NEW FOR 2016!

Docker

Creating Structured Containers

CURATED COURSE

[PACKT]

Docker

Creating Structured Containers

A course in five modules

Rethink what's possible with Docker and optimize its power
with your Course Guide Ankita Thakur

Learn to use the next-platform Docker from start to finish!

To contact your Course Guide
Email: ankitat@packtpub.com

[PACKT]

BIRMINGHAM - MUMBAI

Meet Your Course Guide

Hello and welcome to this Docker course! You now have a clear pathway to become proficient in Docker.

This course has been planned and created for you by me Ankita Thakur – I am your Course Guide, and I am here to help you have a great journey along the pathways of learning that I have planned for you.

I've developed and created this course for you and you'll be seeing me through the whole journey, offering you my thoughts and ideas behind what you're going to learn next and why I recommend each step. I'll provide tests and quizzes to help you reflect on your learning, and code challenges that will be pitched just right for you through the course.

If you have any questions along the way, you can reach out to me over e-mail or telephone and I'll make sure you get everything from the course that we've planned. Details of how to contact me are included on the first page of this course.

What's so cool about Docker?

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 course, 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!

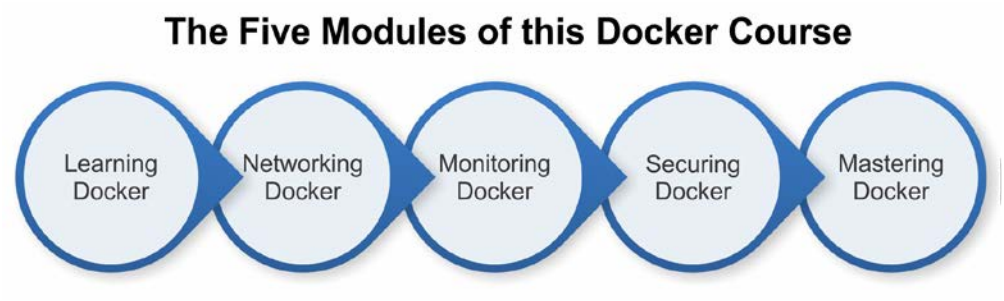
The popular Docker containerization platform has come up with an enabling engine to simplify and accelerate the life cycle management of containers. There are industry-strength and openly automated tools made freely available to facilitate the needs of container networking and orchestration. Therefore, producing and sustaining business-critical distributed applications is becoming easy. Business workloads are methodically containerized to be easily taken to cloud environments, and they are exposed for container crafters and composers to bring forth cloud-based software solutions and services. Precisely speaking, containers are turning out to be the most featured, favored, and fine-tuned runtime environment for IT and business services.

What's in it for me – Course Structure

Docker has been a game-changer when it comes to virtualization. It has now grown to become a key driver of innovation beyond system administration. It is now having an impact on the world of web development and beyond. But how can you make sure you're keeping up with the innovations that it's driving? How can you be sure you're using it to its full potential? This course is meticulously designed and developed in order to empower developers, cloud architects, sysadmins, business managers, and strategists, with all the right and relevant information on the Docker platform and its capacity to power up mission-critical, composite, and distributed applications across industry verticals.

However, I want to highlight that the road ahead may be bumpy on occasions, and some topics may be more challenging than others, but I hope that you will embrace this opportunity and focus on the reward. Remember that we are on this journey together, and throughout this course, we will add many powerful techniques to your arsenal that will help us solve the problems.

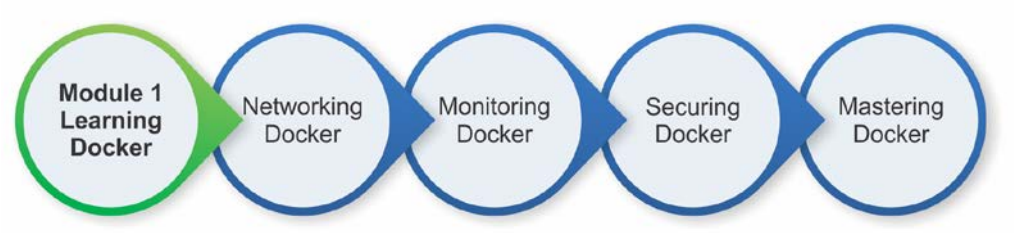
I've created this learning path for you that consists of five models. Each of these modules is a mini-course in their own way, and as you complete each one, you'll have gained key skills and be ready for the material in the next module.



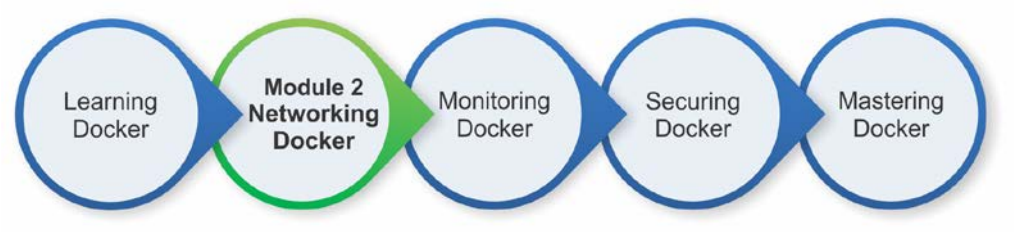
So let's now look at the pathway these modules create—basically all the topics that will be exploring in this learning journey.

Course Journey

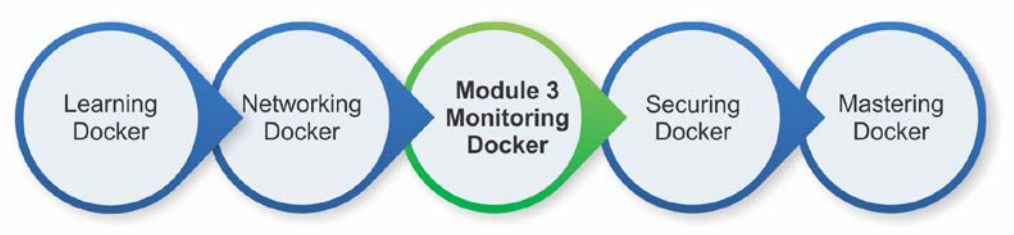
We start the course with our very first module, *Learning Docker*, to help you get familiar with Docker. This module is a step-by-step guide that will walk you through the various features of Docker from Docker software installation to knowing Docker in detail. It will cover best practices to make sure you're confident with the basics, such as building, managing, and storing containers, before diving deeper into advanced topics of Docker.



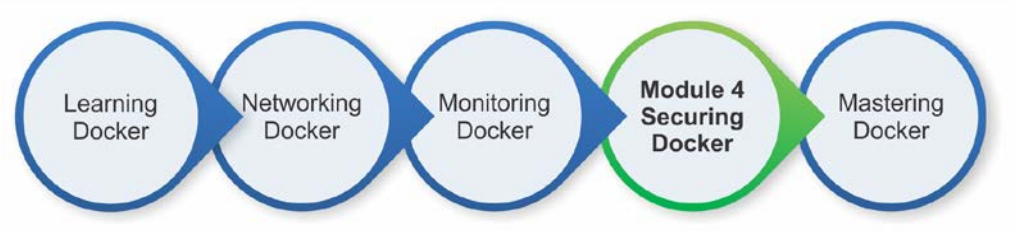
Docker provides the networking primitives that allow administrators to specify how different containers network with each application and connect each of its components, then distribute them across a large number of servers and ensure coordination between them irrespective of the host or VM they are running in. The second module, *Networking Docker*, will show you how to create, deploy, and manage a virtual network for connecting containers spanning single or multiple hosts.



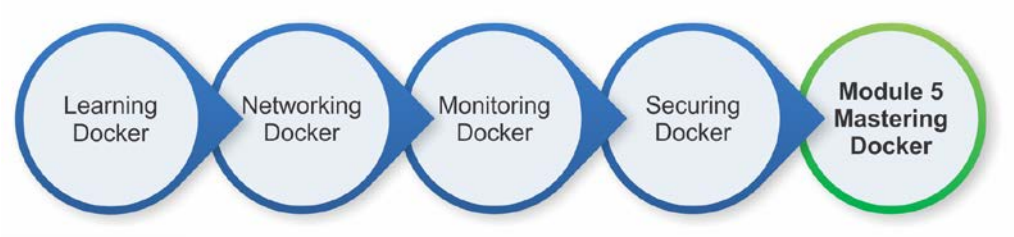
Next, we come to our third module, *Monitoring Docker*. This module will show you how monitoring containers and keeping a keen eye on the working of applications helps improve the overall performance of the applications that run on Docker. This module will cover monitoring containers using Docker's native monitoring functions, various plugins, as well as third-party tools that help in monitoring.



With the rising integration and adoption of Docker containers, there is a growing need to ensure their security. The purpose of our fourth module, *Securing Docker*, is to provide techniques and enhance your skills to secure Docker containers easily and efficiently. It will share the techniques to configure Docker components securely and explore the different security measures/methods one can use to secure the kernel. Furthermore, it will cover the best practices of reporting Docker security findings and will show you how you can safely report any security findings you come across.



Finally, the last module—*Mastering Docker*! Now that you've learned the nitty-gritty of Docker, it's time to take a step ahead and learn some advanced topics. This module will help you deploy Docker in production. You also learn three interesting GUI applications: Shipyard, Panamax, and Tutum.



Ankita Thakur

Your Course Guide

Did you know that average salary for Docker jobs is about $28, 000?

Docker is an emerging field and there is a high demand of people who knows this booming technology.

The Course Roadmap and Timeline

Here's a view of the entire course plan before we begin. This grid gives you a topic overview of the whole course and its modules, so you can see how we will move through particular phases of learning to use Docker, what skills you'll be learning along the way, and what you can do with those skills at each point. I also offer you an estimate of the time you might want to take for each module, although a lot depends on your learning style how much you're able to give the course each week!

	Course Module 1	Course Module 2	Course Module 3	Course Module 4	Course Module 5
Module	**Learning Docker**	Networking Docker	**Monitoring Docker**	**Securing Docker**	**Mastering Docker**
Skills learned	Docker basic fundamentals including how to install Docker	Become an expert Linux administrator by learning Docker networking	Complete knowledge of how to implement monitoring for your containerized applications and make the most of the metrics you are collecting	Complete understanding of Docker security and will be able to protect your container-based applications	Learn how to utilize Docker's full potential
Key Topics	Docker commands, Docker Hub, Docker Engine, Docker containers and images, Docker Machine, Docker Compose, and Docker Swarm	Docker networking on multiple hosts, overcome pitfalls of networking, Docker with Kubernetes, and CNM model	Augment Docker's built-in tools with modern tools such as cAdvisor, Sysdig, and Prometheus	Secure your Docker hosts and nodes, Docker Bench Security, Traffic Authorization, Summon, sVirt, and SELinux	Deploying Docker in production, Shipyard, Panamax, Tutum, and scaling Docker
Suggested Time per Module	**2 weeks** to familiarize yourself with Docker	**1 week** to know the basics of networking and see how Docker networking works	**1 week** to learn how to monitor your Docker applications	**2 weeks** to learn how to secure your Docker	**4 days** to get acquainted with advanced topics of Docker
Things you can do with Docker by this point	Know the nitty-gritty of Docker, being able to orchestrate and manage the creation and deployment of Docker containers	Create, deploy, and manage a virtual network for connecting containers spanning single or multiple hosts	Monitor your Docker containers and their apps using various native and third-party tools	Secure your Docker environment	Become an expert in the innovative containerization tool, Docker

Table of Contents

Course Module 1: Learning Docker

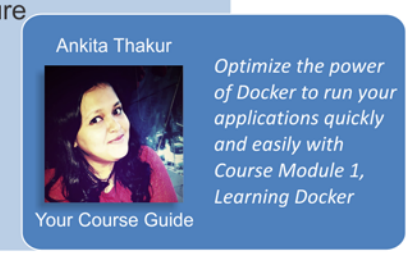
Course Module 2: Networking Docker

Course Module 3: Monitoring Docker

Course Module 4: Securing Docker

Course Module 5: Mastering Docker

Course Module 1
Learning Docker



Course Module 1
Learning Docker

Course Module 1

Docker is a next-generation platform for simplifying application containerization life-cycle. Docker allows you to create a robust and resilient environment in which you can generate portable, composable, scalable, and stable application containers. Let's begin this learning journey with the first module of the course, *Learning Docker*, and get started with Docker—the Linux containerizing technology that has revolutionized application sandboxing.

Ankita Thakur

Your Course Guide

We'll start off by elucidating the installation procedure for Docker and a few troubleshooting techniques. You will be introduced to the process of downloading Docker images and running them as containers. You'll learn how to run **containers as a service (CaaS)** and also discover how to share data among containers. This module will teach you how to use Docker Machine to build new servers from scratch. You'll learn how to take greater control over your containers using some of Docker's most sophisticated and useful tools, such as Docker Compose and Docker Swarm. You'll explore how to establish the link between containers and orchestrate containers using Docker Compose. You will also come across relevant details about application testing inside a container. You will discover how to debug a container using the docker exec command and the nsenter tool. By the end of this module, I'm sure you'll become fluent with the basic components of Docker.

What are you looking for? Let's get started...

1
Getting Started with Docker

These days, Docker technology is gaining more market and more mind shares among information technology (IT) professionals across the globe. In this chapter, we would like to shed more light on Docker, and show why it is being touted as the next best thing for the impending cloud IT era. In order to make this book relevant to software engineers, we have listed the steps needed for crafting highly usable application-aware containers, registering them in a public registry repository, and then deploying them in multiple IT environments (on-premises as well as off-premises). In this book, we have clearly explained the prerequisites and the most important details of Docker, with the help of all the education and experiences that we could gain through a series of careful implementations of several useful Docker containers in different systems. For doing this, we used our own laptops as well as a few leading public **Cloud Service Providers** (**CSP**).

We would like to introduce you to the practical side of Docker for the game-changing Docker-inspired containerization movement.

In this chapter, we will cover the following topics:

- An introduction to Docker
- Docker on Linux
- Differentiating between containerization and virtualization
- Installing the Docker engine
- Understanding the Docker setup
- Downloading the first image
- Running the first container
- Running a Docker container on **Amazon Web Services** (**AWS**)
- Troubleshooting the Docker containers

An introduction to Docker

Due to its overwhelming usage across industry verticals, the IT domain has been stuffed with many new and path-breaking technologies used not only for bringing in more decisive automation but also for overcoming existing complexities.

Virtualization has set the goal of bringing forth IT infrastructure optimization and portability. However, virtualization technology has serious drawbacks, such as performance degradation due to the heavyweight nature of **virtual machines** (**VM**), the lack of application portability, slowness in provisioning of IT resources, and so on. Therefore, the IT industry has been steadily embarking on a Docker-inspired containerization journey. The Docker initiative has been specifically designed for making the containerization paradigm easier to grasp and use. Docker enables the containerization process to be accomplished in a risk-free and accelerated fashion.

Precisely speaking, **Docker** is an open source containerization engine, which automates the packaging, shipping, and deployment of any software applications that are presented as lightweight, portable, and self-sufficient containers, that will run virtually anywhere.

A Docker **container** is a software bucket comprising everything necessary to run the software independently. There can be multiple Docker containers in a single machine and containers are completely isolated from one another as well as from the host machine.

In other words, a Docker container includes a software component along with all of its dependencies (binaries, libraries, configuration files, scripts, jars, and so on). Therefore, the Docker containers could be fluently run on x64 Linux kernel supporting namespaces, control groups, and file systems, such as **Another Union File System** (**AUFS**). AUFS is a layered copy-on-write file system that shares common portions of the operating system between containers.

There have been many tools and technologies aimed at making distributed applications possible, even easy to set up, but none of them have as wide an appeal as Docker does, which is primarily because of its cross-platform nature and friendliness towards both system administrators and developers. It is possible to set up Docker in any OS, be it Windows, OS X, or Linux, and Docker containers work the same way everywhere. This is extremely powerful, as it enables a write-once-run anywhere workflow. Docker containers are guaranteed to run the same way, be it on your development desktop, a bare-metal server, virtual machine, data center, or cloud. No longer do you have the situation where a program runs on the developer's laptop but not on the server.

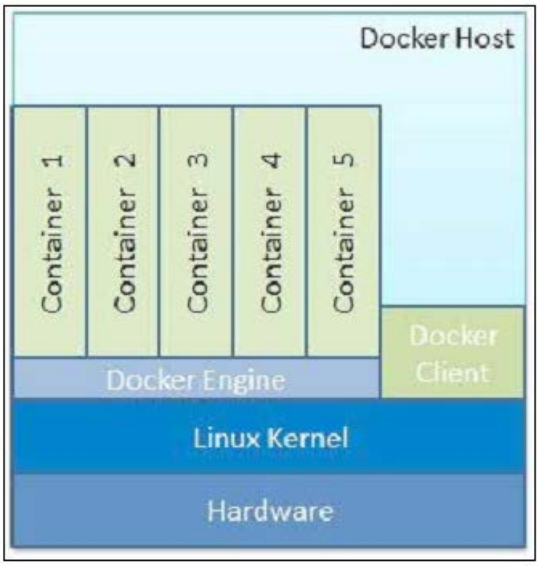
In a nutshell, the Docker solution lets us quickly assemble composite, enterprise-scale, and business-critical applications. For doing this, we can use different and distributed software components: Containers eliminate the friction that comes with shipping code to distant locations. Docker also lets us test the code and then deploy it in production as fast as possible. The Docker solution primarily consists of the following components:

- The Docker engine
- The Docker Hub

The Docker engine is for enabling the realization of purpose-specific as well as generic Docker containers. The Docker Hub is a fast-growing repository of the Docker images that can be combined in different ways for producing publicly findable, network-accessible, and widely usable containers.

Docker on Linux

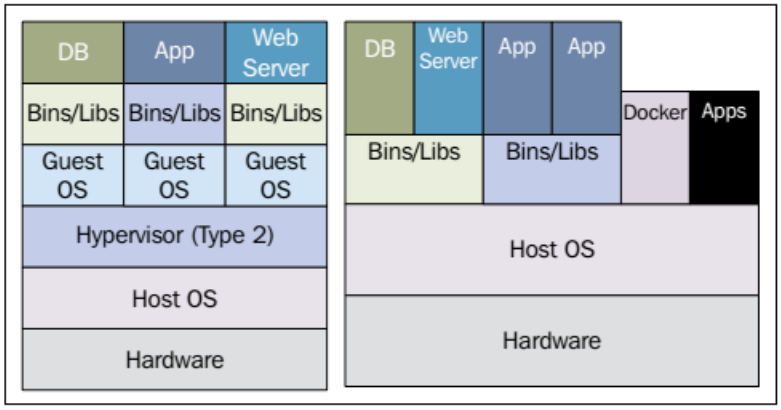
Supposethat we want to directly run the containers on a Linux machine. The Docker engine produces, monitors, and manages multiple containers as illustrated in the following diagram:



The preceding diagram vividly illustrates how future IT systems would have hundreds of application-aware containers, which would innately be capable of facilitating their seamless integration and orchestration for deriving modular applications (business, social, mobile, analytical, and embedded solutions). These contained applications could fluently run on converged, federated, virtualized, shared, dedicated, and automated infrastructures.

Differentiating between containerization and virtualization

It is pertinent, and paramount to extract and expound the game-changing advantages of the Docker-inspired containerization movement over the widely used and fully matured virtualization paradigm. In the containerization paradigm, strategically sound optimizations have been accomplished through a few crucial and well-defined rationalizations and the insightful sharing of the compute resources. Some of the innate and hitherto underutilized capabilities of the Linux kernel have been rediscovered. These capabilities have been rewarded for bringing in much-wanted automation and acceleration, which will enable the fledgling containerization idea to reach greater heights in the days ahead, especially those of the cloud era. The noteworthy business and technical advantages of these include the bare metal-scale performance, real-time scalability, higher availability, and so on. All the unwanted bulges and flab are being sagaciously eliminated to speed up the roll-out of hundreds of application containers in seconds and to reduce the time taken for marketing and valuing in a cost-effective fashion. The following diagram on the left-hand side depicts the virtualization aspect, whereas the diagram on the right-hand side vividly illustrates the simplifications that are being achieved in the containers:



The following table gives a direct comparison between virtual machines and containers:

Virtual Machines (VMs)	**Containers**
Represents hardware-level virtualization	Represents operating system virtualization
Heavyweight	Lightweight
Slow provisioning	Real-time provisioning and scalability
Limited performance	Native performance
Fully isolated and hence more secure	Process-level isolation and hence less secure

The convergence of containerization and virtualization

A hybrid model, having features from both the virtual machines and that of containers, is being developed. It is the emergence of system containers, as illustrated in the preceding right-hand-side diagram. Traditional hypervisors, which implicitly represent hardware virtualization, directly secure the environment with the help of the server hardware. That is, VMs are completely isolated from the other VMs as well as from the underlying system. But for containers, this isolation happens at the process level and hence, they are liable for any kind of security incursion. Furthermore, some vital features that are available in the VMs are not available in the containers. For instance, there is no support for system services like SSH. On the other hand, VMs are resource-hungry and hence, their performance gets substantially degraded. Indeed, in containerization parlance, the overhead of a classic hypervisor and a guest operating system will be eliminated to achieve bare metal performance. Therefore, a few VMs can be provisioned and made available to work on a single machine. Thus, on one hand, we have the fully isolated VMs with average performance and on the other side, we have the containers that lack some of the key features, but are blessed with high performance. Having understood the ensuing needs, product vendors are working on system containers. The objective of this new initiative is to provide full system containers with the performance that you would expect from bare metal servers, but with the experience of virtual machines. The system containers in the preceding right-hand-side diagram represent the convergence of two important concepts (virtualization and containerization) for smarter IT. We will hear and read more about this blending in the future.

