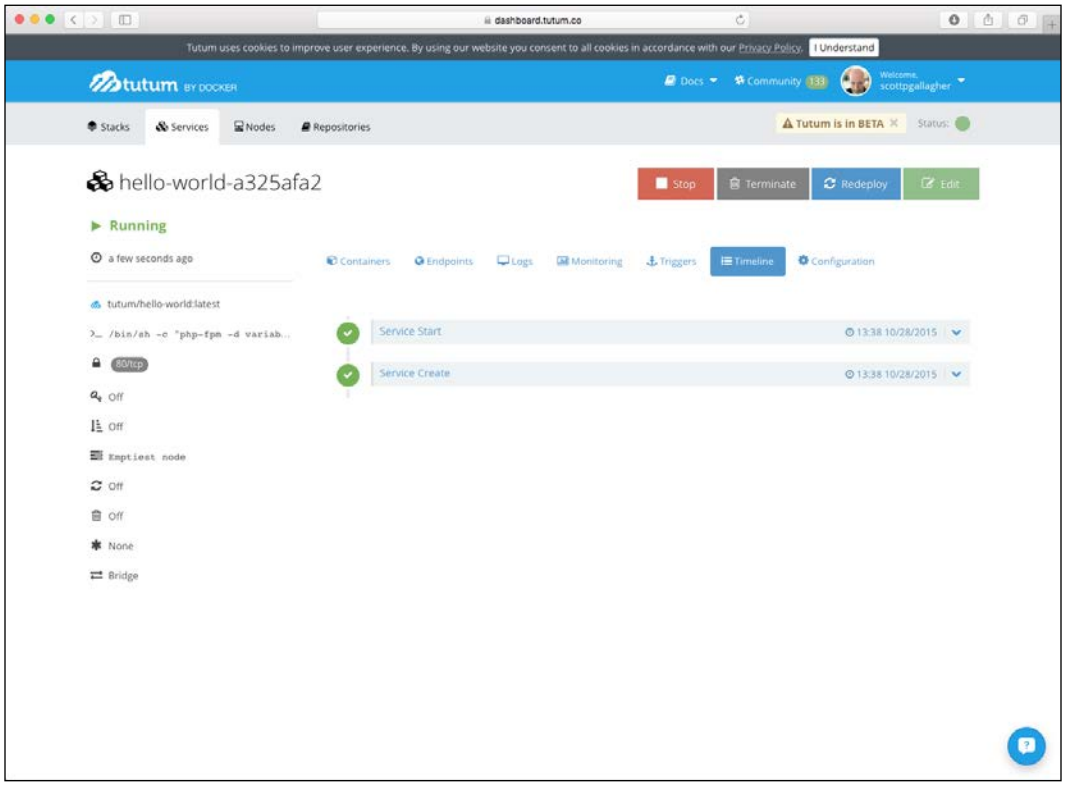
Next up is the **Triggers** section. Now, this section can come in handy if you are looking at scaling something based on the CPU usage that a container has.



For example, you could set a trigger that if the CPU usage goes above 60%, launch another container to help with the load (assuming you are running your service in a load balancer).

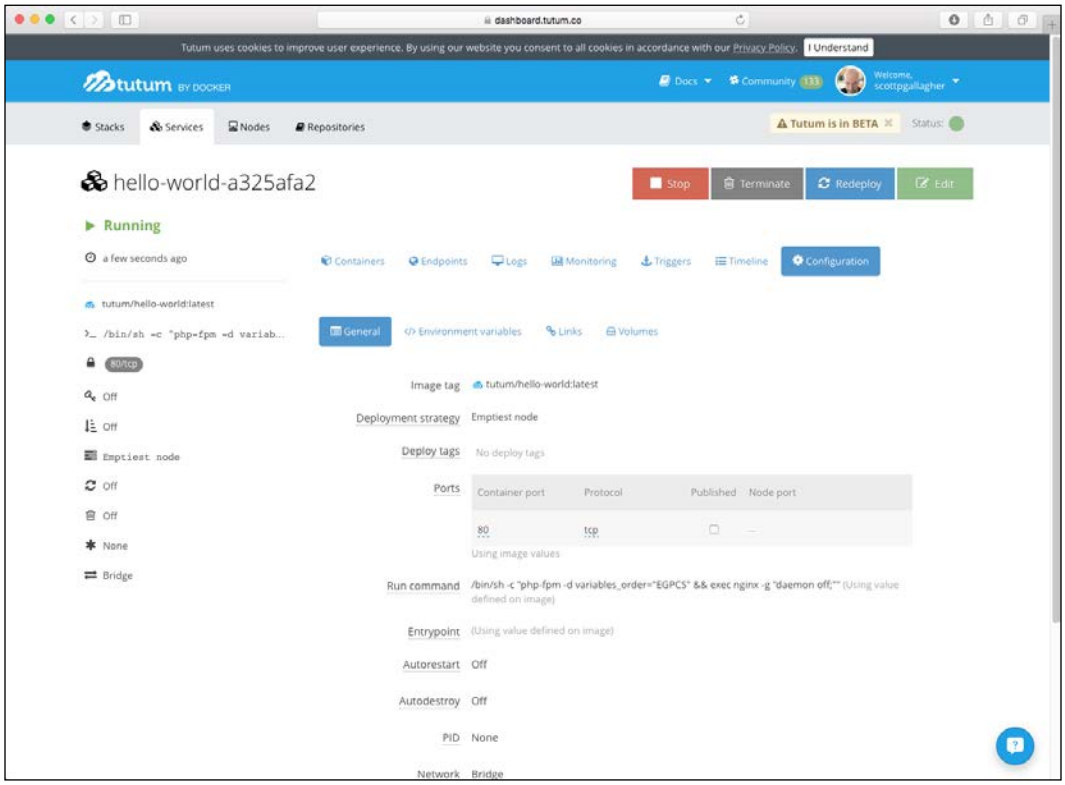
Timeline

Again, we have the **Timeline** section that we saw with regards to the nodes. We can see the lifespan of a container as well.



Configuration

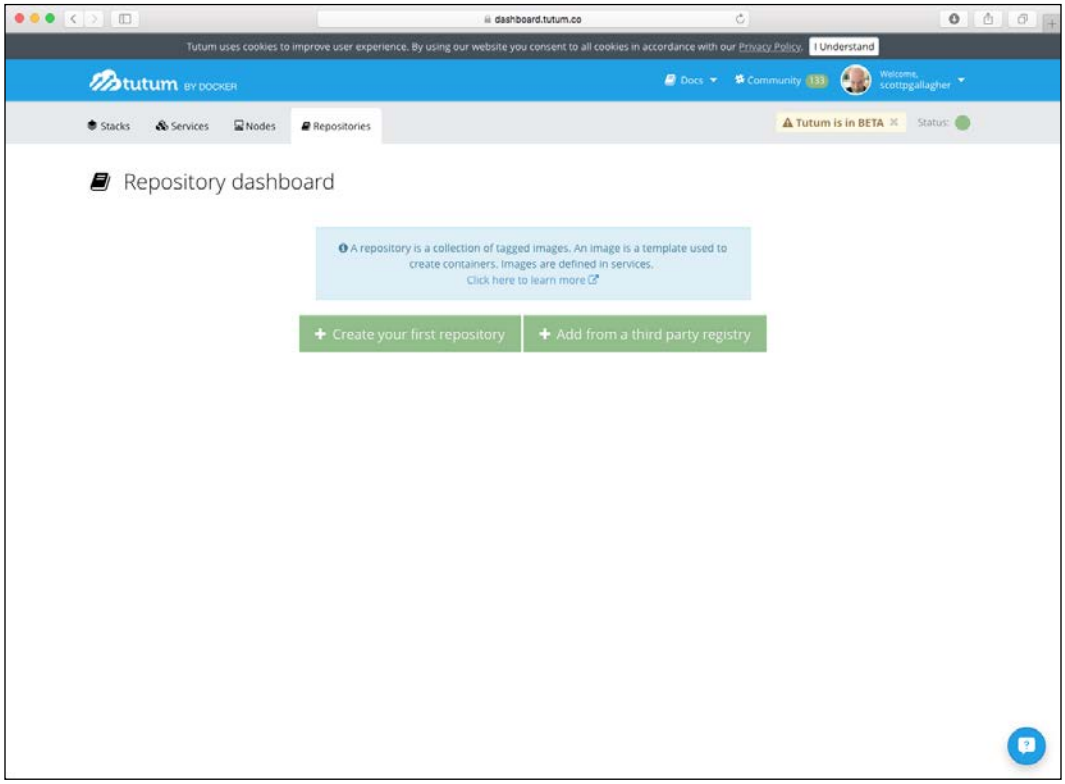
Lastly, we have the **Configuration** section that shows an overview of the container as a whole.