Containerization technologies

Having recognized the role and the relevance of the containerization paradigm for IT infrastructure augmentation and acceleration, a few technologies that leverage the unique and decisive impacts of the containerization idea have come into existence and they have been enumerated as follows:

- **LXC(Linux Containers)**: Thisis the father of all kinds of containers and it represents an operating-system-level virtualization environment for running multiple isolated Linux systems (containers) on a single Linux machine. The article LXCon the Wikipedia website states that:

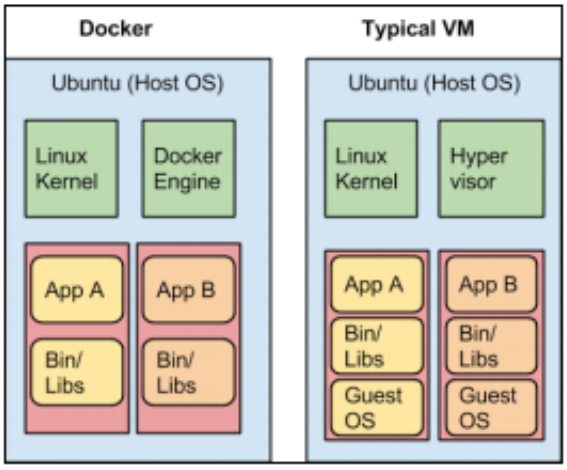
  > *"The Linux kernel provides the cgroups functionality that allows limitation and prioritization of resources (CPU, memory, block I/O, network, etc.) without the need for starting any virtual machines, and namespace isolation functionality that allows complete isolation of an applications' view of the operating environment, including process trees, networking, user IDs and mounted file systems."*

  You can get more information from `http://en.wikipedia.org/wiki/LXC`

- **OpenVZ**: This is an OS-level virtualization technology based on the Linux kernel and the operating system. OpenVZ allows a physical server to run multiple isolated operating system instances, called containers, **virtual private servers** (**VPSs**), or **virtual environments** (**VEs**).
- **The FreeBSD jail**: This is a mechanism that implements an OS-level virtualization, which lets the administrators partition a FreeBSD-based computer system into several independent mini-systems called jails.
- **The AIX Workload partitions(WPARs)**: These are the software implementations of the OS-level virtualization technology, which provide application environment isolation and resource control.
- **Solaris Containers(including Solaris Zones)**: This is an implementation of the OS-level virtualization technology for the x86 and SPARC systems. A Solaris Container is a combination of the system resource controls and boundary separation provided by zones. Zones act as completely isolated virtual servers within a single operating system instance.

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.

Installing Docker

The Docker engine is built on top of the Linux kernel and it extensively leverages its features. Therefore, at this point in time, the Docker engine can only be directly run on Linux OS distributions. Nonetheless, the Docker engine could be run on the Mac and Microsoft Windows operating systems by using the lightweight Linux VMs with the help of adapters, such as Boot2Docker. Due to the surging growing of Docker, it is now being packaged by all major Linux distributions so that they can retain their loyal users as well as attract new users. You can install the Docker engine by using the corresponding packaging tool of the Linux distribution; for example, by using the `apt-get` command for Debian and Ubuntu, and the `yum` command for Red Hat, Fedora, and CentOS. You can look up the instructions for your operating system at `https://docs.docker.com/installation/#installation`.

Note that Docker is called `docker.io` here and just `docker` on other platforms since Ubuntu (and Debian) already has a package named `docker`. Therefore, all the files with the name `docker` are installed as `docker.io`.

Examples are `/usr/bin/docker.io` and `/etc/bash_completion.d/docker.io`.

Installing Docker from the Ubuntu package repository

Docker is supported by Ubuntu from Ubuntu 12.04 onwards. Remember that you still need a 64-bit operating system to run Docker. This section explains the steps involved in installing the Docker engine from the Ubuntu package repository in detail. To install the Ubuntu packaged version, follow these steps:

1. The best practice for installing the Ubuntu packaged version is to begin the installation process by resynchronizing with the Ubuntu package repository. This step will essentially update the package repository to the latest published packages, thus we will ensure that we always get the latest published version by using the command shown here:

   ```
   $ sudo apt-get update
   ```

2. Kick-start the installation by using the following command. This setup will install the Docker engine along with a few more support files, and it will also start the `docker` service instantaneously:

   ```
   $ sudo apt-get install -y docker.io
   ```

That's it! You have now installed Docker onto your system. Remember that the command has been renamed `docker.io`, so you will have to run all Docker commands with `docker.io` instead of `docker`. However, for your convenience, you can create a soft link for `docker.io` called `docker`. This will enable you to execute Docker commands as `docker` instead of `docker.io`. You can do this by using the following command:

```
$ sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker
```

Downloading the example code

The code files for this course are available at `https://github.com/EdwinMoses/Docker-Code`.

Installing the latest Docker using docker.io script

The official distributions might not package the latest version of Docker. In such a case, you can install the latest version of Docker either manually or by using the automated scripts provided by the Docker community.

For installing the latest version of Docker manually, follow these steps:

1. Add the Docker release tool's repository path to your APT sources, as shown here:

   ```
   $ sudo sh -c "echo deb https://get.docker.io/ubuntu \
   docker main > /etc/apt/sources.list.d/docker.list"
   ```

2. Import the Docker release tool's public key by running the following command:

   ```
   $ sudo apt-key adv --keyserver \
   hkp://keyserver.ubuntu.com:80 --recv-keys \
   36A1D7869245C8950F966E92D8576A8BA88D21E9
   ```

3. Resynchronize with the package repository by using the command shown here:

   ```
   $ sudo apt-get update
   ```

4. Install `docker` and then start the `docker` service.

   ```
   $ sudo apt-get install -y lxc-docker
   ```

The `lxc-docker` command will install the Docker image using the name `docker`.

The Docker community has taken a step forward by hiding these details in an automated install script. This script enables the installation of Docker on most of the popular Linux distributions, either through the `curl` command or through the `wget` command, as shown here:

- For the `curl` command:

```
$ sudo curl -sSL https://get.docker.io/ | sh
```

- For the `wget` command:

```
$ sudo wget -qO- https://get.docker.io/ | sh
```

The preceding automated script approach enforces AUFS as the underlying Docker file system. This script probes the AUFS driver, and then installs it automatically if it is not found in the system. In addition, it also conducts some basic tests upon installation for verifying the sanity.

Upgrading Docker

Now that we have Docker installed, we can get going at full steam! There is one problem though: software repositories like APT are usually behind times and often have older versions. Docker is a fast-moving project and a lot has changed in the last few versions. So it is always recommended to have the latest version installed. At the time of writing this, the latest version of Docker was 1.10.0.

To check and download upgrades, all you have to do is to execute this command in a terminal:

```
sudo apt-get update && sudo apt-get upgrade
```

You can upgrade Docker as and when it is updated in the APT repositories. An alternative (and better) method is to build from source. The best way to remain updated is to regularly get the latest version from the public GitHub repository. Traditionally, building software from a source has been painful and done only by people who actually work on the project. This is not so with Docker. From Docker 0.6, it has been possible to build Docker in Docker. This means that upgrading Docker is as simple as building a new version in Docker itself and replacing the binary. Let's see how this is done.

You need to have the following tools installed in a 64-bit Linux machine (VM or bare-metal) to build Docker:

- **Git**: It is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. It is used here to clone the Docker public source code repository. Check out `git-scm.org` for more details.
- **Make**: This utility is a software engineering tool used to manage and maintain computer programs. Make provides most help when the program consists of many component files. A `Makefile` file is used here to kick off the Docker containers in a repeatable and consistent way.

Building Docker from source

To build Docker in Docker, we will first fetch the source code and then run a few `make` commands that will, in the end, create a `docker` binary, which will replace the current binary in the Docker installation path.

Run the following command in your terminal:

```
$ git clone https://git@github.com/dotcloud/docker
```

This command clones the official Docker source code repository from the GitHub repository into a directory named `docker`:

```
$ cd docker
$ sudo make build
```

This will prepare the development environment and install all the dependencies required to create the binary. This might take some time on the first run, so you can go and have a cup of coffee.

If you encounter any errors that you find difficult to debug, you can always go to `#docker` on freenode IRC. The developers and the Docker community are very helpful.

Now we are ready to compile that binary:

```
$ sudo make binary
```

This will compile a binary and place it in the `./bundles/<version>-dev/binary/` directory. And voila! You have a fresh version of Docker ready.

Before replacing your existing binary though, run the tests:

```
$ sudo make test
```

If the tests pass, then it is safe to replace your current binary with the one you've just compiled. Stop the `docker` service, create a backup of the existing binary, and then copy the freshly baked binary in its place:

```
$ sudo service docker stop
$ alias wd='which docker'
$ sudo cp $(wd) $(wd)_
$ sudo cp $(pwd)/bundles/<version>-dev/binary/docker-<version>-dev $(wd)
$ sudo service docker start
```

Congratulations! You now have the up-to-date version of Docker running.

OS X and Windows users can follow the same procedures as SSH in the boot2Docker VM.

User permissions

Create a group called `docker` and add your user to that group to avoid having to add the `sudo` prefix to every `docker` command. The reason you need to run a `docker` command with the `sudo` prefix by default is that the `docker` daemon needs to run with root privileges, but the `docker` client (the commands you run) doesn't. So, by creating a `docker` group, you can run all the client commands without using the `sudo` prefix, whereas the daemon runs with the root privileges:

```
$ sudo groupadd docker # Adds the docker group
$ sudo gpasswd -a $(whoami) docker # Adds the current user to the group
$ sudo service docker restart
```

You might need to log out and log in again for the changes to take effect.

UFW settings

Docker uses a bridge to manage network in the container. **Uncomplicated Firewall** (**UFW**) is the default firewall tool in Ubuntu. It drops all forwarding traffic. You will need to enable forwarding like this:

```
$ sudo vim /etc/default/ufw
# Change:
```

```
# DEFAULT_FORWARD_POLICY="DROP"
# to
DEFAULT_FORWARD_POLICY="ACCEPT"
```

Reload the firewall by running the following command:

```
$ sudo ufw reload
```

Alternatively, if you want to be able to reach your containers from other hosts, then you should enable incoming connections on the Docker port (default `2375`):

```
$ sudo ufw allow 2375/tcp
```

You do not need to follow user permission and UFW settings if you are using boot2Docker.

Installing Docker on Mac OS X

To be able to use Docker on Mac OS X, we have to run the Docker service inside a **virtual machine** (**VM**) since Docker uses Linux-specific features to run. We don't have to get frightened by this since the installation process is very short and straightforward.

Installation

There is an OS X installer that installs everything we need, that is, VirtualBox, boot2docker, and Docker.

VirtualBox is a virtualizer in which we can run the lightweight Linux distribution, and boot2docker is a virtual machine that runs completely in the RAM and occupies just about 27 MB of space.

The latest released version of the OS X installer can be found at `https://github.com/boot2docker/osx-installer/releases/latest`.

Now, let's take a look at how the installation should be done with the following steps:

1. Download the installer by clicking on the button named `Boot2Docker-1.x.0.pkg` to get the `.pkg` file.
2. Double-click on the downloaded `.pkg` file and go through with the installation process.
3. Open the **Finder** window and navigate to your `Applications` folder; locate `boot2docker` and double-click on it. A terminal window will open and issue a few commands.

   This runs a Linux VM, named `boot2docker-vm`, that has Docker pre-installed in VirtualBox. The Docker service in the VM runs in daemon (background) mode, and a Docker client is installed in OS X, which communicates directly with the Docker daemon inside the VM via the Docker Remote API.

4. You will see the following output, which tells you to set some environment variables:

```
To connect the Docker client to the Docker daemon, please set:
    export DOCKER_HOST=tcp://192.168.59.103:2376
    export DOCKER_CERT_PATH=/Users/oscarhane/.boot2docker/certs/
boot2docker-vm
    export DOCKER_TLS_VERIFY=1
```

   We open up the `~/.bash_profile` file and paste three lines from our output, as follows, at the end of this file:

```
export DOCKER_HOST=tcp://192.168.59.103:2376
export.DOCKER_CERT_PATH=/Users/xxx/.boot2docker/certs/boot2docker-
vm
export DOCKER_TLS_VERIFY=1
```

The reason why we do this is so that our Docker client knows where to find the Docker daemon. If you want to find the IP in the future, you can do so by executing the `boot2docker ip` command. Adding the preceding lines will set these variables every time a new terminal session starts. When you're done, close the terminal window. Then, open a new window and type `echo $DOCKER_HOST` to verify that the environment variable is set as it should be. You should see the IP and port your boot2docker VM printed.

5. Type `docker version` to verify that you can use the Docker command, which will show the Docker version currently installed.

Installing Docker on Windows

Just as we have to install a Linux virtual machine when installing Docker in OS X, we have to do the same in Windows in order to run Docker because of the Linux kernel features that Docker builds on. OS X has a native Docker client that directly communicates with the Docker daemon inside the virtual machine, but there isn't one available for Windows yet.

Installation

There is a Windows installer that installs everything we need in order to run Docker. For this, go to `https://github.com/boot2docker/windows-installer/releases/latest`.

Now, let's take a look at how the installation should be done with the help of the following steps:

1. Click on the **docker-install.exe** button to download the `.exe` file.
2. When the download is complete, run the downloaded installer. Follow through the installation process, and you will get VirtualBox, msysGit, and boot2docker installed.
3. Go to your `Program Files` folder and click on the newly installed `boot2docker` to start using Docker. If you are prompted to enter a passphrase, just press *Enter*.
4. Type `docker version` to verify that you can use the Docker command.

Upgrading Docker on Mac OS X and Windows

A new software changes often and to keep `boot2docker` updated, invoke these commands:

```
boot2docker stop
boot2docker download
boot2docker start
```

Downloading the first Docker image

Having installed the Docker engine successfully, the next logical step is to download the images from the Docker registry. The Docker registry is an application repository, which hosts a range of applications that vary between basic Linux images and advanced applications. The `docker pull` subcommand is used for downloading any number of images from the registry. In this section, we will download a tiny version of Linux called the `busybox` image by using the following command:

```
$ sudo docker pull busybox
511136ea3c5a: Pull complete
df7546f9f060: Pull complete
ea13149945cb: Pull complete
4986bf8c1536: Pull complete
busybox:latest: The image you are pulling has been verified. Important: image verification is a tech preview feature and should not be relied on to provide security.
Status: Downloaded newer image for busybox:latest
```

Once the images have been downloaded, they can be verified by using the `docker images` subcommand, as shown here:

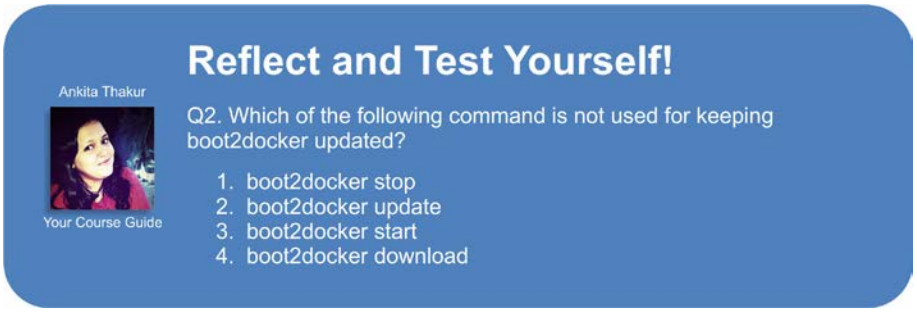
```
$ sudo docker images
REPOSITORY    TAG     IMAGE ID         CREATED      VIRTUAL SIZE
busybox       latest  4986bf8c1536     12 weeks ago 2.433 MB
```

Running the first Docker container

Now, you can start your first Docker container. It is standard practice to start with the basic *Hello World!* application. In the following example, we will echo `Hello World!` by using a `busybox` image, which we have already downloaded, as shown here:

```
$ sudo docker run busybox echo "Hello World!"
"Hello World!"
```

Cool, isn't it? You have set up your first Docker container in no time. In the preceding example, the `docker run` subcommand has been used for creating a container and for printing `Hello World!` by using the `echo` command.



Running a Docker container on Amazon Web Services

Amazon Web Services (**AWS**) announced the availability of Docker containers at the beginning of 2014, as a part of its Elastic Beanstalk offering. At the end of 2014, they revolutionized Docker deployment and provided the users with options shown here for running Docker containers:

- The Amazon EC2 container service (only available in **preview** mode at the time of writing this book)
- Docker deployment by using the Amazon Elastic Beans services

The Amazon EC2 container service lets you start and stop the container-enabled applications with the help of simple API calls. AWS has introduced the concept of a cluster for viewing the state of your containers. You can view the tasks from a centralized service, and it gives you access to many familiar Amazon EC2 features, such as the security groups, the EBS volumes and the IAM roles.

Please note that this service is still not available in the AWS console. You need to install AWS CLI on your machine to deploy, run, and access this service.

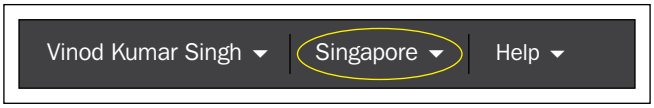
The AWS Elastic Beanstalk service supports the following:

- A single container that supports Elastic Beanstalk by using a console. Currently, it supports the PHP and Python applications.
- A single container that supports Elastic Beanstalk by using a command line tool called *eb*. It supports the same PHP and Python applications.
- Use of multiple container environments by using Elastic beanstalk.

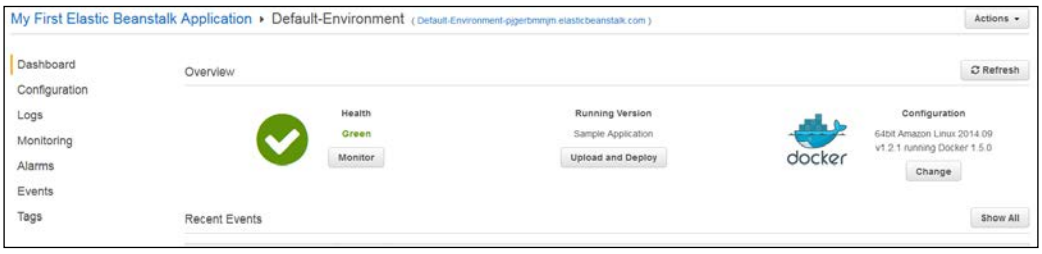
Currently, AWS supports the latest Docker version, which is 1.5.

This section provides a step-by-step process to deploy a sample application on a Docker container running on AWS Elastic Beanstalk.The following are the steps of deployment:

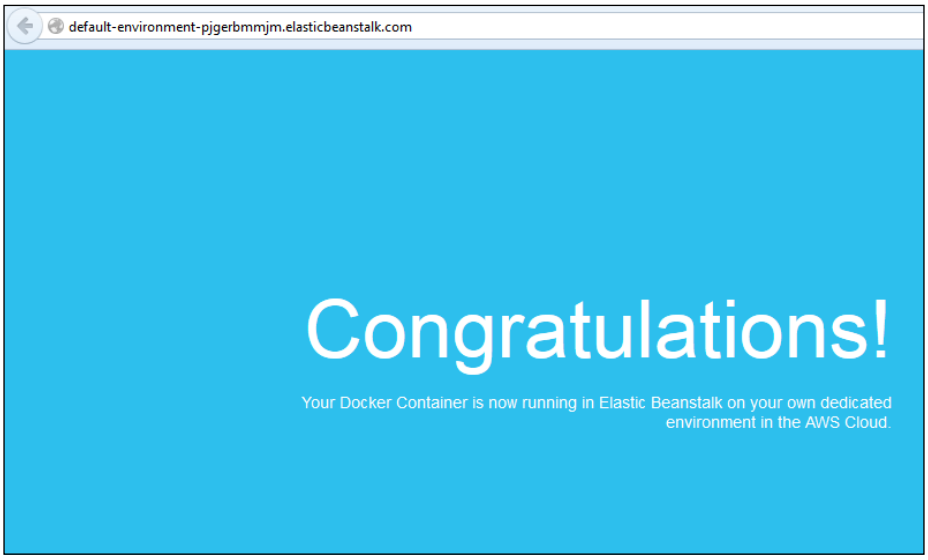
1. Log in to the AWS Elastic Beanstalk console by using this `https://console.aws.amazon.com/elasticbeanstalk/` URL.
2. Select a region where you want to deploy your application, as shown here:



3. Select the **Docker** option, which is in the drop down menu, and then click on **Launch Now**. The next screen will be shown after a few minutes, as shown here:



Now, click on the URL that is next to **Default-Environment (Default-Environment-pjgerbmmjm.elasticbeanstalk.com)**, as shown here:



Troubleshooting

Most of the time, you will not encounter any issues when installing Docker. However, unplanned failures might occur. Therefore, it is necessary to discuss prominent troubleshooting techniques and tips. Let's begin by discussing the troubleshooting knowhow in this section. The first tip is that the running status of Docker should be checked by using the following command:

```
$ sudo service docker status
```

However, if Docker has been installed by using the Ubuntu package, then you will have to use `docker.io` as the service name. If the `docker` service is running, then this command will print the status as `start/running` along with its process ID.

If you are still experiencing issues with the Docker setup, then you could open the Docker log by using the `/var/log/upstart/docker.log` file for further investigation.

Your Coding Challenge

Ankita Thakur

Your Course Guide

Now that we have covered all the basics of controlling your boot2docker VM, you can explore some other way to run Docker containers on your local machine. 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! Try it out.

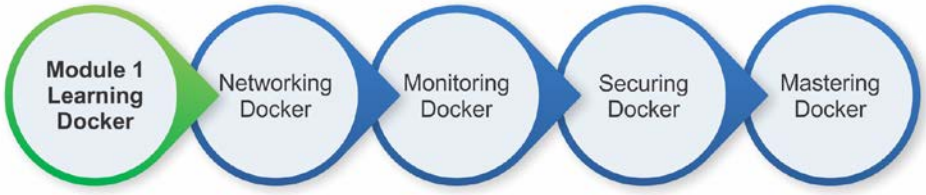
Summary of Module 1 Chapter 1

Ankita Thakur

Your Course Guide

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. Containerization is going to be a dominant and decisive paradigm for the enterprise as well as cloud IT environments in the future because of its hitherto unforeseen automation and acceleration capabilities. There are several mechanisms in place for taking the containerization movement to greater heights. However, Docker has zoomed ahead of everyone in this hot race, and it has successfully decimated the previously-elucidated barriers.

Your Progress through the Course So Far



2
Up and Running

In the last chapter, we set up Docker in our development setup. In this chapter, we will explore the Docker command-line interface.

The following topics will be covered:

- Docker terminologies
- Docker commands
- Dockerfiles
- Docker workflow—pull-use-modify-commit-push workflow

Docker terminologies

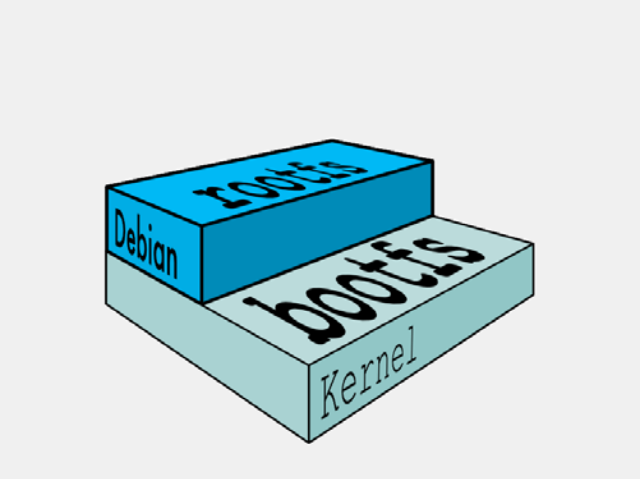
Before we begin our exciting journey into the Docker sphere, let's understand the Docker terminologies that will be used in this book a little better. Very similar in concept to VM images, a Docker image is a snapshot of a system. The difference between a VM image and a Docker image is that a VM image can have running services, whereas a Docker image is just a filesystem snapshot, which means that while you can configure the image to have your favorite packages, you can run only one command in the container. Don't fret though, since the limitation is one command, not one process, so there are ways to get a Docker container to do almost anything a VM instance can.

Docker has also implemented a Git-like distributed version management system for Docker images. Images can be stored in repositories (called a registry), both locally and remotely. The functionalities and terminologies borrow heavily from Git—snapshots are called commits, you pull an image repository, you push your local image to a repository, and so on.

Docker images and containers

A **Docker image** is a collection of all of the files that make up a software application. Each change that is made to the original image is stored in a separate layer. To be precise, any Docker image has to originate from a base image according to the various requirements. Additional modules can be attached to the base image for deriving the various images that can exhibit the preferred behavior. Each time you commit to a Docker image, you are creating a new layer on the Docker image, but the original image and each pre-existing layer remains unchanged. In other words, images are typically of the read-only type. If they are empowered through the systematic attachment of newer modules, then a fresh image will be created with a new name. The Docker images are turning out to be a viable base for developing and deploying the Docker containers.

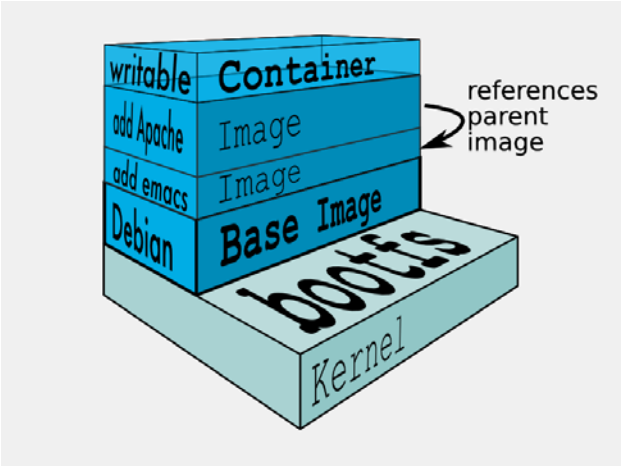
A base image has been illustrated here. Debian is the base image, and a variety of desired capabilities in the form of functional modules can be incorporated on the base image for arriving at multiple images:



Every image has a unique `ID`, as explained in the following section. The base images can be enhanced such that they can create the parent images, which in turn can be used for creating the child images. The base image does not have any parent, that is, the parent images sit on top of the base image. When we work with an image and if we don't specify that image through an appropriate identity (say, a new name), then the `latest` image (recently generated) will always be identified and used by the Docker engine.

As per the Docker home page, a Docker image has a read-only template. For example, an image could contain an Ubuntu operating system, with Apache and your web application installed on it. Docker provides a simple way for building new images or of updating the existing images. You can also download the Docker images that the other people have already created. The Docker images are the building components of the Docker containers. In general, the base Docker image represents an operating system, and in the case of Linux, the base image can be one of its distributions, such as Debian. Adding additional modules to the base image ultimately dawns a container. The easiest way of thinking about a container is as the read-write layer that sits on more read-only images. When the container is run, the Docker engine not only merges all of the required images together, but it also merges the changes from the read-write layer into the container itself. This makes it a self-contained, extensible, and executable system. The changes can be merged by using the Docker `docker commit` subcommand. The new container will accommodate all the changes that are made to the base image. The new image will form a new layer on top of the base image.

The following diagram will tell you everything clearly. The base image is the **Debian** distribution, then there is an addition of two images (the **emacs** and the **Apache** server), and this will result in the container:



Each commit invariably makes a new image. This makes the number of images go up steadily, and so managing them becomes a complicated affair. However, the storage space is not a big challenge because the new image that is generated is only comprised of the newly added modules. In a way, this is similar to the popular object storage in the cloud environments. Every time you update an object, there will be a new object that gets created with the latest modification and then it is stored with a new `ID`. In the case of object storage, the storage size balloons significantly.

A Docker layer

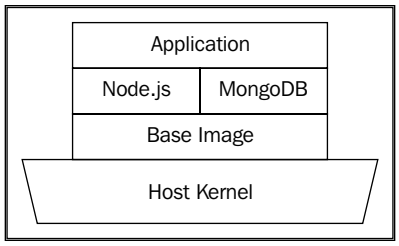
A **Docker layer** could represent either read-only images or read-write images. However, the top layer of a container stack is always the read-write (writable) layer, which hosts a Docker container.

A Docker container

From the preceding diagram, it is clear that the read-write layer is the container layer. There could be several read-only images beneath the container layer. Typically, a container originates from a read-only image through the act of a `commit`. When you `start` a container, you actually refer to an image through its unique `ID`. Docker pulls the required image and its parent image. It continues to pull all the parent images until it reaches the base image.

A Docker container can be correlated to an instance of a VM. It runs sandboxed processes that share the same kernel as the host. The term **container** comes from the concept of shipping containers. The idea is that you can ship containers from your development environment to the deployment environment and the applications running in the containers will behave the same way no matter where you run them.

The following image shows the layers of AUFS:



This is similar in context to a shipping container, which stays sealed until delivery but can be loaded, unloaded, stacked, and transported in between.

The visible filesystem of the processes in the container is based on AUFS (although you can configure the container to run with a different filesystem too). AUFS is a layered filesystem. These layers are all read-only and the merger of these layers is what is visible to the processes. However, if a process makes a change in the filesystem, a new layer is created, which represents the difference between the original state and the new state. When you create an image out of this container, the layers are preserved. Thus, it is possible to build new images out of existing images, creating a very convenient hierarchical model of images.

The docker daemon

The `docker` daemon is the process that manages containers. It is easy to get this confused with the Docker client because the same binary is used to run both the processes. The `docker` daemon, though, needs the `root` privileges, whereas the client doesn't.

Unfortunately, since the `docker` daemon runs with root privileges, it also introduces an attack vector. Read `https://docs.Docker.com/articles/security/` for more details.

Docker client

The Docker client is what interacts with the `docker` daemon to start or manage containers. Docker uses a RESTful API to communicate between the client and the daemon.

REST is an architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements within a distributed hypermedia system. In plain words, a RESTful service works over standard HTTP methods such as the `GET`, `POST`, `PUT`, and `DELETE` methods.

Dockerfile

A Dockerfile is a file written in a **Domain Specific Language** (**DSL**) that contains instructions on setting up a Docker image. Think of it as a Makefile equivalent of Docker.

Let's take a look at the following example:

```
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"]
```

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.

Docker repository

A Docker repository is a namespace that is used for storing a Docker image. For instance, if your app is named `helloworld` and your username or namespace for the Registry is `thedockerbook` then, in the Docker Repository, where this image would be stored in the Docker Registry would be named `thedockerbook/helloworld`.

The base images are stored in the Docker Repository. The base images are the fountainheads for realizing the bigger and better images with the help of a careful addition of new modules. The child images are the ones that have their own parent images. The base image does not have any parent image. The images sitting on a base image are named as parent images because the parent images bear the child images.

Docker commands

Now let's get our hands dirty on the Docker CLI. We will look at the most common commands and their use cases. The Docker commands are modeled after Linux and Git, so if you have used either of these, you will find yourself at home with Docker.

Only the most commonly used options are mentioned here. For the complete reference, you can check out the official documentation at `https://docs.docker.com/reference/commandline/cli/`.

The daemon command

If you have installed the `docker` daemon through standard repositories, the command to start the `docker` daemon would have been added to the `init` script to automatically start as a service on startup. Otherwise, you will have to first run the `docker` daemon yourself for the client commands to work.

Now, while starting the daemon, you can run it with arguments that control the **Domain Name System** (**DNS**) configurations, storage drivers, and execution drivers for the containers:

```
$ export DOCKER_HOST="tcp://0.0.0.0:2375"
$ Docker -d -D -e lxc -s btrfs --dns 8.8.8.8 --dns-search example.com
```

You'll need these only if you want to start the daemon yourself. Otherwise, you can start the `docker` daemon with `$ sudo service Docker start`. For OS X and Windows, you need to run the commands mentioned in *Chapter 1*, *Installing Docker*.

The following table describes the various flags:

Flag	Explanation
`-d`	This runs Docker as a daemon.
`-D`	This runs Docker in debug mode.
`-e [option]`	This is the execution driver to be used. The default execution driver is native, which uses `libcontainer`.
`-s [option]`	This forces Docker to use a different storage driver. The default value is "", for which Docker uses AUFS.
`--dns [option(s)]`	This sets the DNS server (or servers) for all Docker containers.
`--dns-search [option(s)]`	This sets the DNS search domain (or domains) for all Docker containers.
`-H [option(s)]`	This is the socket (or sockets) to bind to. It can be one or more of `tcp://host:port, unix:///path/to/socket, fd://* or fd://socketfd`.

If multiple `docker` daemons are being simultaneously run, the client honors the value set by the `DOCKER_HOST` parameter. You can also make it connect to a specific daemon with the `-H` flag.

Consider this command:

```
$ docker -H tcp://0.0.0.0:2375 run -it ubuntu /bin/bash
```

The preceding command is the same as the following command:

```
$ DOCKER_HOST="tcp://0.0.0.0:2375" docker run -it ubuntu /bin/bash
```

The version command

The `version` command prints out the version information:

```
$ docker -v
Docker version 1.1.1, build bd609d2
```

The info command

The `info` command prints the details of the `docker` daemon configuration such as the execution driver, the storage driver being used, and so on:

```
$ docker info # Running it in boot2docker on OS X
Containers: 0
Images: 0
Storage Driver: aufs
 Root Dir: /mnt/sda1/var/lib/docker/aufs
 Dirs: 0
Execution Driver: native-0.2
Kernel Version: 3.15.3-tinycore64
Debug mode (server): true
Debug mode (client): false
Fds: 10
Goroutines: 10
EventsListeners: 0
Init Path: /usr/local/bin/docker
Sockets: [unix:///var/run/docker.sock tcp://0.0.0.0:2375]
```

The run command

The run command is the command that we will be using most frequently. It is used to run Docker containers:

```
$ docker run [options] IMAGE [command] [args]
```

Flags	Explanation
`-a, --attach=[]`	Attach to the `stdin`, `stdout`, or `stderr` files (standard input, output, and error files.).
`-d, --detach`	This runs the container in the background.
`-i, --interactive`	This runs the container in interactive mode (keeps the `stdin` file open).
`-t, --tty`	This allocates a pseudo `tty` flag (which is required if you want to attach to the container's terminal).
`-p, --publish=[]`	This publishes a container's port to the host (`ip:hostport:containerport`).
`--rm`	This automatically removes the container when exited (it cannot be used with the `-d` flag).

Flags	Explanation
`--privileged`	This gives additional privileges to this container.
`-v, --volume=[]`	This bind mounts a volume (from host => `/host:/container`; from docker => `/container`).
`--volumes-from=[]`	This mounts volumes from specified containers.
`-w, --workdir=""`	This is the working directory inside the container.
`--name=""`	This assigns a name to the container.
`-h, --hostname=""`	This assigns a hostname to the container.
`-u, --user=""`	This is the username or UID the container should run on.
`-e, --env=[]`	This sets the environment variables.
`--env-file=[]`	This reads environment variables from a new line-delimited file.
`--dns=[]`	This sets custom DNS servers.
`--dns-search=[]`	This sets custom DNS search domains.
`--link=[]`	This adds link to another container (`name:alias`).
`-c, --cpu-shares=0`	This is the relative CPU share for this container.
`--cpuset=""`	These are the CPUs in which to allow execution; starts with 0. (For example, 0 to 3).
`-m, --memory=""`	This is the memory limit for this container (`<number><b|k|m|g>`).
`--restart=""`	(v1.2+) This specifies a restart policy in case the container crashes.
`--cap-add=""`	(v1.2+) This grants a capability to a container (refer to *Chapter 4, Security Best Practices*).
`--cap-drop=""`	(v1.2+) This blacklists a capability to a container (refer to *Chapter 4, Security Best Practices*).
`--device=""`	(v1.2+) This mounts a device on a container.

While running a container, it is important to keep in mind that the container's lifetime is associated with the lifetime of the command you run when you start the container. Now try to run this:

```
$ docker run -dt ubuntu ps
b1d037dfcff6b076bde360070d3af0d019269e44929df61c93dfcdfaf29492c9
$ docker attach b1d037
2014/07/16 16:01:29 You cannot attach to a stopped container, start
it first
```

What happened here? When we ran the simple command, `ps`, the container ran the command and exited. Therefore, we got an error.

The `attach` command attaches the standard input and output to a running container.

Another important piece of information here is that you don't need to use the whole 64-character ID for all the commands that require the container ID. The first couple of characters are sufficient. With the same example as shown in the following code:

```
$ docker attach b1d03
2014/07/16 16:09:39 You cannot attach to a stopped container, start
it first
$ docker attach b1d0
2014/07/16 16:09:40 You cannot attach to a stopped container, start
it first
$ docker attach b1d
2014/07/16 16:09:42 You cannot attach to a stopped container, start
it first
$ docker attach b1
2014/07/16 16:09:44 You cannot attach to a stopped container, start
it first
$ docker attach b
2014/07/16 16:09:45 Error: No such container: b
```

A more convenient method though would be to name your containers yourself:

```
$ docker run -dit --name OD-name-example ubuntu /bin/bash
1b21af96c38836df8a809049fb3a040db571cc0cef000a54ebce978c1b5567ea
$ docker attach OD-name-example
root@1b21af96c388:/#
```

The `-i` flag is necessary to have any kind of interaction in the container, and the `-t` flag is necessary to create a pseudo-terminal.

The previous example also made us aware of the fact that even after we exit a container, it is still in a `stopped` state. That is, we will be able to start the container again, with its filesystem layer preserved. You can see this by running the following command:

```
$ docker ps -a
CONTAINER ID IMAGE          COMMAND CREATED    STATUS    NAMES
eb424f5a9d3f ubuntu:latest ps      1 hour ago Exited OD-name-example
```

While this can be convenient, you may pretty soon have your host's disk space drying up as more and more containers are saved. So, if you are going to run a disposable container, you can run it with the `--rm` flag, which will remove the container when the process exits:

```
$ docker run --rm -it --name OD-rm-example ubuntu /bin/bash
root@0fc99b2e35fb:/# exit
exit
$ docker ps -a
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

Running a server

Now, for our next example, we'll try running a web server. This example is chosen because the most common practical use case of Docker containers is the shipping of web applications:

```
$ docker run -it --name OD-pythonserver-1 --rm python:2.7 \
python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000
```

Now we know the problem; we have a server running in a container, but since the container's IP is assigned by Docker dynamically, it makes things difficult. However, we can bind the container's ports to the host's ports and Docker will take care of forwarding the networking traffic. Now let's try this command again with the `-p` flag:

```
$ docker run -p 0.0.0.0:8000:8000 -it --rm --name OD-pythonserver-2 \
python:2.7 python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000 ...
172.17.42.1 - - [18/Jul/2014 14:25:46] "GET / HTTP/1.1" 200 -
```

Now open your browser and go to `http://localhost:8000`. Voilà!

If you are an OS X user and you realize that you are not able to access `http://localhost:8000`, it is because VirtualBox hasn't been configured to respond to **Network Address Translation** (**NAT**) requests to the boot2Docker VM. Adding the following function to your aliases file (`bash_profile` or `.bashrc`) will save a lot of trouble:

```
natboot2docker () {
   VBoxManage controlvm boot2docker-vm natpf1 \
```

```
      "$1,tcp,127.0.0.1,$2,,$3";
  }


  removeDockerNat() {
      VBoxManage modifyvm boot2docker-vm \
      --natpf1 delete $1;
  }
```

After this, you should be able to use the `$ natboot2docker mypythonserver 8000 8000` command to be able to access the Python server. But remember to run the `$ removeDockerDockerNat mypythonserver` command when you are done. Otherwise, when you run the boot2Docker VM next time, you will be faced with a bug that won't allow you to get the IP address or the `ssh` script into it:

```
$ boot2docker ssh
ssh_exchange_identification: Connection closed by remote host
2014/07/19 11:55:09 exit status 255
```

Your browser now shows the `/root` path of the container. What if you wanted to serve your host's directories? Let's try mounting a device:

```
root@eb53f7ec79fd:/# mount -t tmpfs /dev/random /mnt
mount: permission denied
```

As you can see, the `mount` command doesn't work. In fact, most kernel capabilities that are potentially dangerous are dropped, unless you include the `--privileged` flag.

However, you should never use this flag unless you know what you are doing. Docker provides a much easier way to bind mount host volumes and bind mount host volumes with the `-v` and `–volumes` options. Let's try this example again in the directory we are currently in:

```
$ docker run -v $(pwd):$(pwd) -p 0.0.0.0:8000:8000 -it –rm \
--name OD-pythonserver-3 python:2.7 python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000 ...
10.0.2.2 - - [18/Jul/2014 14:40:35] "GET / HTTP/1.1" 200 -
```

You have now bound the directory you are running the commands from to the container. However, when you access the container, you still get the directory listing of the root of the container. To serve the directory that has been bound to the container, let's set it as the working directory of the container (the directory the containerized process runs in) using the `-w` flag:

```
$ docker run -v $(pwd):$(pwd) -w $(pwd) -p 0.0.0.0:8000:8000 -it \ --name
OD-pythonserver-4 python:2.7 python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000 ...
10.0.2.2 - - [18/Jul/2014 14:51:35] "GET / HTTP/1.1" 200 -
```

Boot2Docker users will not be able to utilize this yet, unless you use guest additions and set up shared folders, the guide to which can be found at `https://medium.com/boot2docker-lightweight-linux-for-docker/boot2docker-together-with-virtualbox-guest-additions-da1e3ab2465c`. Though this solution works, it is a hack and is not recommended. Meanwhile, the Docker community is actively trying to find a solution (check out issue `#64` in the boot2Docker GitHub repository and `#4023` in the Docker repository).

Now `http://localhost:8000` will serve the directory you are currently running in, but from a Docker container. Take care though, because any changes you make are written into the host's filesystem as well.

Since v1.1.1, you can bind mount the root of the host to a container using `$ docker run -v /:/my_host:ro ubuntu ls /my_host`, but mounting on the `/` path of the container is forbidden.

The volume can be optionally suffixed with the `:ro` or `:rw` commands to mount the volumes in read-only or read-write mode, respectively. By default, the volumes are mounted in the same mode (read-write or read-only) as they are in the host.

This option is mostly used to mount static assets and to write logs.

But what if I want to mount an external device?

Before v1.2, you had to mount the device in the host and bind mount using the `-v` flag in a privileged container, but v1.2 has added a `--device` flag that you can use to mount a device without needing to use the `--privileged` flag.

For example, to use the webcam in your container, run this command:

```
$ docker run --device=/dev/video0:/dev/video0
```

Docker v1.2 also added a `--restart` flag to specify a restart policy for containers. Currently, there are three restart policies:

- `no`: Do not restart the container if it dies (default).
- `on-failure`: Restart the container if it exits with a non-zero exit code. It can also accept an optional maximum restart count (for example, `on-failure:5`).
- `always`: Always restart the container no matter what exit code is returned.

The following is an example to restart endlessly:

```
$ docker run --restart=always code.it
```

The next line is used to try five times before giving up:

```
$ docker run --restart=on-failure:5 code.it
```

The search command

The `search` command allows us to search for Docker images in the public registry. Let's search for all images related to Python:

```
$ docker search python | less
```

The pull command

The `pull` command is used to pull images or repositories from a registry. By default, it pulls them from the public Docker registry, but if you are running your own registry, you can pull them from it too:

```
$ docker pull python # pulls repository from Docker Hub
$ docker pull python:2.7 # pulls the image tagged 2.7
$ docker pull <path_to_registry>/<image_or_repository>
```

The start command

We saw when we discussed `docker run` that the container state is preserved on exit unless it is explicitly removed. The `docker start` command starts a stopped container:

```
$ docker start [-i] [-a] <container(s)>
```

The stop command

The `stop` command stops a running container by sending the `SIGTERM` signal and then the `SIGKILL` signal after a grace period:

`SIGTERM` and `SIGKILL` are Unix signals. A signal is a form of interprocess communication used in Unix, Unix-like, and other POSIX-compliant operating systems. `SIGTERM` signals the process to terminate. The `SIGKILL` signal is used to forcibly kill a process.