This section is also broken down into subsections that include general information, environmental variables, container links, and attached volumes for the container.

The Repositories tab

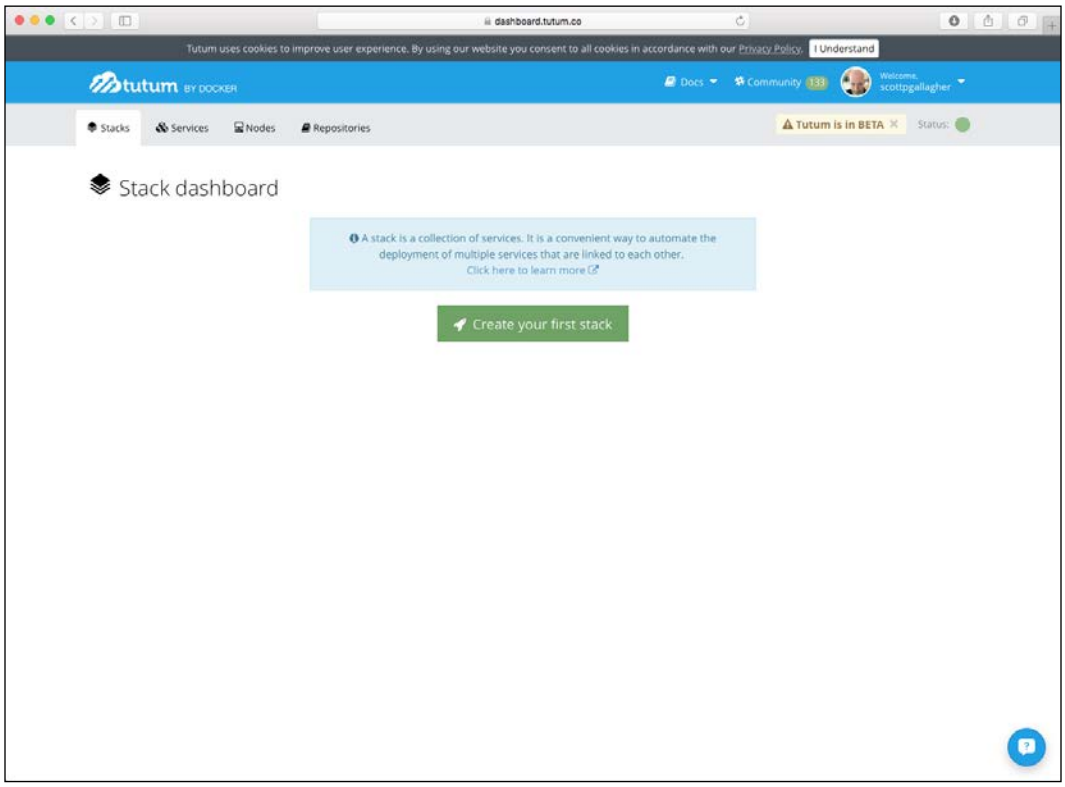
Let's take a look back at the navigation bar at the top and click on the **Repositories** tab.



In this tab, you can add custom repositories beyond Docker Hub; for example, if you were running your own private repositories, where your company would be storing images that you would want to use, you would add that in this section.

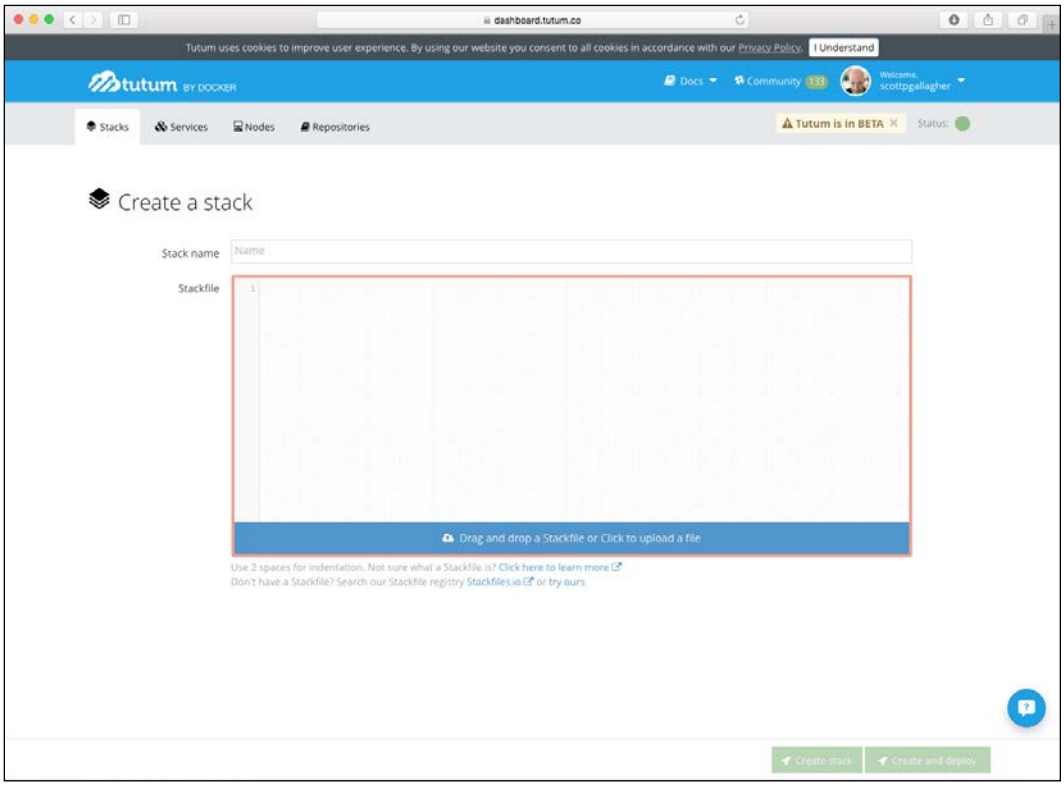
Stacks

There is also the **Stacks** section. Stacks are a collection of services similar to what you would think of when you are using Docker Compose.



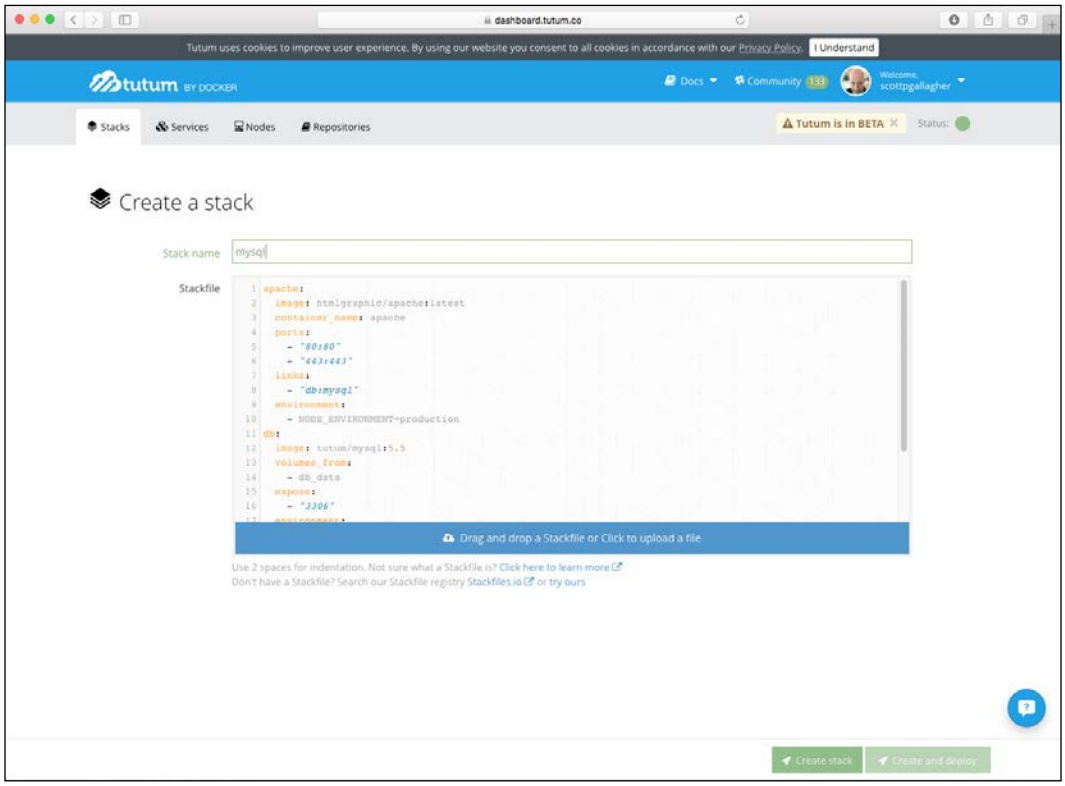
Let's take a look at this section, because it can be very useful while using development environments or for testing.

After we click on **Create your first stack**, we are taken to a page that is similar to the following screenshot:



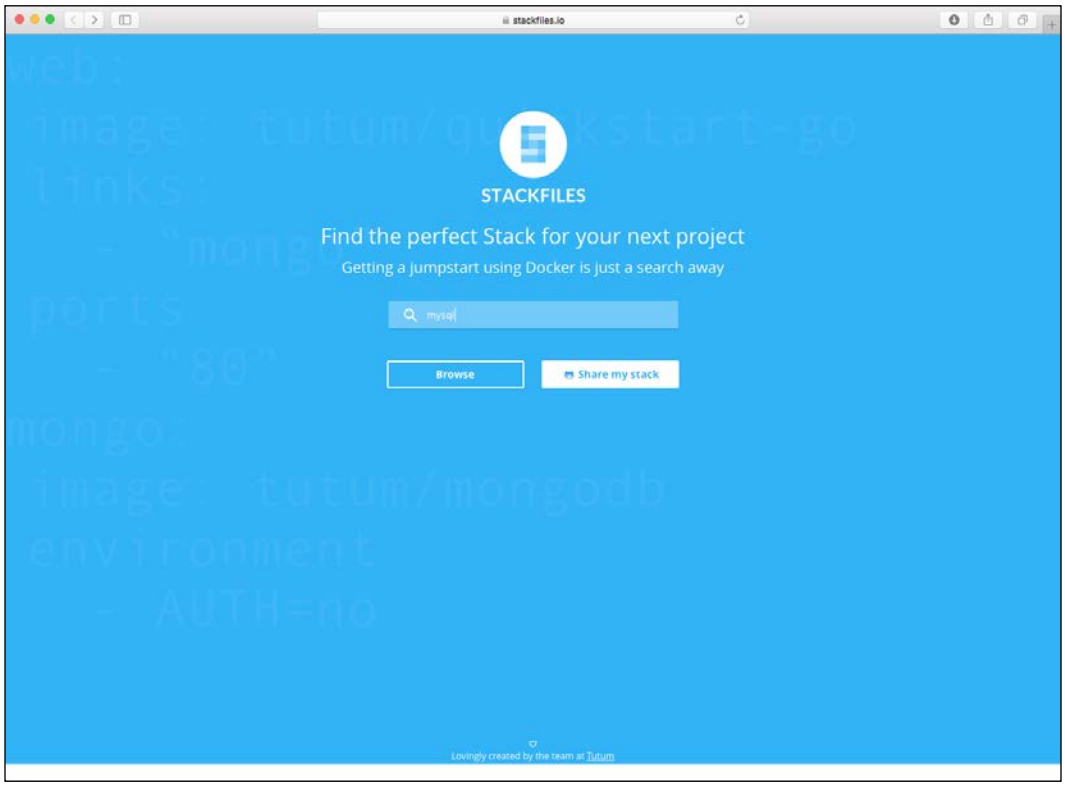
In this screenshot, we can see that we need pieces of information.

We need a name for our stack and we need the stackfile contents. In our case, we are going to use our trustworthy MySQL example and call our stack `mysql`.



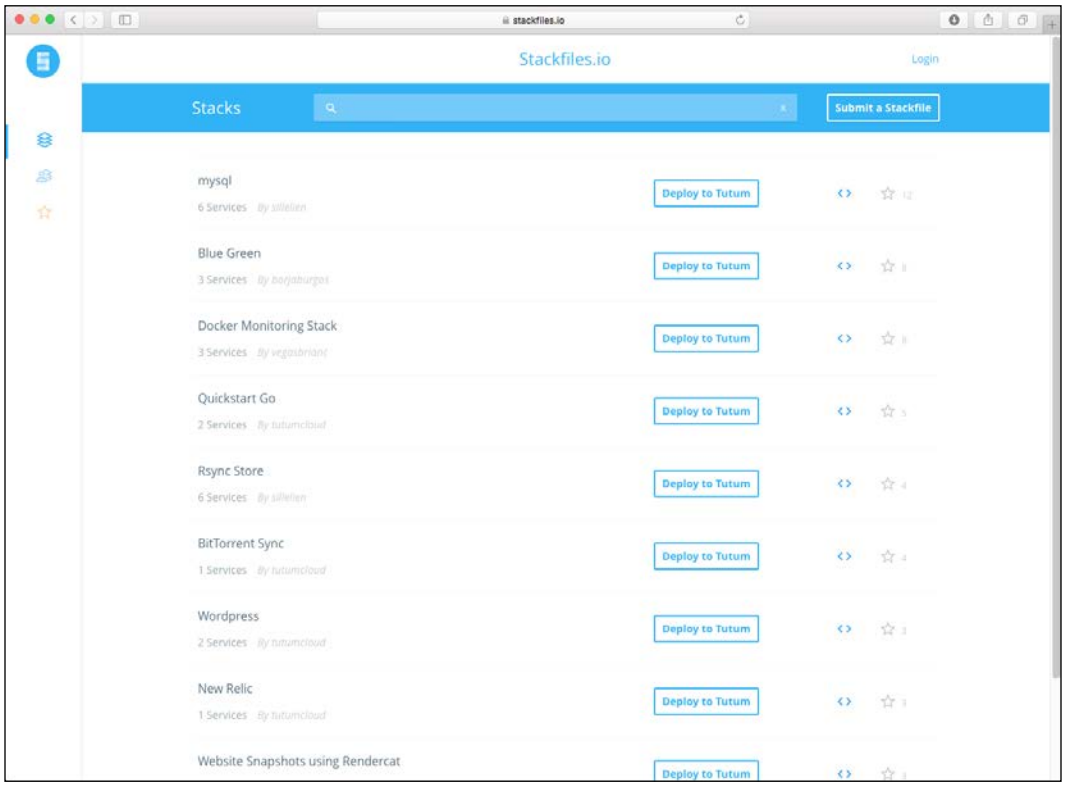
For our stackfile, we are going to use one of the resources that Tutum encourages us to explore. In the bottom section under the **Stackfile** field, there is an option to get a Stackfile from the Stackfile registry, which is located at `https://Stackfiles.io`.