```
docker run -dit --name OD-stop-example ubuntu /bin/bash
$ docker ps
CONTAINER ID IMAGE         COMMAND   CREATED  STATUS    NAMES
679ece6f2a11 ubuntu:latest /bin/bash 5h ago   Up 3s   OD-stop-example
$ docker stop OD-stop-example
OD-stop-example
$ docker ps
CONTAINER ID IMAGE         COMMAND   CREATED  STATUS    NAMES
```

You can also specify the `-t` flag or `--time` flag, which allows you to set the wait time.

The restart command

The `restart` command restarts a running container:

```
$ docker run -dit --name OD-restart-example ubuntu /bin/bash
$ sleep 15s # Suspends execution for 15 seconds
$ docker ps
CONTAINER ID IMAGE         COMMAND   STATUS    NAMES
cc5d0ae0b599 ubuntu:latest /bin/bash Up 20s    OD-restart-example

$ docker restart OD-restart-example
$ docker ps
CONTAINER ID IMAGE         COMMAND   STATUS    NAMES
cc5d0ae0b599 ubuntu:latest /bin/bash Up 2s    OD-restart-example
```

If you observe the status, you will notice that the container was rebooted.

The rm command

The `rm` command removes Docker containers:

```
$ Docker ps -a # Lists containers including stopped ones
CONTAINER ID  IMAGE  COMMAND   CREATED  STATUS NAMES
cc5d0ae0b599  ubuntu /bin/bash 6h ago   Exited OD-restart-example
679ece6f2a11  ubuntu /bin/bash 7h ago   Exited OD-stop-example
e3c4b6b39cff  ubuntu /bin/bash 9h ago   Exited OD-name-example
```

We seem to be having a lot of containers left over after our adventures. Let's remove one of them:

```
$ dockerDocker rm OD-restart-example
cc5d0ae0b599
```

We can also combine two Docker commands. Let's combine the `docker ps -a -q` command, which prints the ID parameters of the containers in the `docker ps -a`, and `docker rm` commands, to remove all containers in one go:

```
$ docker rm $(docker ps -a -q)
679ece6f2a11
e3c4b6b39cff
$ docker ps -a
CONTAINER ID    IMAGE    COMMAND     CREATED    STATUS      NAMES
```

This evaluates the `docker ps -a -q` command first, and the output is used by the `docker rm` command.

The ps command

The `ps` command is used to list containers. It is used in the following way:

```
$ docker ps [option(s)]
```

Flag	Explanation
`-a, --all`	This shows all containers, including stopped ones.
`-q, --quiet`	This shows only container ID parameters.
`-s, --size`	This prints the sizes of the containers.
`-l, --latest`	This shows only the latest container (including stopped containers).
`-n=""`	This shows the last *n* containers (including stopped containers). Its default value is -1.

Flag	Explanation
`--before=""`	This shows the containers created before the specified ID or name. It includes stopped containers.
`--after=""`	This shows the containers created after the specified ID or name. It includes stopped containers.

The `docker ps` command will show only running containers by default. To see all containers, run the `docker ps -a` command. To see only container ID parameters, run it with the `-q` flag.

The logs command

The `logs` command shows the logs of the container:

```
Let us look at the logs of the python server we have been running
$ docker logs OD-pythonserver-4
Serving HTTP on 0.0.0.0 port 8000 ...
10.0.2.2 - - [18/Jul/2014 15:06:39] "GET / HTTP/1.1" 200 -
^CTraceback (most recent call last):
  File ...
  ...
KeyboardInterrupt
```

You can also provide a `--tail` argument to follow the output as the container is running.

The inspect command

The `inspect` command allows you to get the details of a container or an image. It returns those details as a JSON array:

```
$ Docker inspect ubuntu # Running on an image
[{
    "Architecture": "amd64",
    "Author": "",
    "Comment": "",
     .......
     .......
     .......
    "DockerVersion": "0.10.0",
```

```
    "Id":
"e54ca5efa2e962582a223ca9810f7f1b62ea9b5c3975d14a5da79d3bf6020f37",
    "Os": "linux",
    "Parent":
"6c37f792ddacad573016e6aea7fc9fb377127b4767ce6104c9f869314a12041e",
    "Size": 178365
}]
```

Similarly, for a container we run the following command:

```
$ Docker inspect OD-pythonserver-4 # Running on a container
[{
    "Args": [
        "-m",
        "SimpleHTTPServer",
        "8000"
    ],
    ......
    ......
    "Name": "/OD-pythonserver-4",
    "NetworkSettings": {
        "Bridge": "Docker0",
        "Gateway": "172.17.42.1",
        "IPAddress": "172.17.0.11",
        "IPPrefixLen": 16,
        "PortMapping": null,
        "Ports": {
            "8000/tcp": [
                {
                    "HostIp": "0.0.0.0",
                    "HostPort": "8000"
                }
            ]
        }
    },
    ......
    ......
    "Volumes": {
```

```
        "/home/Docker": "/home/Docker"
    },
    "VolumesRW": {
        "/home/Docker": true
    }
}]
```

Docker inspect provides all of the low-level information about a container or image. In the preceding example, find out the IP address of the container and the exposed port and make a request to the `IP:port`. You will see that you are directly accessing the server running in the container.

However, manually looking through the entire JSON array is not optimal. So the `inspect` command provides a flag, `-f` (or the `--format` flag), which allows you to specify exactly what you want using `Go` templates. For example, if you just want to get the container's IP address, run the following command:

```
$ docker inspect -f  '{{.NetworkSettings.IPAddress}}' \
OD-pythonserver-4;
172.17.0.11
```

The `{{.NetworkSettings.IPAddress}}` is a `Go` template that was executed over the JSON result. `Go` templates are very powerful, and some of the things that you can do with them have been listed at `http://golang.org/pkg/text/template/`.

The top command

The `top` command shows the running processes in a container and their statistics, mimicking the Unix `top` command.

Let's download and run the `ghost` blogging platform and check out what processes are running in it:

```
$ docker run -d -p 4000:2368 --name OD-ghost dockerfile/ghost
ece88c79b0793b0a49e3d23e2b0b8e75d89c519e5987172951ea8d30d96a2936

$ docker top OD-ghost-1
PID                 USER                COMMAND
1162                root                bash /ghost-start
1180                root                npm
1186                root                sh -c node index
1187                root                node index
```

Yes! We just set up our very own `ghost` blog, with just one command. This brings forth another subtle advantage and shows something that could be a future trend. Every tool that exposes its services through a TCP port can now be containerized and run in its own sandboxed world. All you need to do is expose its port and bind it to your host port. You don't need to worry about installations, dependencies, incompatibilities, and so on, and the uninstallation will be clean because all you need to do is stop all the containers and remove the image.

Ghost is an open source publishing platform that is beautifully designed, easy to use, and free for everyone. It is coded in Node.js, a server-side JavaScript execution engine.

The attach command

The `attach` command attaches to a running container.

Let's start a container with Node.js, running the node interactive shell as a daemon, and later attach to it.

Node.js is an event-driven, asynchronous I/O web framework that runs applications written in JavaScript on Google's V8 runtime environment.

The container with Node.js is as follows:

```
$ docker run -dit --name OD-nodejs shykes/nodejs node
8e0da647200efe33a9dd53d45ea38e3af3892b04aa8b7a6e167b3c093e522754

$ docker attach OD-nodejs
console.log('Docker rocks!');Docker rocks!
```

The kill command

The `kill` command kills a container and sends the `SIGTERM` signal to the process running in the container:

```
Let us kill the container running the ghost blog.
$ docker kill OD-ghost-1
OD-ghost-1

$ docker attach OD-ghost-1 # Verification
```

```
2014/07/19 18:12:51 You cannot attach to a stopped container, start
it first
```

The cp command

The `cp` command copies a file or folder from a container's filesystem to the host path. Paths are relative to the root of the filesystem.

It's time to have some fun. First, let's run an Ubuntu container with the `/bin/bash` command:

```
$ docker run -it –name OD-cp-bell ubuntu /bin/bash
```

Now, inside the container, let's create a file with a special name:

```
# touch $(echo -e '\007')
```

The `\007` character is an ASCII `BEL` character that rings the system bell when printed on a terminal. You might have already guessed what we're about to do. So let's open a new terminal and execute the following command to copy this newly created file to the host:

```
$ docker cp OD-cp-bell:/$(echo -e '\007') $(pwd)
```

For the `docker cp` command to work, both the container path and the host path must be complete, so do not use shortcuts such as `.`, `,`, `*`, and so on.

So we created an empty file whose filename is the `BEL` character, in a container. Then we copied the file to the current directory in the host container. Just one last step is remaining. In the host tab where you executed the `docker cp` command, run the following command:

```
$ echo *
```

You will hear the system bell ring! We could have copied any file or directory from the container to the host. But it doesn't hurt to have some fun!

If you found this interesting, you might like to read `http://www.dwheeler.com/essays/fixing-unix-linux-filenames.html`. This is a great essay that discusses the edge cases in filenames, which can cause simple to complicated issues in a program.

The port command

The `port` command looks up the public-facing port that is bound to an exposed port in the container:

```
$ docker port CONTAINER PRIVATE_PORT
$ docker port OD-ghost 2368

4000
```

Ghost runs a server at the `2368` port that allows you to write and publish a blog post. We bound a host port to the `OD-ghost` container's port `2368` in the example for the `top` command.

Running your own project

By now, we are considerably familiar with the basic Docker commands. Let's up the ante. For the next couple of commands, I am going to use one of my side projects. Feel free to use a project of your own.

Let's start by listing out our requirements to determine the arguments we must pass to the `docker run` command.

Our application is going to run on Node.js, so we will choose the well-maintained `dockerfile/nodejs` image to start our base container:

- We know that our application is going to bind to port `8000`, so we will expose the port to `8000` of the host.
- We need to give a descriptive name to the container so that we can reference it in future commands. In this case, let's choose the name of the application:

  ```
  $ docker run -it --name code.it dockerfile/nodejs /bin/bash
  [ root@3b0d5a04cdcd:/data ]$ cd /home
  [ root@3b0d5a04cdcd:/home ]$
  ```

Once you have started your container, you need to check whether the dependencies for your application are already available. In our case, we only need Git (apart from Node.js), which is already installed in the `dockerfile/nodejs` image.

Now that our container is ready to run our application, all that is remaining is for us to fetch the source code and do the necessary setup to run the application:

```
$ git clone https://github.com/shrikrishnaholla/code.it.git
$ cd code.it && git submodule update --init --recursive
```

This downloads the source code for a plugin used in the application.

Then run the following command:

```
$ npm install
```

Now all the node modules required to run the application are installed.

Next, run this command:

```
$ node app.js
```

Now you can go to `localhost:8000` to use the application.

The diff command

The `diff` command shows the difference between the container and the image it is based on. In this example, we are running a container with `code.it`. In a separate tab, run this command:

```
$ docker diff code.it
C /home
A /home/code.it
...
```

The commit command

The `commit` command creates a new image with the filesystem of the container. Just as with Git's `commit` command, you can set a commit message that describes the image:

```
$ docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]
```

Flag	Explanation
`-p, --pause`	This pause the container during commit (availabe from v1.1.1+ onwards).
`-m, --message=""`	This is a commit message. It can be a description of what the image does.
`-a, --author=""`	This displays the author details.

For example, let's use this command to commit the container we have set up:

```
$ docker commit -m "Code.it – A browser based text editor and
interpreter" -a "Shrikrishna Holla <s**a@gmail.com>" code.it
shrikrishna/code.it:v1
```

Replace the author details and the username portion of the image name in this example if you are copying these examples.

The output will be a lengthy image ID. If you look at the command closely, we have named the image `shrikrishna/code.it:v1`. This is a convention. The first part of an image/repository's name (before the forward slash) is the Docker Hub username of the author. The second part is the intended application or image name. The third part is a tag (usually a version description) separated from the second part by a colon.

`Docker Hub` is a public registry maintained by Docker, Inc. It hosts public Docker images and provides services to help you build and manage your Docker environment. More details about it can be found at `https://hub.docker.com`.

A collection of images tagged with different versions is a repository. The image you create by running the `docker commit` command will be a local one, which means that you will be able to run containers from it but it won't be available publicly. To make it public or to push to your private Docker registry, use the `docker push` command.

The images command

The `images` command lists all the images in the system:

```
$ docker images [OPTIONS] [NAME]
```

Flag	Explanation
`-a, --all`	This shows all images, including intermediate layers.
`-f, --filter=[]`	This provides filter values.
`--no-trunc`	This doesn't truncate output (shows complete ID).
`-q, --quiet`	This shows only the image IDs.

Now let's look at a few examples of the usage of the `image` command:

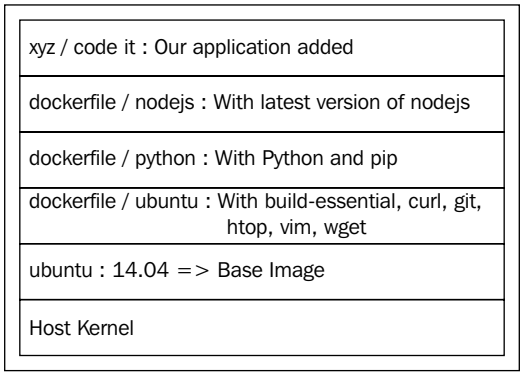
```
$ docker images
REPOSITORY            TAG    IMAGE ID        CREATED     VIRTUAL SIZE
shrikrishna/code.it   v1     a7cb6737a2f6    6m ago      704.4 MB
```

This lists all top-level images, their repository and tags, and their virtual size.

Docker images are nothing but a stack of read-only filesystem layers. A union filesystem, such as AUFS, then merges these layers and they appear to be one filesystem.

In Docker-speak, a read-only layer is an image. It never changes. When running a container, the processes think that the entire filesystem is read-write. But the changes go only at the topmost writeable layer, which is created when a container is started. The read-only layers of the image remain unchanged. When you commit a container, it freezes the top layer (the underlying layers are already frozen) and turns it into an image. Now, when a container is started this image, all the layers of the image (including the previously writeable layer) are read-only. All the changes are now made to a new writeable layer on top of all the underlying layers. However, because of how union filesystems (such as AUFS) work, the processes believe that the filesystem is read-write.

A rough schematic of the layers involved in our `code.it` example is as follows:



At this point, it might be wise to think just how much effort is to be made by the union filesystems to merge all of these layers and provide a consistent performance. After some point, things inevitably break. AUFS, for instance, has a 42-layer limit. When the number of layers goes beyond this, it just doesn't allow the creation of any more layers and the build fails. Read `https://github.com/docker/docker/issues/1171` for more information on this issue.

The following command lists the most recently created images:

```
$ docker images | head
```

The `-f` flag can be given arguments of the `key=value` type. It is frequently used to get the list of dangling images:

```
$ docker images -f "dangling=true"
```

This will display untagged images, that is, images that have been committed or built without a tag.

The rmi command

The `rmi` command removes images. Removing an image also removes all the underlying images that it depends on and were downloaded when it was pulled:

```
$ docker rmi [OPTION] {IMAGE(s)]
```

Flag	Explanation
`-f, --force`	This forcibly removes the image (or images).
`--no-prune`	This command does not delete untagged parents.

This command removes one of the images from your machine:

```
$ docker rmi test
```

The save command

The `save` command saves an image or repository in a tarball and this streams to the `stdout` file, preserving the parent layers and metadata about the image:

```
$ docker save -o codeit.tar code.it
```

The `-o` flag allows us to specify a file instead of streaming to the `stdout` file. It is used to create a backup that can then be used with the `docker load` command.

The load command

The `load` command loads an image from a tarball, restoring the filesystem layers and the metadata associated with the image:

```
$ docker load -i codeit.tar
```

The `-i` flag allows us to specify a file instead of trying to get a stream from the `stdin` file.

The export command

The `export` command saves the filesystem of a container as a tarball and streams to the `stdout` file. It flattens filesystem layers. In other words, it merges all the filesystem layers. All of the metadata of the image history is lost in this process:

```
$ sudo Docker export red_panda > latest.tar
```

Here, `red_panda` is the name of one of my containers.

The import command

The `import` command creates an empty filesystem image and imports the contents of the tarball to it. You have the option of tagging it the image:

```
$ docker import URL|- [REPOSITORY[:TAG]]
```

URLs must start with `http`.