Upon entering `stackfiles.io`, we are presented with an easy search box.



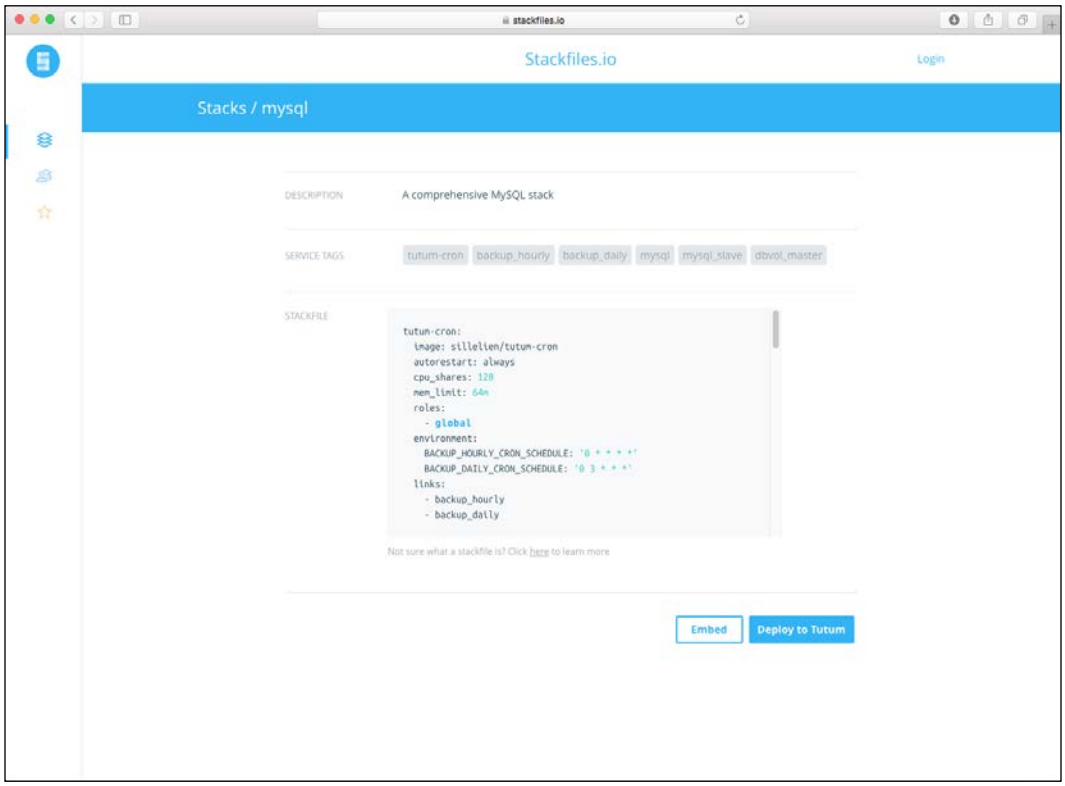
Again, for our test, we want to find the `mysql` stackfile, so we enter `mysql` in the box and click on **Browse**.

Now, for our example, we want a `mysql` one and we can see it right on the top.



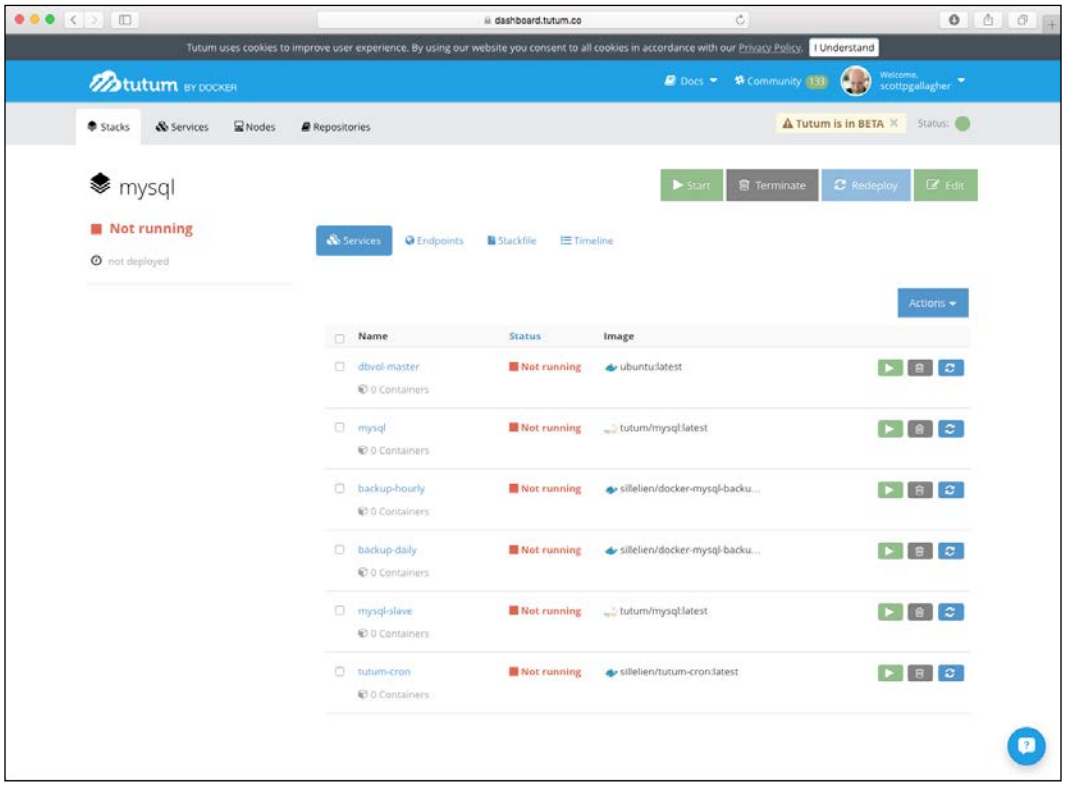
However, you could use a different one or search for one as well to see if there is one already done for you. Again, always work smarter, not harder!

So, if you drill into the `mysql` stackfile, you can see what all it is doing.



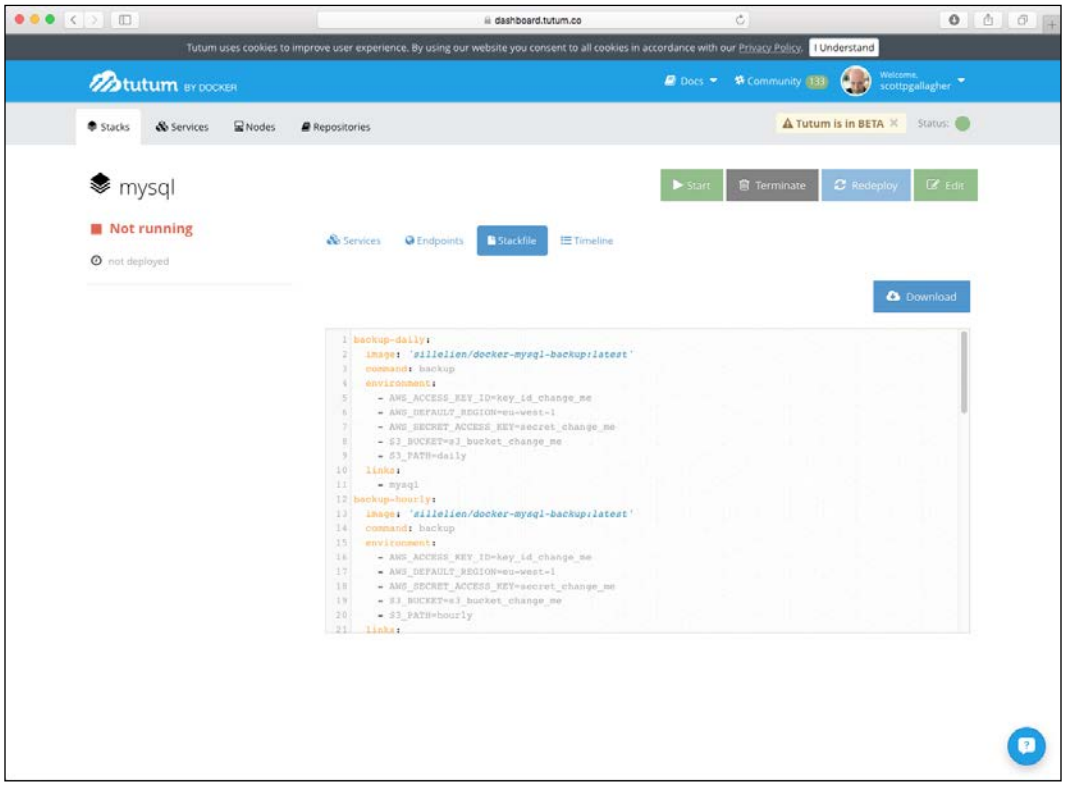
In our case, we are just going to copy this, go back to our Tutum stack deployment page, and paste it among the contents of the stackfile.

After we paste its contents in our **Stackfile** field and click on the **Launch stack** button, we will see our stack come to life.



After a few minutes, it will fire up for us and we will have created and be running our first stack. We can then manipulate the various pieces of the stack by starting/stopping them, terminating them, redeploying them, or even editing their configurations.

We can also look at the stackfile being used and edit it if needed to our likings or download it to share it with others as well.



Summary

We have now looked at three very powerful GUI tools that you can add to your Docker arsenal. With these tools, you can manipulate everything from your host environments, the images that live on those hosts, as well as the containers running on those hosts. You can scale them, manipulate them, and even remove them as needed.

In the next and the final chapter, we will be looking at some advanced Docker topics such as how to scale your containers, and debugging and troubleshooting them. We will also look at the common issues that can arise as well as common solutions to these issues. We will also cover various APIs that pertain to Docker as well as how to contribute to Docker. We will dive into configuration management tools, advanced networking, as well as Docker volume management.

13
Advanced Docker

We've made it to the last chapter, and you've stuck with it until the end! In this chapter, we will be taking a look at some advanced Docker topics. Let's take a peek into what we will be covering in this chapter:

- Scaling Docker
- Using the discovery services
- Debugging or troubleshooting Docker
- Common issues and solutions
- Various Docker APIs
- Keeping your containers in check
- Contributing to Docker
- Advanced Docker networking

Scaling Docker

In this section, we will learn how to scale Docker. Earlier, in *Chapter 7*, *Docker Compose*, we looked at using Docker Compose to do our scaling. In this section, we will look at other technologies that we can utilize to do the scaling for us. We will take a look at two such technologies—one that you can use through the command line and the other two that can be used through a web interface.

- **Kubernetes**: We have looked at another command line earlier to scale Docker—Docker Compose. There are other tools out there that you can use to scale your Docker environments from the command line. One such tool is Kubernetes:

  ```
  $ kubectl scale [--resource-version=version] [--current-
  replicas=count] --replicas=COUNT RESOURCE NAME


  $ kubectl scale --current-replicas=1 --replicas=2 Host Node
  ```

  You can find out more about it at `http://kubernetes.io/v1.0/docs/user-guide/kubectl/kubectl_scale.html`.

- **Mist.io**: With Mist.io, you can perform all your Docker actions in this software, everything from adding your cloud environments to locally run Docker installations. You can then see all the machines or nodes that are on that host and check whether they are running or have been stopped. You can also view information about them such as any alerts that they may have as well as their usage. You can also scale environments within the web console as well. While Mist.io is free to use, there is a fee if you want to use their monitoring service. It does come with a free trial for 15 days though. Scaling is done just by selecting the node that you want to scale and entering a value to scale to, the rest is all done automatically for you.
- **Shipyard**: When it comes to being able to scale easily, I am not sure there is an easier way than using Shipyard. Like Mist.io, you can easily scale nodes by using Shipyard. In *Chapter 10*, *Shipyard*, we saw how easy it was to do tasks such as scale running containers using Shipyard.

Using discovery services

In this section, we will learn how to scale Docker, but in a different way. Previously, we looked at using Docker Compose to do our scaling. In this section, we will look at other technologies that we can utilize to do the scaling for us automatically. There are some discovery services that we can tap into for this usage. We will focus on two of them in this section as they are the more popular ones.

Consul

One of the more popular options for discovery services with regards to Docker is Consul. Consul is an extremely easy-to-use discovery service that offers a lot of options that we can tie this into automatically updating the items in Consul by using a program called **Registrator** or by automatically taking those items that are updated in Consul and then turning around and updating a configuration file to show those updated items by using the `consul-template` program. Information about Consul can be found at `https://consul.io/`. For more information on Registrator, visit `http://gliderlabs.com/registrator/latest/`. And, to know more about `consul-template`, refer to `https://github.com/hashicorp/consul-template`.

Adding these three pieces to your technology arsenal can greatly increase the level of performance and uptime that you can provide. You can add new nodes to a service on the fly, and have the configuration on a particular container be updated on the fly. You can also move the updated nodes into a service and then remove the other ones that aren't updated so that you can provide a method for zero downtime with rolling updates as well. You can also go the other way if you notice something you updated isn't functioning properly. You can roll an older version of something into a discovery service while rolling out the newer version if a bug or security vulnerability is discovered. The possibilities of what you can do with these three pieces can be endless.

etcd

If you are going extremely lightweight with your host environments and using CoreOS, then you are very familiar with etcd. It uses a dynamic configuration registry to do discovery. When etcd is configured on each CoreOS host, they can do key-value distribution and replication, which allows them to discover each other as well as new etcd hosts.

etcd focuses on being:

- Simple
- Secure
- Fast
- Reliable

To find out more about etcd, refer to `https://en.wikipedia.org/wiki/CoreOS#ETCD`. You can also visit `https://github.com/coreos/etcd`, which contains information not just about what etcd can do, but also the ways you can get support for it, roadmap, mailing list, and reported bugs. You can also refer to `https://coreos.com/etcd/` and `https://github.com/coreos/etcd`.

Two of the more well-known projects that are using etcd are:

- Kubernetes
- Cloud Foundry

To view other projects that also use etcd, visit `https://github.com/search?utf8=%E2%9C%93&q=etcd`.

Debugging or troubleshooting Docker

Now that we have our Docker containers running in our production level service, we need to know how we can troubleshoot them—how do we fix common problems with containers, what should we be looking out for, and how can we quickly debug issues that do arise in our environments to avoid any serious downtime? Let's take a look at some of the topics that we can cover.

Docker commands

There are quite a few built-in Docker commands that you can use to help debug and troubleshoot Docker. With focus on running the containers themselves, here are the ones that can help you:

- **Docker history**: This lets you view the history of Docker image
- **Docker events**: This lets you view the live stream of the container events
- **Docker logs**: This lets you view output from a container
- **Docker diff**: This lets you view the changes of a container's filesystem
- **Docker stats**: This helps you view the live stream of a container's resource usage

GUI applications

The best way to be able to debug or troubleshoot your containers is to have a visual overview of all your containers. There are a few options for you out there that we can use:

- Shipyard (`https://shipyard-project.com`)
- Mist.io (`http://mist.io`)
- DockerUI (`https://github.com/crosbymichael/dockerui`)

Now only these options will allow you to get an overview of the status on all your running containers. You can also manipulate these containers, that is, you can restart them or view the logs for a particular container. While some of the options will do more than others, it is important to review them all to see what is the best fit for what you would like to see and be able to perform.

Resources

While there are a lot of resources out there for Docker, you would want to make sure you are focusing on the following two at all times, as they are the official means by which you can get information or obtain help:

- **Docker documentation**: This is an official documentation straight from Docker
- **Docker IRC room**: This is the official communication for the Docker community and a place where you can not only get help from others in the Docker community, but also assistance from those who work at Docker

Common issues and solutions

What are some common issues that others have run into putting their environments into production while using various Docker products? What are the solutions to those common issues? How can we mitigate against these issues so that no further instances occur? Let's take a look at what we can do!

Docker images

When you are using images, remember two things:

- Each image you pull takes up space
- Each time you run an image, that particular run is stored using disk space

If you are running low on space, this might be something to keep an eye on before it becomes a problem. If the space fills up, the containers might stop working, and this might lead to loss of data. Now you can view the images that you currently have by running a simple command:

```
$ docker images
```

To remove a particular image, we can run another command:

```
$ docker rmi <image_name>
```

But what about those images whose run is stored using disk space? How do we view them? There is a switch that can be added onto the `images` subcommand to view them:

```
$ docker images -a
```

You can remove these, by using their image ID:

```
$ docker rmi <image_ID>
```

Docker volumes

As of Docker v1.9, you can manage volumes through the Docker CLI. Let's take a look at what all can we do and how:

```
$ docker volume --help

Usage:  docker volume [OPTIONS] [COMMAND]

Manage Docker volumes

Commands:
  create                  Create a volume
  inspect                 Return low-level information on a volume
  ls                      List volumes
  rm                      Remove a volume

Run 'docker volume COMMAND --help' for more information on a command

  --help=false       Print usage
```

So we can do quite a lot; we can create volumes, inspect the volumes, list volumes, and remove volumes. Let's take a look at each, going through the lifecycle of a volume, that is, from creation to deletion:

```
$ docker volume create --name test
test
$ docker volume ls
local               test
```

Now you will notice this one was created locally. You can use the `--driver` flag and specify which volume driver to use:

```
$ docker volume inspect test
[
    {
        "Name": "test",
        "Driver": "local",
        "Mountpoint": "/var/lib/docker/volumes/test/_data"
    }
]
```

With this, we can see the name of the volume, which driver was used to create it, and where it's located on our system:

```
$ docker volume rm test
test
```

Using resources

Be sure to use all the resources that are out there. Those resources could include:

- Docker IRC room
- Docker documentation
- Docker commands

Various Docker APIs

Some of the various Docker APIs can immensely help you when you are writing up a script in the coding language of your choice. You can tie that into pulling the strings on Docker and have it to do the work for you without having to break out into another program or scripting language.

docker.io accounts API

This API is used just for account management. With it, you can:

- Get a single user
- Update various parameters for a particular user
- List e-mail addresses for a user
- Add an e-mail address for a user
- Delete an e-mail address for a user

There is not a lot that you can do with this API as it is mainly focused around what you can do with one's user account. In reality, there isn't a lot of information baked into one's user account, and as you can see, the e-mail address is the main focal point of one's account.

For more information, please visit `https://docs.docker.com/reference/api/docker_io_accounts_api/`.

Remote API

Let's just start off by saying that the Remote API is very intense, and that's not a bad thing. When it comes to APIs, you want them to be able to do just anything you want so that you never have to leave your code to perform these actions. Here is the high-level overview of what you can do with this API:

- Endpoints
- Containers
- Images

So you heard me say it was very intense, but based on what you can do with it, it doesn't look very intense until you take a peek into it yourself. Think of all the things that you can normally do with a container or an image and then you will understand why I state that it is intense. Things such as creating containers or images, listing them out, and getting information about containers or images might include getting information about the files and folders inside a container, copying files or folders from a container, and removing a container or image. There are also ways to manipulate or "hijack", as the documentation puts it such as using the `docker run` command. You can retrieve the various codes from the `run` command and determine what the command is doing.

For more information on the Remote API, refer to `https://docs.docker.com/engine/reference/api/docker_remote_api/` and to know more about the latest Remote API, visit `https://docs.docker.com/reference/api/docker_remote_api_v1.20/`.

Keeping your containers in check

What are some of the tools that we can use to keep our containers the way we have set them up? How do we ensure that they stay the way we want them to? How do we ensure that if they do drift off or things change on them, we are able to put them back in place to where we want them to be? Let's see how we can achieve that.

Kubernetes

Kubernetes is an open source project that was developed by Google to help with the automating deployment of your containers as well as scaling and the operations of your containers, not only on one host, but across multiple hosts. Kubernetes has been set to work on almost every environment that can be imagined, from locally in a Vagrant or VMware environment to cloud solutions such as AWS or Microsoft Azure. There will be some terminology that will need to be learned beyond the Docker terms, but if you understand how Docker operates, learning the Kubernetes terminology will come naturally. For example, instead of hosts, Kubernetes calls them **pods**. Kubernetes uses a single master node to control all its pods. The documentation can provide a lot more information including examples on how to administer your pods, set up pod clusters, and much more.

More information on Kubernetes can be found at `http://kubernetes.io`.

Chef

The reason we are focusing on Chef in this section is that AWS uses it as part of one of the solutions that they offer—in the form of OpsWorks. OpsWorks allows you to set up and use Chef to automate not only your Docker containers, but also other aspects of your AWS environment. I have personally set up and used Chef to do a lot of system automation throughout my personal environments. With that being said, Chef can be a little tricky at first to learn how to set up the server and client environments. There is a steep learning curve at first as with almost any configuration management system, but Chef does seem to have a little bit of a larger one with respect to all the moving pieces that are involved with the server environment and setup.

I wanted to draw focus to Chef though because if you are going to be viewing your environment within AWS, it might be a good idea to use Chef since it does offer it as a service within AWS. OpsWorks allows you to easily set up and control your environments as well as use their built-in Chef cookbooks. You can learn more about Chef at `http://chef.io`.

Other solutions

Some other solutions that are worth checking out or even use, if you already have the setup, to manage your Docker environment are:

- Puppet (`http://puppetlabs.com`)
- Ansible (`http://www.ansible.com/`)
- SaltStack (`http://saltstack.com/`)

Contributing to Docker

So you want to contribute to Docker? Do you have a great idea that you would like to see in Docker or one of its components? Let's get you the information and tools that you need to have. If you aren't a programmer-type person, there are other ways you can help contribute as well. Docker has a massive audience and you can help with supporting other users of their services. Let's learn how you can do that!

Contributing to the code

One of the biggest ways you can contribute to Docker is helping with the Docker code. Since Docker is all open source, you can download the code to your local machine and work on new features and present them as pull requests back to Docker. Those will then get reviewed on a regular basis and if they feel what you have contributed should be in the service, they will approve the pull request. This can be very interesting when you get to know something you have written has been accepted.

You first need to know how you can get the setup to contribute. Everything is pretty much available at `https://github.com/docker`, which is open for you to help contribute to. But how do we go about getting the setup to help contribute? The best place to start is by following the guide at `https://docs.docker.com/project/who-written-for/`. The software you will need to contribute can be found by following another guide at `https://docs.docker.com/project/software-required/`.

These guides will help you get all the setup with the knowledge you will need, as well as the software. The last link that you will need to review is `https://github.com/docker/docker/blob/master/CONTRIBUTING.md`. This page will provide information on how to report issues, contribution tips and guidelines, community guidelines, and other important information about how to successfully contribute.

Contributing to support

You can also contribute to Docker by other means beyond contributing to the Docker code or feature sets. You can help by using the knowledge you have obtained to help others in their support channels. Currently, Docker uses IRC rooms where users can gather online and either provide support to other users or ask questions about the various services that they offer. The community is very open and someone is always willing to help. I have found it of great help when I run into something that I come across and scratch my head. It's also nice to get help and to help others back (a nice give and take). It also is a great place that harvests ideas for you to use. You can see what questions others are asking, based on their setups, and it could spur ideas that you may want to think about using in your environment.

You can also follow the GitHub issues that are brought up about the services. These could be feature requests and how Docker may implement them or the issues that have cropped up through the usage of services. You can help test out the issues that others are experiencing to see whether you can replicate it or find a possible solution to it.

Other contributions

There are other ways to contribute to Docker as well. You can do things such as presenting at conferences about Docker. You can also promote the service and gather interest at your institution. You can start the communication through your organization's means of communications such as e-mail distribution lists, group discussions, IT roundtables, or regularly scheduled meetings. You can also schedule your own meetings within your organization to get people talking or you can do Docker meetups. These meetups are designed to not only include your organization, but also the city or town members that your organization is in to get more widespread communication and promotion of the services. You can search whether there are already meetups in your area by visiting `https://www.docker.com/community/meetup-groups`.

Advanced Docker networking

Lastly, one of the up and coming features of Docker that we will be taking a look at will be that of the Docker networking. Now at its current form, this is a solution that has not yet been implemented, but is a feature set that will be coming soon. So, it's good to get ahead of the curve on this one and learn it so that you are ready to implement it or architect your future environments around it.

Installation

Since this feature is not part of the current Docker release, you need to install the experimental release to get this completed. To install Docker experimental releases, simply use the `curl` command that you have seen previously. Now this will only work on Linux and Mac currently. In future, experimental builds might be installed on Windows systems. So to install, use the following command:

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

On Mac, run:

```
$ curl -L https://experimental.docker.com/builds/Darwin/x86_64/docker-
latest > /usr/local/bin/docker
$ chmod +x /usr/local/bin/docker
```

Now you will get a warning message if you already have Docker installed:

```
Warning: the "docker" command appears to already exist on this system.


If you already have Docker installed, this script can cause trouble,
which is
why we're displaying this warning and provide the opportunity to cancel
the
installation.


If you installed the current Docker package using this script and are
using it
again to update Docker, you can safely ignore this message.


You may press Ctrl+C now to abort this script.
sleep 20
```

You want to make sure you are installing experimental builds to a machine that is not a production-related one. For example, you probably don't want to install an experimental release to your laptop if you are using it to develop and test Docker-related items on. Best practice would be to install it on a virtual machine that you can throw away if it gets broken.

After running the `curl` command, you will be able to see the networking option from the list of Docker commands now:

```
$ docker

Usage: docker [OPTIONS] COMMAND [arg...]
       docker daemon [ --help | ... ]
       docker [ --help | -v | --version ]

A self-sufficient runtime for containers.

Options:

  --config=~/.docker              Location of client config files
  -D, --debug=false               Enable debug mode
  -H, --host=[]                   Daemon socket(s) to connect to
  -h, --help=false                Print usage
  -l, --log-level=info            Set the logging level
  --no-legacy-registry=false      Do not contact legacy registries
  --tls=false                     Use TLS; implied by --tlsverify
  --tlscacert=~/.docker/ca.pem    Trust certs signed only by this CA
  --tlscert=~/.docker/cert.pem    Path to TLS certificate file
  --tlskey=~/.docker/key.pem      Path to TLS key file
  --tlsverify=false               Use TLS and verify the remote
  -v, --version=false             Print version information and quit

Commands:
    attach    Attach to a running container
    build     Build an image from a Dockerfile
    commit    Create a new image from a container's changes
    cp        Copy files/folders between a container and the local
filesystem
```

```
    create    Create a new container
    diff      Inspect changes on a container's filesystem
    events    Get real time events from the server
    exec      Run a command in a running container
    export    Export a container's filesystem as a tar archive
    history   Show the history of an image
    images    List images
    import    Import the contents from a tarball to create a filesystem
image
    info      Display system-wide information
    inspect   Return low-level information on a container or image
    kill      Kill a running container
    load      Load an image from a tar archive or STDIN
    login     Register or log in to a Docker registry
    logout    Log out from a Docker registry
    logs      Fetch the logs of a container

    network   Network management
    pause     Pause all processes within a container
    port      List port mappings or a specific mapping for the CONTAINER
    ps        List containers
    pull      Pull an image or a repository from a registry
    push      Push an image or a repository to a registry
    rename    Rename a container
    restart   Restart a container
    rm        Remove one or more containers
    rmi       Remove one or more images
    run       Run a command in a new container
    save      Save an image(s) to a tar archive
    search    Search the Docker Hub for images
    start     Start one or more stopped containers
    stats     Display a live stream of container(s) resource usage
statistics
    stop      Stop a running container
    tag       Tag an image into a repository
    top       Display the running processes of a container
    unpause   Unpause all processes within a container
```

```
    version   Show the Docker version information
    volume    Manage Docker volumes
    wait      Block until a container stops, then print its exit code

Run 'docker COMMAND --help' for more information on a command.
```

Creating your own network

In the preceding command output, I have highlighted the section that we will be focusing on—the `network` subcommand in Docker. There is also another command you may want to take a look at, and that is the `volume` subcommand, but we will be focusing on the `network` subcommand.

Let's create ourselves a network that our Docker containers can use to communicate on. From the output of the `docker network` command, we can see our options:

```
$ docker network

docker: "network" requires a minimum of 1 argument.
See 'docker network --help'.

Usage: docker network [OPTIONS] COMMAND [OPTIONS] [arg...]

Commands:
  create                  Create a network
  rm                      Remove a network
  ls                      List all networks
  info                    Display information of a network

Run 'docker network COMMAND --help' for more information on a command.
```

Doing a `docker ls` will give us a view of what our current network setup is:

```
$ docker network ls

NETWORK ID          NAME                TYPE
02f3d3834733          none                    null
b22ff5151bcb           host                       host
f4b7c38b83b1          bridge                  bridge
```

Now let's get to creating ourselves a network. Using the `network` subcommand as well as the `create` option, we can create ourselves a network:

```
$ docker network create <name>
$ docker network create docker-net
21625dd96ac08e1713621d951cfa140cebee96c9fae9f8ff44748f86a4c731d7
$ docker network ls

NETWORK ID          NAME                TYPE
02f3d3834733         none                     null
b22ff5151bcb          host                       host
f4b7c38b83b1         bridge                   bridge
21625dd96ac0        docker-net            bridge
```

Now that we have our network, how do we tell our containers about it? That comes with a `--publish-service=` switch when you use your `docker run` command:

```
$ docker run -it --publish-service=<name>.<network_name> ubuntu:latest /
bin/bash

$ docker run -it --publish-service=web.docker-net ubuntu:latest /bin/bash
```

We can also create networks and provide drivers for those networks so that they can span across multiple hosts. By default, there is a driver named overlay that will allow you to do this. Now this is the first of many drivers that will be coming on board, either when this network feature is baked into Docker or at a later time, for sure. When you create the network is when you will specify the overlay driver. However, there is one thing that this driver does need. It will need access for not only itself, but also the other Docker hosts that you want to network together:

```
$ docker network create -d overlay docker-overlay
```

Networking plugins

Going back to our previous example of using the overlay driver, this is also considered a Docker network plugin. While networking has the use for plugins, keep in mind that volumes also have the option to do plugins or drivers as well. With regards to networking plugins though, there is quite a list of plugins that are already available, and I can only assume that others will be added quickly. Currently that list of networking plugins consists of:

- Weave

- Project Calico
- Nuage Networks
- Cisco
- VMware
- Microsoft
- Midokura

To use these plugins, we simply change what we are using in the `--publish-service=` option, for example:

```
$ docker run -it --publish-service=service.network.cisco ubuntu:latest /
bin/bash
$ docker run -it --publish-service=service.network.vmware ubuntu:latest /
bin/bash
$ docker run -it --publish-service=service.network.microsoft
ubuntu:latest /bin/bash
```

Note that some of the names may change before they actually come to production level.

Summary

In this chapter, we looked at a lot of items in depth. We covered various aspects of Docker such as how we can scale our environments and use Docker services. Later, you came to know about the various techniques that can be used to debug or troubleshoot the issues that crop up while using Docker along with the solutions. You then learned how contribution of codes can be done to Docker and its networking.

I hope you have enjoyed this book and will continue to refine your skill set when it comes to Docker. It really is a technology that is on the tip of everyone's tongue these days, so knowing it will not only benefit you at your current position, but also any future positions you may be looking at. Throughout the chapters, you should be able to pick up on some ways to get in touch with me if you do have any questions or want to provide any feedback. I am frequently on the IRC rooms that Docker has, so hit me up sometime to chat. Good luck and use the resources out there to your advantage!

Index

D

P

R

S

T

Thank you for buying

Mastering Docker

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Open Source

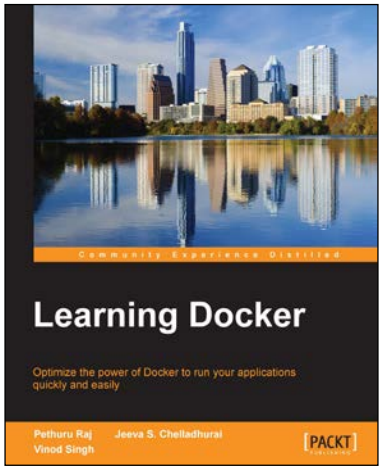
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING

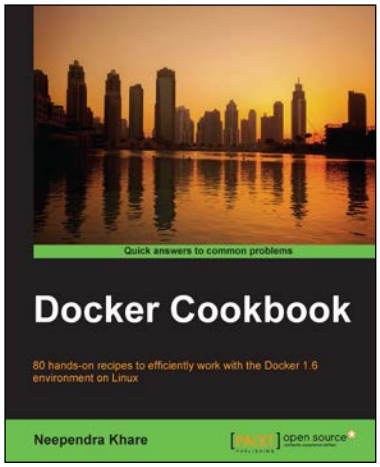


Learning Docker

ISBN: 978-1-78439-793-7 Paperback: 240 pages

Optimize the power of Docker to run your applications quickly and easily

1. Learn to compose, use, and publish the Docker containers.
2. Leverage the features of Docker to deploy your existing applications.
3. Explore real-world examples of securing and managing Docker containers.



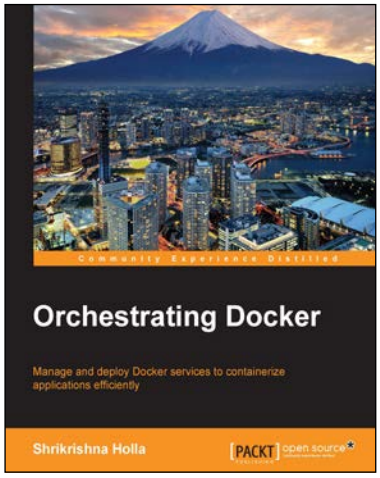
Docker Cookbook

ISBN: 978-1-78398-486-2 Paperback: 248 pages

80 hands-on recipes to efficiently work with the Docker 1.6 environment on Linux

1. Provides practical techniques and knowledge of various emerging and developing APIs to help you create scalable services.
2. Create, manage, and automate production-quality services while dealing with inherent issues.
3. Each recipe is carefully organized with instructions to complete the task efficiently.

[PACKT] open source
community experience distilled
PUBLISHING

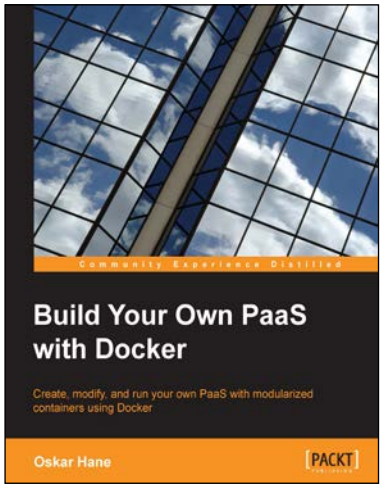


Orchestrating Docker

ISBN: 978-1-78398-478-7 Paperback: 154 pages

Manage and deploy Docker services to containerize applications efficiently

1. Set up your own Heroku-like PaaS by getting accustomed to the Docker ecosystem.
2. Run your applications on development machines, private servers, or the cloud, with minimal cost of a virtual machine.
3. A comprehensive guide to the smooth management and development of Docker containers and its services.



Build Your Own PaaS with Docker

ISBN: 978-1-78439-394-6 Paperback: 138 pages

Create, modify, and run your own PaaS with modularized containers using Docker

1. Build your own PaaS using the much-appreciated software Docker.
2. Isolate services in containers to have a fully modularized and portable system.
3. Step-by-step tutorials that take you through the process of creating your own PaaS.