```
$ docker import http://example.com/test.tar.gz # Sample url
```

If you would like to import from a local directory or archive, you can use the - parameter to take the data from the `stdin` file:

```
$ cat sample.tgz | docker import – testimage:imported
```

The tag command

You can add a `tag` command to an image. It helps identify a specific version of an image.

For example, the `python` image name represents `python:latest`, the latest version of Python available, which can change from time to time. But whenever it is updated, the older versions are tagged with the respective Python versions. So the `python:2.7` command will have Python 2.7 installed. Thus, the `tag` command can be used to represent versions of the images, or for any other purposes that need identification of the different versions of the image:

```
$ docker tag IMAGE [REGISTRYHOST/][USERNAME/]NAME[:TAG]
```

The `REGISTRYHOST` command is only needed if you are using a private registry of your own. The same image can have multiple tags:

```
$ docker tag shrikrishna/code.it:v1 shrikrishna/code.it:latest
```

Whenever you are tagging an image, follow the `username/repository:tag` convention.

Now, running the `docker images` command again will show that the same image has been tagged with both the `v1` and `latest` commands:

```
$ docker images
REPOSITORY            TAG     IMAGE ID      CREATED     VIRTUAL SIZE
shrikrishna/code.it   v1      a7cb6737a2f6  8 days ago  704.4 MB
shrikrishna/code.it   latest  a7cb6737a2f6  8 days ago  704.4 MB
```

The login command

The `login` command is used to register or log in to a Docker registry server. If no server is specified, `https://index.docker.io/v1/` is the default:

```
$ Docker login [OPTIONS] [SERVER]
```

Flag	Explanation
`-e, --email=""`	Email
`-p, --password=""`	Password
`-u, --username=""`	Username

If the flags haven't been provided, the server will prompt you to provide the details. After the first login, the details will be stored in the `$HOME/.dockercfg` path.

The push command

The `push` command is used to push an image to the public image registry or a private Docker registry:

```
$ docker push NAME[:TAG]
```

The history command

The `history` command shows the history of the image:

```
$ docker history shykes/nodejs
IMAGE          CREATED         CREATED BY                      SIZE
6592508b0790   15 months ago   /bin/sh -c wget http://nodejs.  15.07 MB
0a2ff988ae20   15 months ago   /bin/sh -c apt-get install ...  25.49 MB
43c5d81f45de   15 months ago   /bin/sh -c apt-get update       96.48 MB
b750fe79269d   16 months ago   /bin/bash                       77 B
27cf78414709   16 months ago                                   175.3 MB
```

The events command

Once started, the `events` command prints all the events that are handled by the `docker` daemon, in real time:

```
$ docker events [OPTIONS]
```

Flag	Explanation
`--since=""`	This shows all events created since timestamp (in Unix).
`--until=""`	This stream events until timestamp.

For example the `events` command is used as follows:

```
$ docker events
```

Now, in a different tab, run this command:

```
$ docker start code.it
```

Then run the following command:

```
$ docker stop code.it
```

Now go back to the tab running Docker events and see the output. It will be along these lines:

```
[2014-07-21 21:31:50 +0530 IST]
c7f2485863b2c7d0071477e6cb8c8301021ef9036afd4620702a0de08a4b3f7b: (from
dockerfile/nodejs:latest) start


[2014-07-21 21:31:57 +0530 IST]
c7f2485863b2c7d0071477e6cb8c8301021ef9036afd4620702a0de08a4b3f7b: (from
dockerfile/nodejs:latest) stop


[2014-07-21 21:31:57 +0530 IST]
c7f2485863b2c7d0071477e6cb8c8301021ef9036afd4620702a0de08a4b3f7b: (from
dockerfile/nodejs:latest) die
```

You can use flags such as `--since` and `--until` to get the event logs of specific timeframes.

The wait command

The `wait` command blocks until a container stops, then prints its exit code:

```
$ docker wait CONTAINER(s)
```

The build command

The build command builds an image from the source files at a specified path:

```
$ Docker build [OPTIONS] PATH | URL | -
```

Flag	Explanation
`-t, --tag=""`	This is the repository name (and an optional tag) to be applied to the resulting image in case of success.
`-q, --quiet`	This suppresses the output, which by default is verbose.
`--rm=true`	This removes intermediate containers after a successful build.
`--force-rm`	This always removes intermediate containers, even after unsuccessful builds.
`--no-cache`	This command does not use the cache while building the image.

This command uses a Dockerfile and a context to build a Docker image.

A Dockerfile is like a Makefile. It contains instructions on the various configurations and commands that need to be run in order to create an image. We will look at writing Dockerfiles in the next section.

It would be a good idea to read the section about Dockerfiles first and then come back here to get a better understanding of this command and how it works.

The files at the `PATH` or `URL` paths are called **context** of the build. The context is used to refer to the files or folders in the Dockerfile, for instance in the `ADD` instruction (and that is the reason an instruction such as `ADD ../file.txt` won't work. It's not in the context!).

When a GitHub URL or a URL with the `git://` protocol is given, the repository is used as the context. The repository and its submodules are recursively cloned in your local machine, and then uploaded to the `docker` daemon as the context. This allows you to have Dockerfiles in your private Git repositories, which you can access from your local user credentials or from the **Virtual Private Network** (**VPN**).

Uploading to Docker daemon

Remember that Docker engine has both the `docker` daemon and the Docker client. The commands that you give as a user are through the Docker client, which then talks to the `docker` daemon (either through a TCP or a Unix socket), which does the necessary work. The `docker` daemon and Docker host can be in different hosts (which is the premise with which boot2Docker works), with the `DOCKER_HOST` environment variable set to the location of the remote `docker` daemon.

When you give a context to the `docker build` command, all the files in the local directory get tared and are sent to the `docker` daemon. The `PATH` variable specifies where to find the files for the context of the build in the `docker` daemon. So when you run `docker build .`, all the files in the current folder get uploaded, not just the ones listed to be added in the Dockerfile.

Since this can be a bit of a problem (as some systems such as Git and some IDEs such as Eclipse create hidden folders to store metadata), Docker provides a mechanism to ignore certain files or folders by creating a file called `.dockerignore` in the `PATH` variable with the necessary exclusion patterns. For an example, look up `https://github.com/docker/docker/blob/master/.dockerignore`.

If a plain URL is given or if the Dockerfile is streamed through the `stdin` file, then no context is set. In these cases, the `ADD` instruction works only if it refers to a remote URL.

Now let's build the `code.it` example image through a Dockerfile. The instructions on how to create this Dockerfile are provided in the *Dockerfile* section.

At this point, you would have created a directory and placed the Dockerfile inside it. Now, on your terminal, go to that directory and execute the `docker build` command:

```
$ docker build -t shrikrishna/code.it:docker .
Sending build context to Docker daemon  2.56 kB
Sending build context to Docker daemon
Step 0 : FROM Dockerfile/nodejs
 ---> 1535da87b710
Step 1 : MAINTAINER Shrikrishna Holla <s**a@gmail.com>
 ---> Running in e4be61c08592
 ---> 4c0eabc44a95
Removing intermediate container e4be61c08592
Step 2 : WORKDIR /home
 ---> Running in 067e8951cb22
 ---> 81ead6b62246
Removing intermediate container 067e8951cb22
. . . . .
. . . . .
Step 7 : EXPOSE  8000
 ---> Running in 201e07ec35d3
 ---> 1db6830431cd
Removing intermediate container 201e07ec35d3
Step 8 : WORKDIR /home
 ---> Running in cd128a6f090c
 ---> ba05b89b9cc1
Removing intermediate container cd128a6f090c
Step 9 : CMD     ["/usr/bin/node", "/home/code.it/app.js"]
 ---> Running in 6da5d364e3e1
 ---> 031e9ed9352c
Removing intermediate container 6da5d364e3e1
Successfully built 031e9ed9352c
```

Now, you will be able to look at your newly built image in the output of Docker images

```
REPOSITORY          TAG        IMAGE ID     CREATED      VIRTUAL SIZE
shrikrishna/code.it Dockerfile 031e9ed9352c 21 hours ago 1.02 GB
```

To see the caching in action, run the same command again

```
$ docker build -t shrikrishna/code.it:dockerfile .
Sending build context to Docker daemon  2.56 kB
Sending build context to Docker daemon
Step 0 : FROM dockerfile/nodejs
 ---> 1535da87b710
Step 1 : MAINTAINER Shrikrishna Holla <s**a@gmail.com>
 ---> Using cache
 ---> 4c0eabc44a95
Step 2 : WORKDIR /home
 ---> Using cache
 ---> 81ead6b62246
Step 3 : RUN     git clone https://github.com/shrikrishnaholla/code.
it.git
 ---> Using cache
 ---> adb4843236d4
Step 4 : WORKDIR code.it
 ---> Using cache
 ---> 755d248840bb
Step 5 : RUN     git submodule update --init --recursive
 ---> Using cache
 ---> 2204a519efd3
Step 6 : RUN     npm install
 ---> Using cache
 ---> 501e028d7945
Step 7 : EXPOSE  8000
 ---> Using cache
 ---> 1db6830431cd
Step 8 : WORKDIR /home
 ---> Using cache
 ---> ba05b89b9cc1
Step 9 : CMD     ["/usr/bin/node", "/home/code.it/app.js"]
 ---> Using cache
 ---> 031e9ed9352c
Successfully built 031e9ed9352c
```

Now experiment with this caching. Change one of the lines in the middle (the port number for example), or add a `RUN echo "testing cache"` line somewhere in the middle and see what happens.

An example of building an image using a repository URL is as follows:

```
$ docker build -t shrikrishna/optimus:git_url \ git://github.com/
shrikrishnaholla/optimus
Sending build context to Docker daemon 1.305 MB
Sending build context to Docker daemon
Step 0 : FROM        dockerfile/nodejs
 ---> 1535da87b710
Step 1 : MAINTAINER  Shrikrishna Holla
 ---> Running in d2aae3dba68c
 ---> 0e8636eac25b
Removing intermediate container d2aae3dba68c
Step 2 : RUN         git clone https://github.com/pesos/optimus.git
/home/optimus
 ---> Running in 0b46e254e90a
. . . . .
. . . . .
. . . . .
Step 5 : CMD         ["/usr/local/bin/npm", "start"]
 ---> Running in 0e01c71faa0b
 ---> 0f0dd3deae65
Removing intermediate container 0e01c71faa0b
Successfully built 0f0dd3deae65
```

Dockerfile

We have seen how to create images by committing containers. What if you want to update the image with new versions of dependencies or new versions of your own application? It soon becomes impractical to do the steps of starting, setting up, and committing over and over again. We need a repeatable method to build images. In comes Dockerfile, which is nothing more than a text file that contains instructions to automate the steps you would otherwise have taken to build an image. `docker build` will read these instructions sequentially, committing them along the way, and build an image.

The `docker build` command takes this Dockerfile and a context to execute the instructions, and builds a Docker image. Context refers to the path or source code repository URL given to the `docker build` command.

A Dockerfile contains instructions in this format:

```
# Comment
INSTRUCTION arguments
```

Any line beginning with `#` will be considered as a comment. If a `#` sign is present anywhere else, it will be considered a part of arguments. The instruction is not case-sensitive, although it is an accepted convention for instructions to be uppercase so as to distinguish them from the arguments.

Let's look at the instructions that we can use in a Dockerfile.

The FROM instruction

The `FROM` instruction sets the base image for the subsequent instructions. A valid Dockerfile's first non-comment line will be a `FROM` instruction:

```
FROM <image>:<tag>
```

The image can be any valid local or public image. If it is not found locally,the `Docker build` command will try to pull it from the public registry. The `tag` command is optional here. If it is not given, the `latest` command is assumed. If the incorrect `tag` command is given, it returns an error.

The MAINTAINER instruction

The `MAINTAINER` instruction allows you to set the author for the generated images:

```
MAINTAINER <name>
```

The RUN instruction

The `RUN` instruction will execute any command in a new layer on top of the current image, and commit this image. The image thus committed will be used for the next instruction in the Dockerfile.

The `RUN` instruction has two forms:

- The `RUN <command>` form
- The `RUN ["executable", "arg1", "arg2"...]` form

In the first form, the command is run in a shell, specifically the `/bin/sh -c <command>` shell. The second form is useful in instances where the base image doesn't have a `/bin/sh` shell. Docker uses a cache for these image builds. So in case your image build fails somewhere in the middle, the next run will reuse the previously successful partial builds and continue from the point where it failed.

The cache will be invalidated in these situations:

- When the `docker build` command is run with the `--no-cache` flag.
- When a non-cacheable command such as `apt-get update` is given. All the following `RUN` instructions will be run again.
- When the first encountered `ADD` instruction will invalidate the cache for all the following instructions from the Dockerfile if the contents of the context have changed. This will also invalidate the cache for the `RUN` instructions.

The CMD instruction

The `CMD` instruction provides the default command for a container to execute. It has the following forms:

- The `CMD ["executable", "arg1", "arg2"...]` form
- The `CMD ["arg1", "arg2"...]` form
- The `CMD command arg1 arg2` ... form

The first form is like an exec and it is the preferred form, where the first value is the path to the executable and is followed by the arguments to it.

The second form omits the executable but requires the `ENTRYPOINT` instruction to specify the executable.

If you use the shell form of the `CMD` instruction, then the `<command>` command will execute in the `/bin/sh -c` shell.

If the user provides a command in `docker run`, it overrides the `CMD` command.

The difference between the `RUN` and `CMD` instructions is that a `RUN` instruction actually runs the command and commits it, whereas the `CMD` instruction is not executed during build time. It is a default command to be run when the user starts a container, unless the user provides a command to start it with.

For example, let's write a `Dockerfile` that brings a `Star Wars` output to your terminal:

```
FROM ubuntu:14.04
MAINTAINER shrikrishna
RUN apt-get -y install telnet
CMD ["/usr/bin/telnet", "towel.blinkenlights.nl"]
```

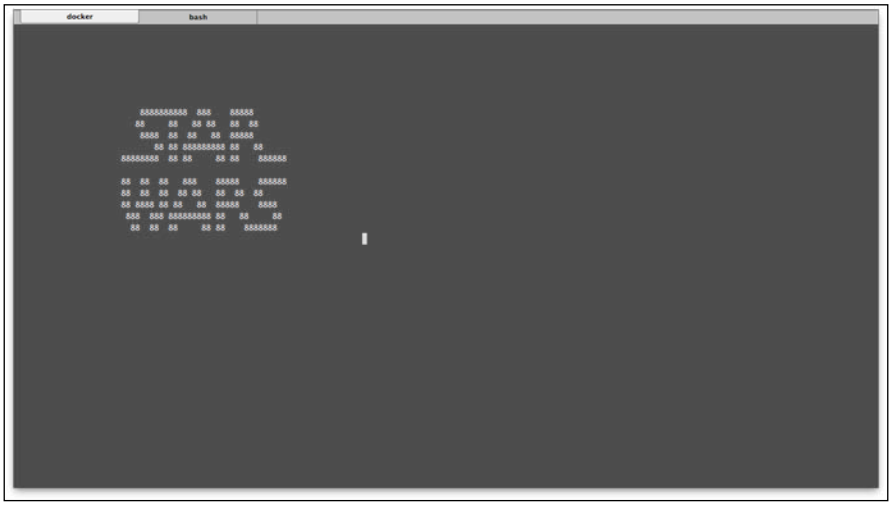
Save this in a folder named `star_wars` and open your terminal at this location. Then run this command:

```
$ docker build -t starwars .
```

Now you can run it using the following command:

```
$ docker run -it starwars
```

The following screenshot shows the `starwars` output:



Thus, you can watch **Star Wars** in your terminal!

This *Star Wars* tribute was created by Simon Jansen, Sten Spans, and Mike Edwards. When you've had enough, hold *Ctrl* + *]*. You will be given a prompt where you can type `close` to exit.

The ENTRYPOINT instruction

The `ENTRYPOINT` instruction allows you to turn your Docker image into an executable. In other words, when you specify an executable in an `ENTRYPOINT`, containers will run as if it was just that executable.

The `ENTRYPOINT` instruction has two forms:

1. The `ENTRYPOINT ["executable", "arg1", "arg2"...]` form.
2. The `ENTRYPOINT command arg1 arg2 ...` form.

This instruction adds an entry command that will not be overridden when arguments are passed to the `docker run` command, unlike the behavior of the `CMD` instruction. This allows arguments to be passed to the `ENTRYPOINT` instruction. The `docker run <image> -arg` command will pass the `-arg` argument to the command specified in the `ENTRYPOINT` instruction.

Parameters, if specified in the `ENTRYPOINT` instruction, will not be overridden by the `docker run` arguments, but parameters specified via the `CMD` instruction will be overridden.

As an example, let's write a Dockerfile with `cowsay` as the `ENTRYPOINT` instruction:

The `cowsay` is a program that generates ASCII pictures of a cow with a message. It can also generate pictures using premade images of other animals, such as Tux the Penguin, the Linux mascot.

```
FROM ubuntu:14.04
RUN apt-get -y install cowsay
ENTRYPOINT ["/usr/games/cowsay"]
CMD ["Docker is so awesomooooooooo!"]
```

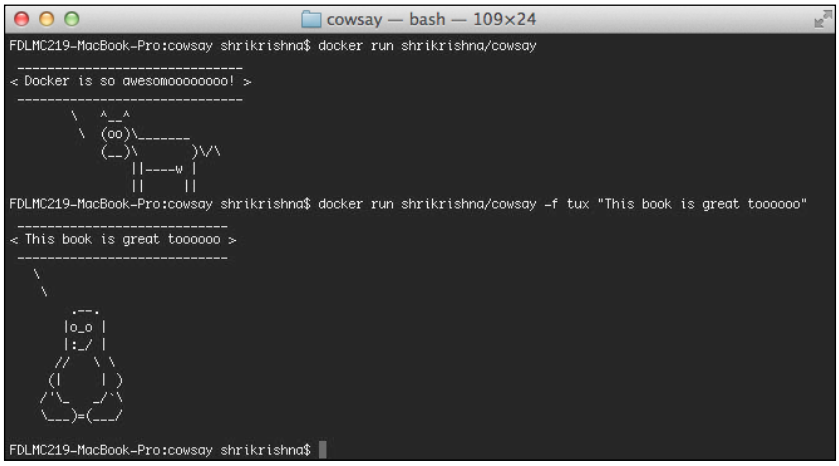
Save this with the name `Dockerfile` in a folder named `cowsay`. Then through terminal, go to that directory, and run this command:

```
$ docker build -t cowsay .
```

Once the image is built, run the following command:

```
$ docker run cowsay
```

The following screenshot shows the output of the preceding command:



If you look at the screenshot closely, the first run has no arguments and it used the argument we configured in the Dockerfile. However, when we gave our own arguments in the second run, it overrode the default and passed all the arguments (The `-f` flag and the sentence) to the `cowsay` folder.

If you are the kind who likes to troll others, here's a tip: apply the instructions given at `http://superuser.com/a/175802` to set up a pre-exec script (a function that is called whenever a command is executed) that passes every command to this Docker container, and place it in the `.bashrc` file. Now cowsay will print every command that it execute in a text balloon, being said by an ASCII cow!

The WORKDIR instruction

The `WORKDIR` instruction sets the working directory for the `RUN`, `CMD`, and `ENTRYPOINT` Dockerfile commands that follow it:

```
WORKDIR /path/to/working/directory
```

This instruction can be used multiple times in the same Dockerfile. If a relative path is provided, the `WORKDIR` instruction will be relative to the path of the previous `WORKDIR` instruction.

The EXPOSE instruction

The `EXPOSE` instruction informs Docker that a certain port is to be exposed when a container is started:

```
EXPOSE port1 port2 …
```

Even after exposing ports, while starting a container, you still need to provide port mapping using the `-p` flag to `Docker run`. This instruction is useful when linking containers, which we will see in *Chapter 3*, *Linking Containers*.

The ENV instruction

The ENV command is used to set environment variables:

```
ENV <key> <value>
```

This sets the `<key>` environment variable to `<value>`. This value will be passed to all future `RUN` instructions. This is equivalent to prefixing the command with `<key>=<value>`.

The environment variables set using the `ENV` command will persist. This means that when a container is run from the resulting image, the environment variable will be available to the running process as well. The `docker inspect` command shows the values that have been assigned during the creation of the image. However, these can be overridden using the `$ docker run –env <key>=<value>` command.

The USER instruction

The USER instruction sets the username or UID to use when running the image and any following the `RUN` directives:

```
USER xyz
```

The VOLUME instruction

The `VOLUME` instruction will create a mount point with the given name and mark it as holding externally mounted volumes from the host or from other containers:

```
VOLUME [path]
```

Here is an example of the `VOLUME` instruction:

```
VOLUME ["/data"]
```

Here is another example of this instruction:

```
VOLUME /var/log
```

Both formats are acceptable.

The ADD instruction

The `ADD` instruction is used to copy files into the image:

```
ADD <src> <dest>
```

The `ADD` instruction will copy files from `<src>` into the path at `<dest>`.

The `<src>` path must be the path to a file or directory relative to the source directory being built (also called the context of the build) or a remote file URL.

The `<dest>` path is the absolute path to which the source will be copied inside the destination container.

If you build by passing a Dockerfile through the `stdin` file (`docker build - < somefile`), there is no build context, so the Dockerfile can only contain a URL-based `ADD` statement. You can also pass a compressed archive through the `stdin` file (`docker build - < archive.tar.gz`). Docker will look for a Dockerfile at the root of the archive and the rest of the archive will get used as the context of the build.

The `ADD` instruction obeys the following rules:

- The `<src>` path must be inside the context of the build. You cannot use `ADD ../file as ..` syntax, as it is beyond the context.
- If `<src>` is a URL and the `<dest>` path doesn't end with a trailing slash (it's a file), then the file at the URL is copied to the `<dest>` path.
- If `<src>` is a URL and the `<dest>` path ends with a trailing slash (it's a directory), then the content at the URL is fetched and a filename is inferred from the URL and saved into the `<dest>/filename` path. So, the URL cannot have a simple path such as `example.com` in this case.
- If `<src>` is a directory, the entire directory is copied, along with the filesystem metadata.
- If `<src>` is a local tar archive, then it is extracted into the `<dest>` path. The result at `<dest>` is union of:
    - Whatever existed at the path `<dest>`.

    - Contents of the extracted tar archive, with conflicts in favor of the path `<src>`, on a file-by-file basis.

- If `<dest>` path doesn't exist, it is created along with all the missing directories along its path.

The COPY instruction

The COPY instruction copies a file into the image:

```
COPY <src> <dest>
```

The `Copy` instruction is similar to the `ADD` instruction. The difference is that the `COPY` instruction does not allow any file out of the context. So, if you are streaming Dockerfile via the `stdin` file or a URL (which doesn't point to a source code repository), the `COPY` instruction cannot be used.

The ONBUILD instruction

The `ONBUILD` instruction adds to the image a trigger that will be executed when the image is used as a base image for another build:

```
ONBUILD [INSTRUCTION]
```

This is useful when the source application involves generators that need to compile before they can be used. Any build instruction apart from the `FROM`, `MAINTAINER`, and `ONBUILD` instructions can be registered.

Here's how this instruction works:

1. During a build, if the `ONBUILD` instruction is encountered, it registers a trigger and adds it to the metadata of the image. The current build is not otherwise affected in any way.
2. A list of all such triggers is added to the image manifest as a key named `OnBuild` at the end of the build (which can be seen through the `Docker inspect` command).
3. When this image is later used as a base image for a new build, as part of processing the `FROM` instruction, the `OnBuild key` triggers are read and executed in the order they were registered. If any of them fails, the `FROM` instruction aborts, causing the build to fail. Otherwise, the `FROM` instruction completes and the build continues as usual.
4. Triggers are cleared from the final image after being executed. In other words they are not inherited by *grand-child builds*.

Let's bring `cowsay` back! Here's a Dockerfile with the `ONBUILD` instruction:

```
FROM ubuntu:14.04
RUN apt-get -y install cowsay
RUN apt-get -y install fortune
ENTRYPOINT ["/usr/games/cowsay"]
CMD ["Docker is so awesomooooooooo!"]
ONBUILD RUN /usr/games/fortune | /usr/games/cowsay
```

Now save this file in a folder named `OnBuild`, open a terminal in that folder, and run this command:

```
$ Docker build -t shrikrishna/onbuild .
```

We need to write another Dockerfile that builds on this image. Let's write one:

```
FROM shrikrishna/onbuild
RUN  apt-get moo
CMD ['/usr/bin/apt-get', 'moo']
```

The `apt-get moo` command is an example of Easter eggs typically found in many open source tools, added just for the sake of fun!

Building this image will now execute the `ONBUILD` instruction we gave earlier:

```
$ docker build -t shrikrishna/apt-moo apt-moo/
Sending build context to Docker daemon  2.56 kB
Sending build context to Docker daemon
Step 0 : FROM shrikrishna/onbuild
# Executing 1 build triggers
Step onbuild-0 : RUN /usr/games/fortune | /usr/games/cowsay
 ---> Running in 887592730f3d
 _____________________________________
/ It was all so different before \
\ everything changed.               /
 -------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
```

```
                ||     ||
 ---> df01e4ca1dc7
 ---> df01e4ca1dc7
Removing intermediate container 887592730f3d
Step 1 : RUN  apt-get moo
 ---> Running in fc596cb91c2a
                 (__)
                 (oo)
           /------\/
          / |    ||
         *  /\---/\
            ~~   ~~
..."Have you mooed today?"...
 ---> 623cd16a51a7
Removing intermediate container fc596cb91c2a
Step 2 : CMD ['/usr/bin/apt-get', 'moo']
 ---> Running in 22aa0b415af4
 ---> 7e03264fbb76
Removing intermediate container 22aa0b415af4
Successfully built 7e03264fbb76
```

Now let's use our newly gained knowledge to write a Dockerfile for the `code.it` application that we previously built by manually satisfying dependencies in a container and committing. The Dockerfile would look something like this:

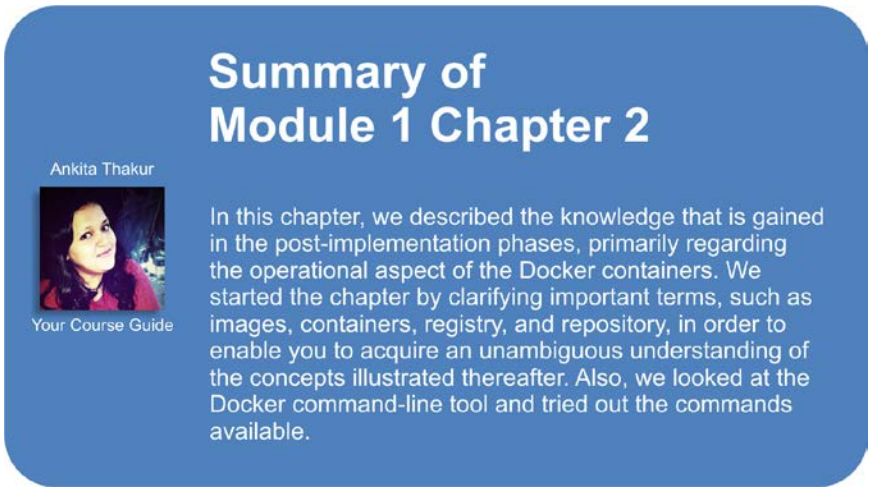
```
# Version 1.0
FROM dockerfile/nodejs
MAINTAINER Shrikrishna Holla <s**a@gmail.com>

WORKDIR /home
RUN     git clone \ https://github.com/shrikrishnaholla/code.it.git

WORKDIR code.it
RUN     git submodule update --init --recursive
RUN     npm install

EXPOSE  8000
```

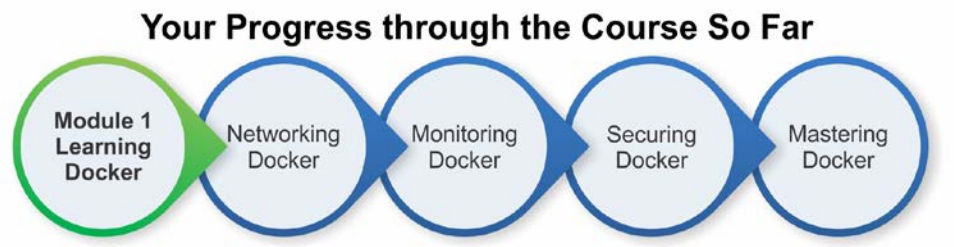
```
WORKDIR /home
CMD     ["/usr/bin/node", "/home/code.it/app.js"]
```

Create a folder named `code.it` and save this content as a file named `Dockerfile`.

It is good practice to create a separate folder for every Dockerfile even if there is no context needed. This allows you to separate concerns between different projects. You might notice as you go that many Dockerfile authors club `RUN` instructions (for example, check out the Dockerfiles in `dockerfile.github.io`). The reason is that AUFS limits the number of possible layers to 42. For more information, check out this issue at `https://github.com/docker/docker/issues/1171`.

You can go back to the section on *Docker build* to see how to build an image out of this Dockerfile.





3
Container Image Storage

In this chapter, 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.

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
```

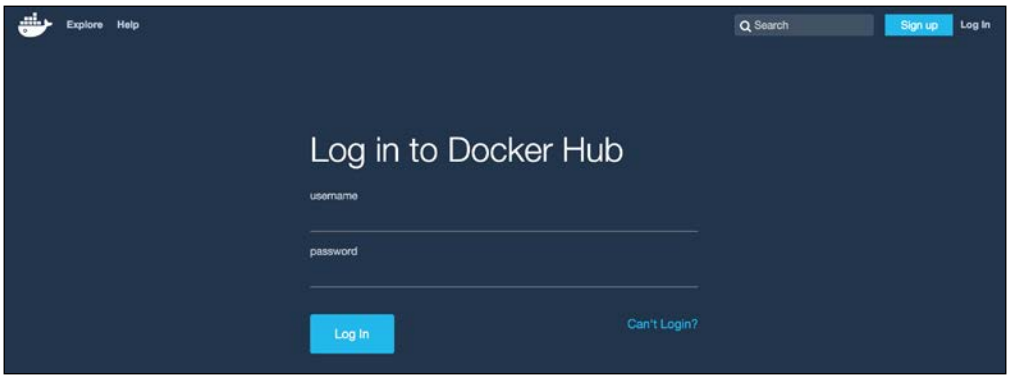
This will prompt you for your Docker Hub credentials and the e-mail address you are using on Docker Hub:

```
$ 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.

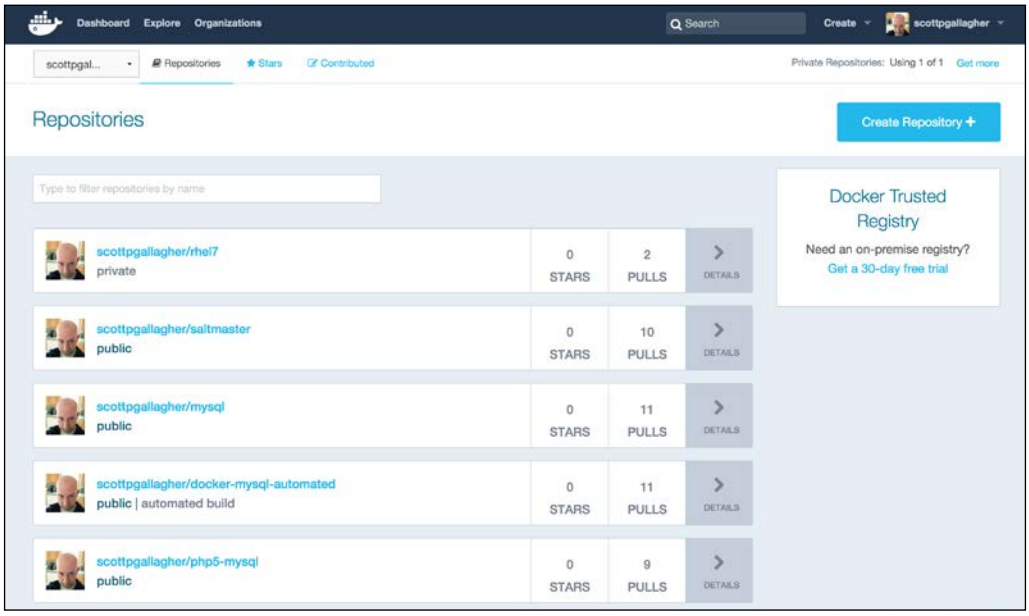
We just glazed over Docker Hub as a storage location to push our images to. In the next section, we will focus on that Docker Hub, which is a free public option, but also has a private option that you can use to secure your images. We will focus on the web aspect of Docker Hub and the management you can do there.

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



Dashboard

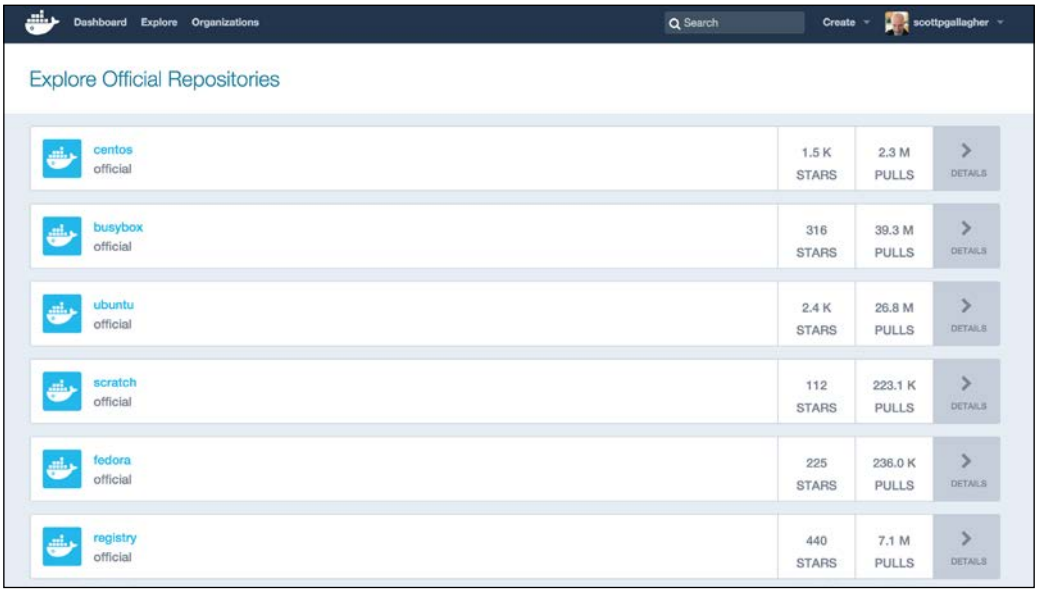
After logging into the Docker Hub, you will be taken to the following landing page. This page is known as the **Dashboard** of Docker Hub.



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:

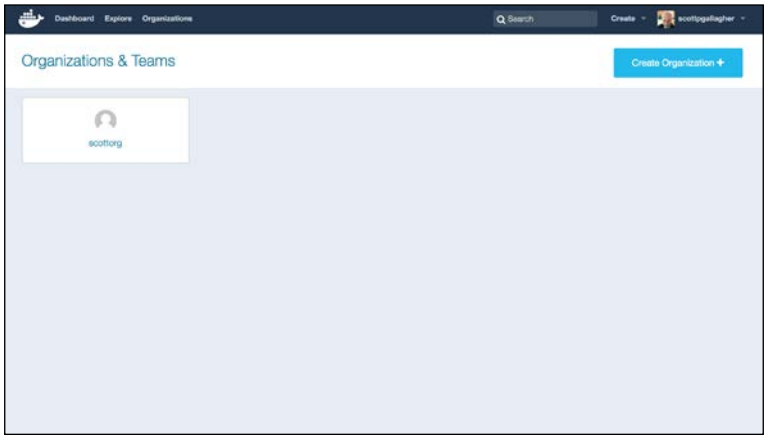


As you can see in the screenshot, this is a link to show you all the official repositories that Docker has to offer. Official repositories are those that come directly from Docker or from the company responsible for the product. They are regularly updated and patched as needed.

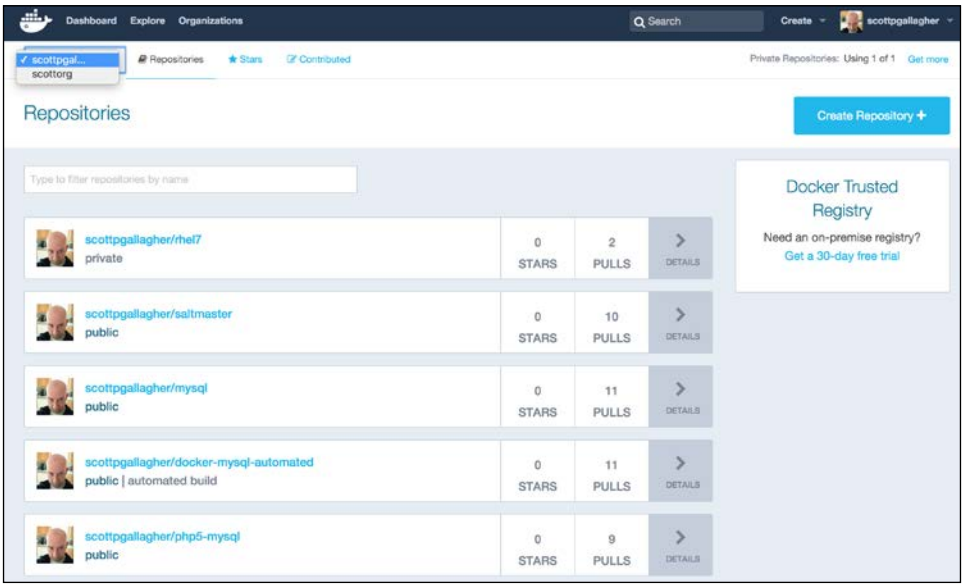
Organizations

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.



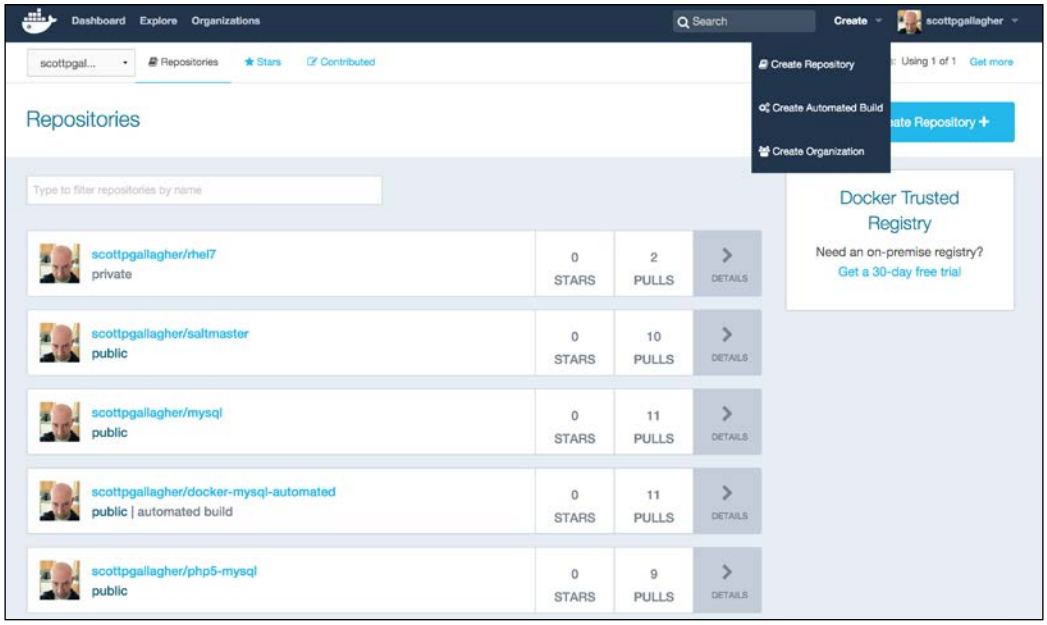
This is a drop-down list, where you can switch between all the organizations you belong to.

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:

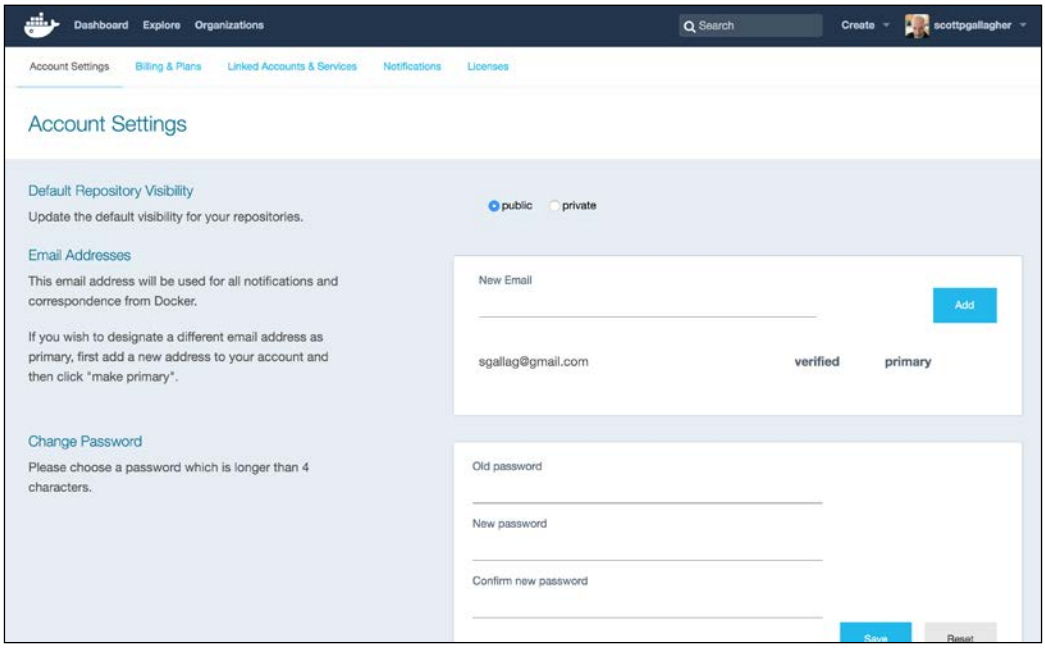
- Create repository
- Create automated build
- Create organization

A pictorial representation is shown in the following screenshot:

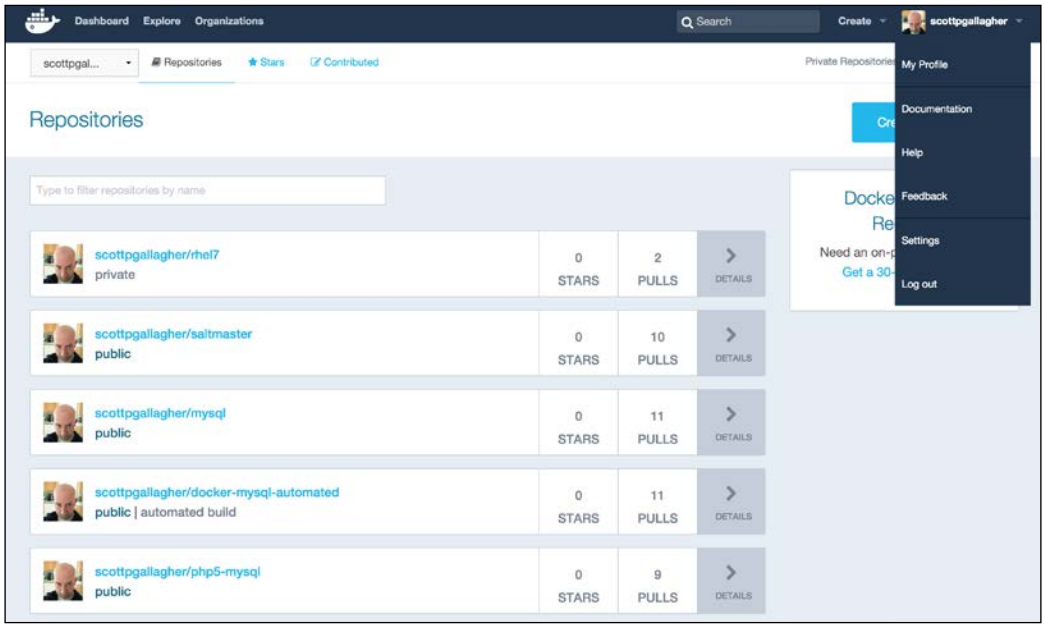


Settings

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.



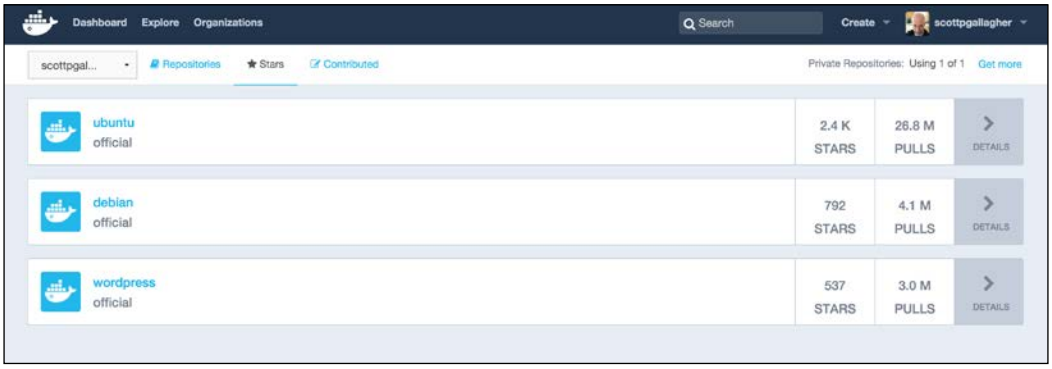
The **Account Settings** page can be found under the drop-down menu that is accessed in the upper-right corner of the dashboard on selecting **Settings**.



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

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.

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.

Your Coding Challenge

Ankita Thakur

Your Course Guide

So far you have seen the example of running just one service inside a container. If you want to run an application, which requires us to run one or more services simultaneously, then, either you will need to run them on the same container or run them on different containers and link them together. WordPress is one such example that requires a database and web service.

Docker only likes one process per container running in the foreground. Thus, in order to make Docker happy, we have a controlling process that manages the database and web services. The controlling process, in this case, is supervisord (`http://supervisord.org/`). This is a trick that you use to make Docker happy. Try building a WordPress image.

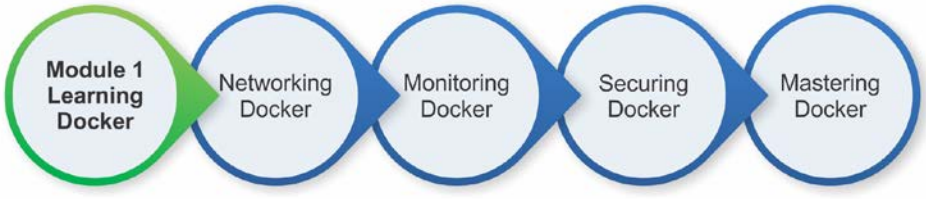
Summary of Module 1 Chapter 3

Ankita Thakur

Your Course Guide

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 showed 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.

Your Progress through the Course So Far



4
Working with Docker containers and images

In the previous chapter, we explained stimulating and sustainable concepts, which showed the Docker's way of crafting futuristic and flexible application-aware containers. We discussed all the relevant details of producing the Docker containers in multiple environments (on-premise as well as off-premise). Using these techniques, you can easily replicate these features in your own environments to get a rewarding experience. Therefore, the next step for us is to understand the container's life cycle aspects in a decisive manner. You will learn the optimal utilization of containers of our own as well as those of other third-party containers in an effective and risk-free way. Containers are to be found, assessed, accessed, and leveraged toward bigger and better applications. There are several tools that have emerged to streamline the handling of containers.

In this chapter, we will dig deeper and describe the critical aspects of container handling at length. A number of practical tips and execution commands for the leveraging of containers will also be discussed in this chapter.

In this chapter, we will cover the following topics:

- Working with the Docker images and containers
- The meaning of the Docker registry and its repository
- The Docker Hub Registry
- Searching the Docker images
- Working with an interactive container
- Tracking the changes inside the containers

- Controlling and housekeeping the Docker containers
- Building images from containers
- Launching a container as a daemon

Docker Hub Registry

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

The good folks in the Docker community have built a repository of images and they have made it publicly available at a default location, `index.docker.io`. This default location is called the Docker index. The `docker pull` subcommand is programmed to look for the images at this location. Therefore, when you `pull` a `busybox` image, it is effortlessly downloaded from the default registry. This mechanism helps in speeding up the spinning of the Docker containers. The Docker Index is the official repository that contains all the painstakingly curated images that are created and deposited by the worldwide Docker development community.

This so-called cure is enacted to ensure that all the images stored in the Docker index are secure and safe through a host of quarantine tasks. There are proven verification and validation methods for cleaning up any knowingly or unknowingly introduced malware, adware, viruses, and so on, from these Docker images. The digital signature is a prominent mechanism of the utmost integrity of the Docker images. Nonetheless, if the official image has been either corrupted, or tampered with, then the Docker engine will issue a warning and then continue to run the image.

In addition to the official repository, the Docker Hub Registry also provides a platform for the third-party developers and providers for sharing their images for general consumption. The third-party images are prefixed by the user ID of their developers or depositors. For example, `thedockerbook/helloworld` is a third-party image, wherein `thedockerbook` is the user ID and `helloworld` is the image repository name. You can download any third-party image by using the `docker pull` subcommand, as shown here:

```
$ sudo docker pull thedockerbook/helloworld
```

Apart from the preceding repository, the Docker ecosystem also provides a mechanism for leveraging the images from any third-party repository hub other than the Docker Hub Registry, and it provides the images hosted by the local repository hubs. As mentioned earlier, the Docker engine has been programmed to look for images in `index.docker.io` by default, whereas in the case of the third-party or the local repository hub, we must manually specify the path from where the image should be pulled. A manual repository path is similar to a URL without a protocol specifier, such as `https://`, `http://` and `ftp://`. Following is an example of pulling an image from a third party repository hub:

```
$ sudo docker pull registry.example.com/myapp
```

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

Searching Docker images

As we discussed in the previous section, the Docker Hub repository typically hosts both the official images as well as the images that have been contributed by the third-party Docker enthusiasts. At the time of writing this book, more than 14,000 images (also called the Dockerized application) were available for the users. These images can be used either as is, or as a building block for the user-specific applications.

You can search for the Docker images in the Docker Hub Registry by using the `docker search` subcommand, as shown in this example:

```
$ sudo docker search mysql
```

The search on `mysql` will list 400 odd images, as follows:

```
NAME              DESCRIPTION           STARS  OFFICIAL   AUTOMATED
mysql             MySQL is the...       147    [OK]
tutum/mysql       MySQL Server..        60                [OK]
orchardup/mysql                         34                [OK]
. . . OUTPUT TRUNCATED . . .
```

As you can see in the preceding search output excerpts, the images are ordered based on their star rating. The search result also indicates whether or not the image is official. In order to stay in focus, in this example, we will show only two images. Here, you can see the official version of `mysql`, which pulls a `147` star rating image as its first result. The second result shows that this version of the `mysql` image was published by the user `tutum`. The Docker containers are fast becoming the standard for the building blocks of the distributed applications. A dynamic repository of the Docker images will be realized with the help of the enthusiastic contribution of several community members across the globe. The Repository-based software engineering will make it easier for users and programmers to quickly code and assemble their projects. The official repositories can be freely downloaded from the Docker Hub Registry, and these are curated images. They represent a community effort that is focused on providing a great base of images for applications, so that the developers and the system administrators can focus on building new features and functionalities, while minimizing their repetitive work on commodity scaffolding and plumbing.

Based on the search queries in the Docker Hub Registry and the discussions with many of the developer community members, the Docker company, which spearheaded the Docker movement so powerfully and passionately, came to the conclusion that the developer community wanted pre-built stacks of their favorite programming languages. Specifically, the developers wanted to get to work as quickly as possible writing code without wasting time wrestling with environments, scaffolding, and dependencies.

Working with an interactive container

In the first chapter, we ran our first `Hello World!` container to get a feel of how the containerization technology works. In this section, we are going to run a container in an interactive mode. The `docker run` subcommand takes an image as an input and launches it as a container. You have to pass the `-t` and `-i` flags to the docker run subcommand in order to make the container interactive. The `-i` flag is the key driver, which makes the container interactive by grabbing the standard input (`STDIN`) of the container. The `-t` flag allocates a pseudo-TTY or a pseudo terminal (terminal emulator) and then assigns that to the container.

In the following example, we are going to launch an interactive container by using the `ubuntu:14.04` image and `/bin/bash` as the command:

```
$ sudo docker run -i -t ubuntu:14.04 /bin/bash
```

Since the `ubuntu` image has not been downloaded yet, if we use the `docker pull` subcommand, then we will get the following message and the `run` command will start pulling the `ubuntu` image automatically with the following message:

```
Unable to find image 'ubuntu:14.04' locally
Pulling repository ubuntu
```

As soon as the download is completed, the container will be launched along with the `ubuntu:14.04` image. It will also launch a bash shell within the container, because we have specified `/bin/bash` as the command to be executed. This will land us in a bash prompt, as shown here:

```
root@742718c21816:/#
```

The preceding bash prompt will confirm that our container has been launched successfully, and it is ready to take our input. If you are wondering about the Hex number `742718c21816` in the prompt, then it is nothing but the hostname of the container. In the Docker parlance, the hostname is the same as the container `ID`.

Let's quickly run a few commands interactively, and then confirm that what we mentioned about the prompt is correct, as shown here:

```
root@742718c21816:/# hostname
742718c21816
root@742718c21816:/# id
uid=0(root) gid=0(root) groups=0(root)
root@742718c21816:/# echo $PS1
```

```
${debian_chroot:+($debian_chroot)}\u@\h:\w\$
root@742718c21816:/#
```

From the preceding three commands, it is quite evident that the prompt was composed by using the user ID, the hostname, and the current working directory.

Now, let's use one of the niche features of Docker for detaching it from the interactive container and then look at the details that Docker manages for this container. Yes, we can detach it from our container by using the *Ctrl* + *P* and *Ctrl* + *Q* escape sequence. This escape sequence will detach the TTY from the container and land us in the Docker host prompt `$`, however the container will continue to run. The `docker ps` subcommand will list all the running containers and their important properties, as shown here:

```
$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
742718c21816        ubuntu:14.04        "/bin/bash"         About a
minute ago   Up About a minute                          jolly_lovelace
```

The `docker ps` subcommand will list out the following details:

- `CONTAINER ID`: This shows the container `ID` associated with the container. The container `ID` is a 64 Hex digit long random number. By default, the `docker ps` subcommand will show only 12 Hex digits. You can display all the 64 digits by using the `--no-trunc` flag (for example: `sudo docker ps --no-trunc`).
- `IMAGE`: This shows the image from which the Docker container has been crafted.
- `COMMAND`: This shows you the command executed during the container launch.
- `CREATED`: This tells you when the container was created.
- `STATUS`: This tells you the current status of the container.
- `PORTS`: This tells you if any port has been assigned to the container.
- `NAMES`: The Docker engine auto-generates a random container name by concatenating an adjective and a noun. Either the container `ID` or its name can be used to take further action on the container. The container name can be manually configured by using the `--name` option in the `docker run` subcommand.

Having looked at the container status, let's attach it back to our container by using the `docker attach` subcommand as shown in the following example. We can either use the container `ID` or use its name. In this example, we have used the container name. If you don't see the prompt, then press the *Enter* key again:

```
$ sudo docker attach jolly_lovelace
root@742718c21816:/#
```

The Docker allows attaching with a container any number of times, which proves to be very handy for screen sharing.

The `docker attach` subcommand takes us back to the container prompt. Let's experiment a little more with the interactive container that is up and running by using these commands:

```
root@742718c21816:/# pwd
/
root@742718c21816:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
root@742718c21816:/# cd usr
root@742718c21816:/usr# ls
bin  games  include  lib  local  sbin  share  src
root@742718c21816:/usr# exit
exit
$
```

As soon as the bash exit command is issued to the interactive container, it will terminate the bash shell process, which in turn will stop the container. As a result, we will land on the Docker Host's prompt `$`.

Tracking changes inside containers

In the previous section, we demonstrated how to craft a container taking `ubuntu` as a base image, and then running some basic commands, such as detaching and attaching the containers. In that process, we also exposed you to the `docker ps` subcommand, which provides the basic container management functionality. In this section, we will demonstrate how we can effectively track the changes that we introduced in our container and compare it with the image from which we launched the container.

Let's launch a container in the interactive mode, as we had done in the previous section:

```
$ sudo docker run -i -t ubuntu:14.04 /bin/bash
```

Let's change the directory to `/home`, as shown here:

```
root@d5ad60f174d3:/# cd /home
```

Now we can create three empty files by using the `touch` command as shown in the following code snippet. The first `ls -l` command will show that there are no files in the directory and the second `ls -l` command will show that there are three empty files:

```
root@d5ad60f174d3:/home# ls -l
total 0
root@d5ad60f174d3:/home# touch {abc,cde,fgh}
root@d5ad60f174d3:/home# ls -l
total 0
-rw-r--r-- 1 root root 0 Sep 29 10:54 abc
-rw-r--r-- 1 root root 0 Sep 29 10:54 cde
-rw-r--r-- 1 root root 0 Sep 29 10:54 fgh
root@d5ad60f174d3:/home#
```

The Docker engine elegantly manages its filesystem and it allows us to inspect a container filesystem by using the `docker diff` subcommand. In order to inspect the container filesystem, we can either detach it from the container or use another terminal of our Docker host and then issue the `docker diff` subcommand. Since we know that any `ubuntu` container has its hostname, which is a part of its prompt, and it is also the container's `ID`, we can directly run the `docker diff` subcommand by using the container `ID` that is taken from the prompt, as shown here:

```
$ sudo docker diff d5ad60f174d3
```

In the given example, the `docker diff` subcommand will generate four lines, shown here:

```
C /home
A /home/abc
A /home/cde
A /home/fgh
```

The preceding output indicates that the `/home` directory has been modified, which has been denoted by `C`, and the `/home/abc`, `/home/cde` and the `/home/fgh` files have been added, and these are denoted by `A`. In addition, `D` denotes deletion. Since we have not deleted any files, it is not in our sample output.

Controlling Docker containers

So far, we have discussed a few practical examples for clearly articulating the nitty-gritty of the Docker containers. In this section, let us introduce a few basic as well as a few advanced command structures for meticulously illustrating how the Docker containers can be managed.

The Docker engine enables you to `start`, `stop`, and `restart` a container with a set of `docker` subcommands. Let's begin with the `docker stop` subcommand, which stops a running container. When a user issues this command, the Docker engine sends SIGTERM (-15) to the main process, which is running inside the container. The **SIGTERM** signal requests the process to terminate itself gracefully. Most of the processes would handle this signal and facilitate a graceful exit. However, if this process fails to do so, then the Docker engine will wait for a grace period. Even after the grace period, if the process has not been terminated, then the Docker engine will forcefully terminate the process. The forceful termination is achieved by sending SIGKILL (-9). The **SIGKILL** signal cannot be caught or ignored, and so it will result in an abrupt termination of the process without a proper clean-up.

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

```
$ sudo docker run -i -t ubuntu:14.04 /bin/bash
root@da1c0f7daa2a:/#
```

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

```
$ sudo docker stop da1c0f7daa2a
da1c0f7daa2a
```

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

```
root@da1c0f7daa2a:/# exit
$
```

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

```
$ sudo docker ps -a
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS                      PORTS
NAMES
da1c0f7daa2a        ubuntu:14.04        "/bin/bash"           20
minutes ago        Exited (0) 10 minutes ago
desperate_engelbart
$
```

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

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

```
$ sudo docker start da1c0f7daa2a
da1c0f7daa2a
$
```

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

```
$ sudo docker attach da1c0f7daa2a
root@da1c0f7daa2a:/#
```

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

```
$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS                      PORTS
NAMES
da1c0f7daa2a        ubuntu:14.04        "/bin/bash"            25
minutes ago        Up 3 minutes
desperate_engelbart
$
```

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

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

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

```
$ sudo docker run -i -t ubuntu:14.04 /bin/bash
root@c439077aa80a:/# while true; do date; sleep 5; done
Thu Oct  2 03:11:19 UTC 2014
Thu Oct  2 03:11:24 UTC 2014
```

```
Thu Oct  2 03:11:29 UTC 2014
Thu Oct  2 03:11:34 UTC 2014
Thu Oct  2 03:11:59 UTC 2014
Thu Oct  2 03:12:04 UTC 2014
Thu Oct  2 03:12:09 UTC 2014
Thu Oct  2 03:12:14 UTC 2014
Thu Oct  2 03:12:19 UTC 2014
Thu Oct  2 03:12:24 UTC 2014
Thu Oct  2 03:12:29 UTC 2014
Thu Oct  2 03:12:34 UTC 2014
$
```

Our little script has very faithfully printed the date and time every 5 seconds with an exception at the following position:

```
Thu Oct  2 03:11:34 UTC 2014
Thu Oct  2 03:11:59 UTC 2014
```

Here, we encountered a delay of 25 seconds, because this is when we initiated the `docker pause` subcommand on our container on the second terminal screen, as shown here:

```
$ sudo docker pause c439077aa80a
c439077aa80a
```

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

```
$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS                  PORTS               NAMES
c439077aa80a        ubuntu:14.04        "/bin/bash"         47
seconds ago      Up 46 seconds (Paused)
ecstatic_torvalds
```

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

```
$ sudo docker unpause c439077aa80a
c439077aa80a
```

We explained the `pause` and the `unpause` commands at the beginning of this section. Lastly, the container and the script running within it had been stopped by using the `docker stop` subcommand, as shown here:

```
$ sudo docker stop c439077aa80a
c439077aa80a
```

Housekeeping containers

In many of the previous examples, when we issued `docker ps -a` we saw the many stopped containers. These containers could continue to stay in the stopped status for ages, if we chose not to intervene. At the outset, it may look like a glitch, but in reality, we can perform operations, such as committing an image from a container, restarting the stopped container, and so on. However, not all the stopped containers will be reused, and each of these unused containers will take up the disk space in the filesystem of the Docker host. The Docker engine provides a couple of ways to alleviate this issue. Let's start exploring them.

During a container startup, we can instruct the Docker engine to clean up the container as soon as it reaches the stopped state. For this purpose, the `docker run` subcommand supports an `--rm` option (for example: `sudo docker run -i -t --rm ubuntu:14.04 /bin/bash`).

The other alternative is to list all the containers by using the `-a` option of the `docker ps` subcommand and then manually remove them by using the `docker rm` subcommand, as shown here:

```
$ sudo docker ps -a
CONTAINER ID IMAGE         COMMAND     CREATED        STATUS
                    PORTS    NAMES
7473f2568add ubuntu:14.04 "/bin/bash" 5 seconds ago Exited
 (0) 3 seconds ago          jolly_wilson
$ sudo docker rm 7473f2568add
7473f2568add
$
```

Two `docker` subcommands, that is, `docker rm` and `docker ps`, could be combined to automatically delete all the containers that are not currently running, as shown in the following command:

```
$ sudo docker rm 'sudo docker ps -aq --no-trunc'
```

In the preceding command, the command inside the back quotes will produce a list of the full container IDs of every container, running or otherwise, which will become the argument for the `docker rm` subcommand. Unless forced with the `-f` option to do otherwise, the `docker rm` subcommand will only remove the container that is not in the running state. It will generate the following error for the running container and then continue to the next container on the list:

```
Error response from daemon: You cannot remove a running container.
Stop the container before attempting removal or use -f
```

Building images from containers

So far, we have crafted a handful of containers by using the standard base images `busybox` and `ubuntu`. In this section, let us see how we can add more software to our base image on a running container and then convert that container into an image for future use.

Let's take `ubuntu:14.04` as our base image, install the `wget` application, and then convert the running container to an image by performing the following steps:

1. Launch an `ubuntu:14.04` container by using the `docker run` subcommand, shown here:

   ```
   $ sudo docker run -i -t ubuntu:14.04 /bin/bash
   ```

2. Having launched the container, let's quickly verify if `wget` is available for our image or not. We have used the `which` command with `wget` as an argument for this purpose and, in our case, it returns empty, which essentially means that it could not find any `wget` installation in this container. This command is run as follows:

   ```
   root@472c96295678:/# which wget

   root@472c96295678:/#
   ```

3. Now let's move on to the next step which involves the `wget` installation. Since it is a brand new `ubuntu` container, before installing `wget`, we must synchronize with the `ubuntu` package repository, as shown here:

   ```
   root@472c96295678:/# apt-get update
   ```

4. Once the `ubuntu` package repository synchronization is over, we can proceed toward installing `wget`, as shown here:

   ```
   root@472c96295678:/# apt-get install -y wget
   ```

5. Having completed the `wget` installation, let's confirm our installation of `wget` by invoking the `which` command with `wget` as an argument, as shown here:

```
root@472c96295678:/#which wget
/usr/bin/wget
root@472c96295678:/#
```

6. Installation of any software would alter the base image composition, which we can also trace by using the `docker diff` subcommand introduced in *Tracking changes inside containers* section of this chapter. From a second terminal or screen, we can issue the `docker diff` subcommand, as follows:

```
$ sudo docker diff 472c96295678
```

   The preceding command would show a few hundred lines of modification to the `ubuntu` image. This modification includes the update on package repository, `wget` binary, and the support files for `wget`.

7. Finally, let's move to the most important step of committing the image. The Docker `commit` subcommand can be performed on a running or a stopped container. When commit is performed on a running container, the Docker engine will pause the container during the `commit` operation in order to avoid any data inconsistency. We strongly recommend performing the `commit` operation on a stopped container. We can commit a container to an image by the `docker commit` subcommand, as shown here:

```
$ sudo docker commit 472c96295678 \
              learningdocker/ubuntu_wget
a530f0a0238654fa741813fac39bba2cc14457aee079a7ae1fe1c64dc7e1ac
25
```

We have committed our image by using the name `learningdocker/ubuntu_wget`.

Step by step, we saw how to create an image from a container. Now, let's quickly list the images of our Docker host and see if this newly created image is a part of the image list by using the following command:

```
$ sudo docker images
REPOSITORY                      TAG                 IMAGE ID
CREATED             VIRTUAL SIZE
learningdocker/ubuntu_wget   latest              a530f0a02386
48 seconds ago      221.3 MB
```

```
busybox                         buildroot-2014.02   e72ac664f4f0
2 days ago          2.433 MB
ubuntu                          14.04               6b4e8a7373fe
2 days ago          194.8 MB
```

From the preceding `docker images` subcommand output, it is quite evident that our image creation from the container has been quite successful.

Now that you have learned how to create an image from the containers by using a few easy steps, we would encourage you to predominantly use this method for testing purposes. The most elegant and the most recommended way of creating an image is to use the `Dockerfile` method, which will be introduced in the next chapter.

Launching a container as a daemon

We have already experimented with an interactive container, tracked the changes that were made to the containers, created images from the containers and then gained insights in the containerization paradigm. Now, let's move on to understanding the real workhorse of the Docker technology. Yes that's right. In this section, we will walk you through the steps that are required for launching a container in the detached mode; in other words, we will learn about the steps that are required for launching a container as a daemon. We will also view the text that is generated in the container.

The `docker run` subcommand supports an option `-d`, which will launch a container in a detached mode, that is, it will launch a container as a daemon. For the purpose of illustration, let's resort to our date and time script, which we used in the `pause and unpause` container example, as shown here:

```
$ sudo docker run -d ubuntu \
        /bin/bash -c "while true; do date; sleep 5; done"
0137d98ee363b44f22a48246ac5d460c65b67e4d7955aab6cbb0379ac421269b
```

The `docker logs` subcommand is used for viewing the output generated by our daemon container, as shown here:

```
$ sudo docker logs \
0137d98ee363b44f22a48246ac5d460c65b67e4d7955aab6cbb0379ac421269b
Sat Oct  4 17:41:04 UTC 2014
Sat Oct  4 17:41:09 UTC 2014
Sat Oct  4 17:41:14 UTC 2014
Sat Oct  4 17:41:19 UTC 2014
```

Ankita Thakur

Your Course Guide

Your Coding Challenge

Image tags group images of the same type. Think how you can perform the following tasks. What command you'll run?

- To pull an image with a specific tag
- To pull all images corresponding to all tags
- To pull an image with a specific digest

Summary of Module 1 Chapter 4

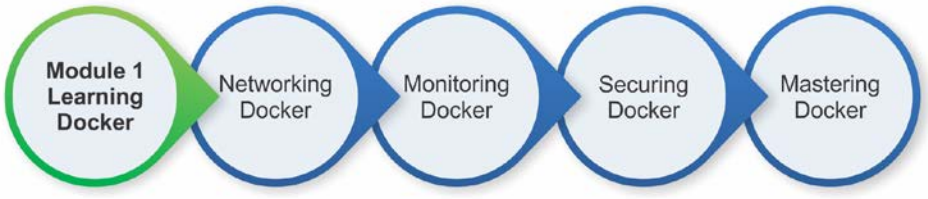
Ankita Thakur

Your Course Guide

In this chapter, we explained how to search for the images in the Docker repository. We also discussed the operation and handling of the Docker containers, how to track the changes inside the containers, how to control and housekeep the containers.

We 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.

Your Progress through the Course So Far



5
Publishing Images

In the previous chapter, we learned how to build Docker images. The next logical step is to publish these images in a public repository for public discovery and consumption. So, this chapter focuses on publishing images on the Docker Hub, and how to get the most out of the Docker Hub. We can create a new Docker image, using a `commit` command and a `Dockerfile`, build on it, and push it to the Docker Hub. The concept of a trusted repository will be discussed. This trusted repository is created from GitHub or Bitbucket. This can then be integrated with the Docker Hub to automatically build images, as a result of updates in the repository. This repository on GitHub is used to store the `Dockerfile`, which was previously created. Also, we will illustrate how worldwide organizations can enable their teams of developers to craft and contribute a variety of Docker images to be deposited in the Docker Hub. The Docker Hub REST APIs can be used for user management and manipulation of repository programmatically.

The following topics are covered in this chapter:

- How to push images to the Docker Hub
- Automatic building of images
- Private repositories on the Docker Hub
- Creating organizations on the Docker Hub
- The Docker Hub REST API

Pushing images to the Docker Hub

Here, we will create a Docker image on the local machine, and push this image to the Docker Hub. You need to perform the following steps in this section:

1. Create a Docker image on the local machine by doing one of the following:
    - Using the `docker commit` sub command
    - Using the `docker commit` sub command with `Dockerfile`
2. Push this created image to the Docker Hub.
3. Delete the image from the Docker Hub.

We will use the Ubuntu base image, run the container, add a new directory and a new file, and then create a new image. In *Chapter 3*, *Building Images*, we have seen the creation of the Docker image using `Dockerfile`. You may refer to this to check for details of the `Dockerfile` syntax.

We will run the container with the name `containerforhub` from the base `ubuntu` image, as shown in the following terminal code:

```
$ sudo docker run -i --name="containerforhub" -t ubuntu /bin/bash
root@e3bb4b138daf:/#
```

Next, we'll create a new directory and file in the `containerforhub` container. We will also update the new file with some sample text to test later:

```
root@bd7cc5df6d96:/# mkdir mynewdir
root@bd7cc5df6d96:/# cd mynewdir
root@bd7cc5df6d96:/mynewdir# echo 'this is my new container to make
image and then push to hub' >mynewfile
root@bd7cc5df6d96:/mynewdir# cat mynewfile
This is my new container to make image and then push to hub
root@bd7cc5df6d96:/mynewdir#
```

Let's build the new image with the `docker commit` command from the container, which has just been created. Note that the `commit` command would be executed from the host machine, from where the container is running, and not from inside this container:

```
$ sudo docker commit -m="NewImage" containerforhub
vinoddandy/imageforhub
3f10a35019234af2b39d5fab38566d586f00b565b99854544c4c698c4a395d03
```

Now, we have a new Docker image available on the local machine with the `vinoddandy/imageforhub` name. At this point in time, a new image with `mynewdir` and `mynewfile` is created locally.

We will log in to the Docker Hub using the `sudo docker login` command, as discussed earlier in this chapter.

Let's push this image to the Docker Hub from the host machine:

```
$ sudo docker push vinoddandy/imageforhub
The push refers to a repository [vinoddandy/imageforhub] (len: 1)
Sending image list
Pushing tag for rev [c664d94bbc55] on {https://cdn-registry-
1.docker.io/v1/repositories/vinoddandy/imageforhub/tags/latest
}
```

Now, we'll `login` to the Docker Hub and verify the image in **Repositories**.

To test the image from the Docker Hub, let's remove this image from the local machine. To remove the image, first we need to stop the container and then delete the container:

```
$ sudo docker stop containerforhub
$ sudo docker rm containerforhub
$
```

We will also delete the `vinoddandy/imageforhub` image:

```
$ sudo docker rmi vinoddandy/imageforhub
```

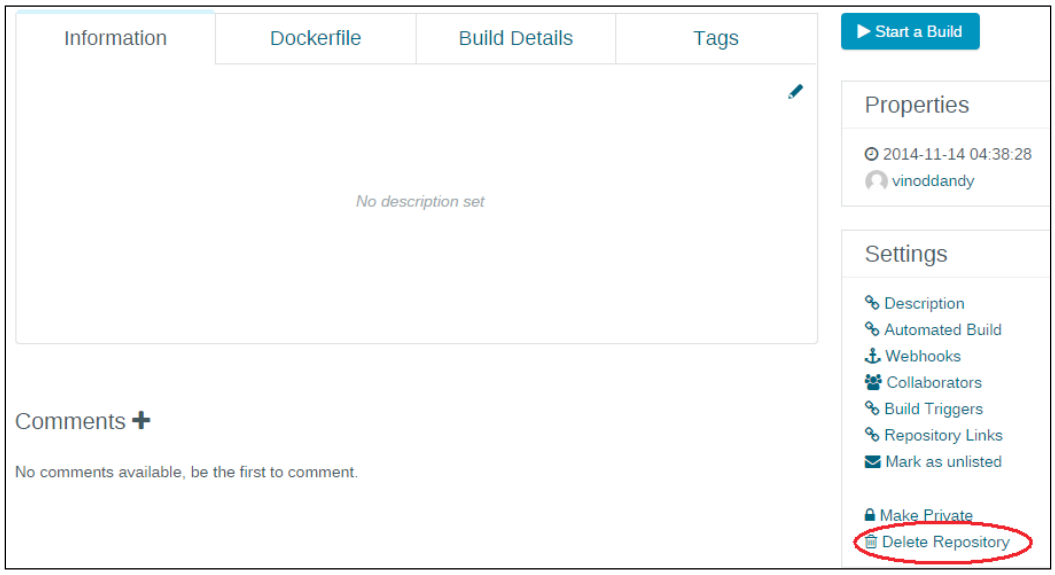
We will pull the newly created image from the Docker Hub, and run the new container on the local machine:

```
$ sudo docker run -i --name="newcontainerforhub" -t
vinoddandy/imageforhub /bin/bash
Unable to find image 'vinoddandy/imageforhub' locally
Pulling repository vinoddandy/imageforhub
c664d94bbc55: Pulling image (latest) from vinoddandy/imageforhub,
endpoint: http
c664d94bbc55: Download complete
5506de2b643b: Download complete
root@9bd40f1b5585:/# cat /mynewdir/mynewfile
This is my new container to make image and then push to hub
root@9bd40f1b5585:/#
```

So, we have pulled the latest image from the Docker Hub and created the container with the new image `vinoddandy/imageforhub`. Make a note that the `Unable to find image 'vinoddandy/imageforhub' locally` message confirms that the image is downloaded from the remote repository of the Docker Hub.

The text in `mynewfile` verifies that it is the same image, which was created earlier.

Finally, we will delete the image from the Docker Hub using `https://registry.hub.docker.com/u/vinoddandy/imageforhub/` and then click on **Delete Repository**, as shown in the following screenshot:



We'll again create this image but using the `Dockerfile` process. So, let's create the Docker image using the `Dockerfile` concept explained in *Chapter 3*, *Building Images*, and push this image to the Docker Hub.

The `Dockerfile` on the local machine is as follows:

```
##########################################
# Dockerfile to build a new image
##########################################
# Base image is Ubuntu
FROM ubuntu:14.04
# Author: Dr. Peter
```

```
MAINTAINER Dr. Peter <peterindia@gmail.com>
# create 'mynewdir' and 'mynewfile'
RUN mkdir mynewdir
RUN touch /mynewdir/mynewfile
# Write the message in file
RUN echo 'this is my new container to make image and then push to
hub' \
 >/mynewdir/mynewfile
```

Now we build the image locally using the following command:

```
$ sudo docker build -t="vinoddandy/dockerfileimageforhub" .
Sending build context to Docker daemon  2.56 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
---> 5506de2b643b
Step 1 : MAINTAINER Vinod Singh <vinod.puchi@gmail.com>
---> Running in 9f6859e2ca75
---> a96cfbf4a810
removing intermediate container 9f6859e2ca75
Step 2 : RUN mkdir mynewdir
---> Running in d4eba2a31467
---> 14f4c15610a7
removing intermediate container d4eba2a31467
Step 3 : RUN touch /mynewdir/mynewfile
---> Running in 7d810a384819
---> b5bbd55f221c
removing intermediate container 7d810a384819
Step 4 : RUN echo 'this is my new container to make image and then
push to hub'
/mynewdir/mynewfile
---> Running in b7b48447e7b3
---> bcd8f63cfa79
removing intermediate container b7b48447e7b3
successfully built 224affbf9a65
ubuntu@ip-172-31-21-44:~/dockerfile_image_hub$
```

We'll run the container using this image, as shown here:

```
$ sudo docker run -i --name="dockerfilecontainerforhub" -t
vinoddandy/dockerfileimageforhub
root@d3130f21a408:/# cat /mynewdir/mynewfile
this is my new container to make image and then push to hub
```

This text in `mynewdir` confirms that the new image is built properly with a new directory and a new file.

Repeat the `login` process, in the Docker Hub, and push this newly created image:

```
$ sudo docker login
Username (vinoddandy):
Login Succeeded
$ sudo docker push vinoddandy/dockerfileimageforhub
The push refers to a repository [vinoddandy/dockerfileimageforhub]
(len: 1)
Sending image list
Pushing repository vinoddandy/dockerfileimageforhub (1 tags)
511136ea3c5a: Image already pushed, skipping
d497ad3926c8: Image already pushed, skipping
b5bbd55f221c: Image successfully pushed
bcd8f63cfa79: Image successfully pushed
224affbf9a65: Image successfully pushed
Pushing tag for rev [224affbf9a65] on
{https://cdn-registry-1.docker.io/v1/repos
itories/vinoddandy/dockerfileimageforhub/tags/latest}
$
```

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

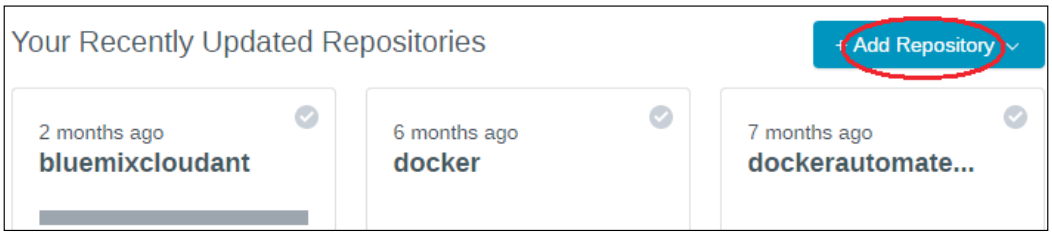
Automating the building process for images

We learnt how to build images locally and push those images to the Docker Hub. The Docker Hub also has the capability to automatically build the image from `Dockerfile` kept in the repository of GitHub or Bitbucket. Automated builds are supported on both private and public repositories of GitHub and Bitbucket. The Docker Hub Registry keeps all the automated build images. The Docker Hub Registry is based on open source and can be accessed from `https://github.com/docker/docker-registry`.

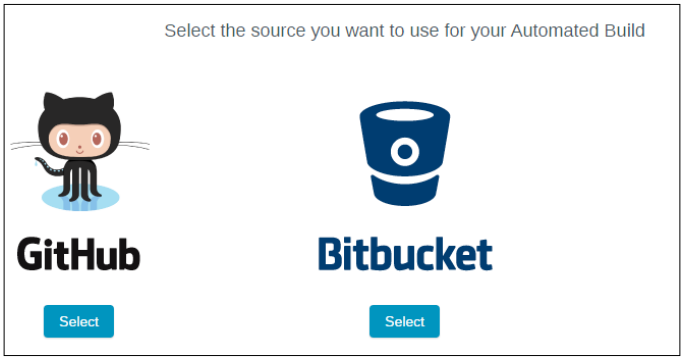
We will discuss the steps needed to implement the automated build process:

1. We first connect the Docker Hub to my GitHub account.

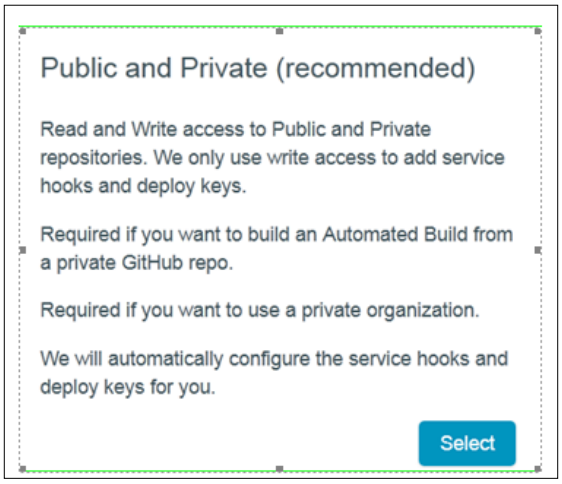
   Login to the Docker Hub, and click on **View Profile** and then navigate to **Add Repository** | **Automated Build**, as shown in the following screenshot:



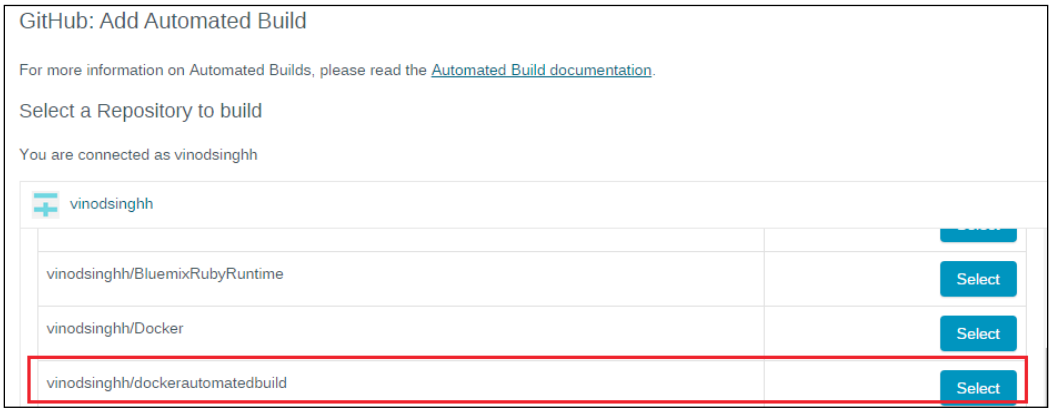
2. We now select **GitHub**:



3. Once **GitHub** is selected, it will ask for authorization. Here, we will select **Public and Private**, as shown here:

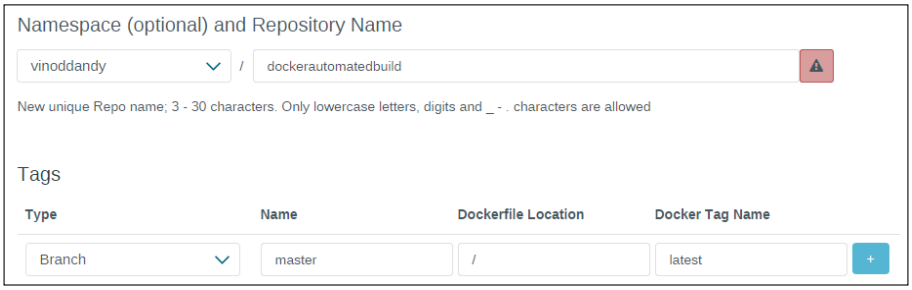


4. After clicking on **Select**, it will now show your GitHub repository:

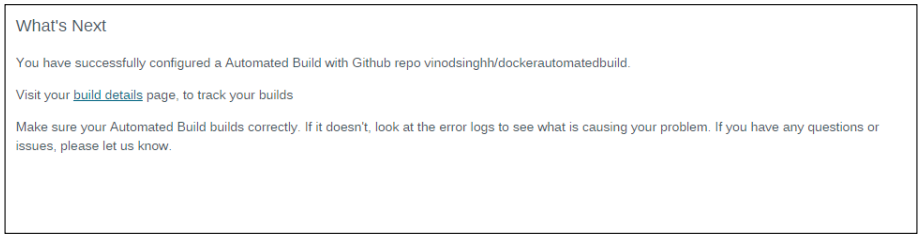


5. Click on the **Select** button of your repository **vinodsinghh/ dockerautomationbuild**, shown in the preceding screenshot:

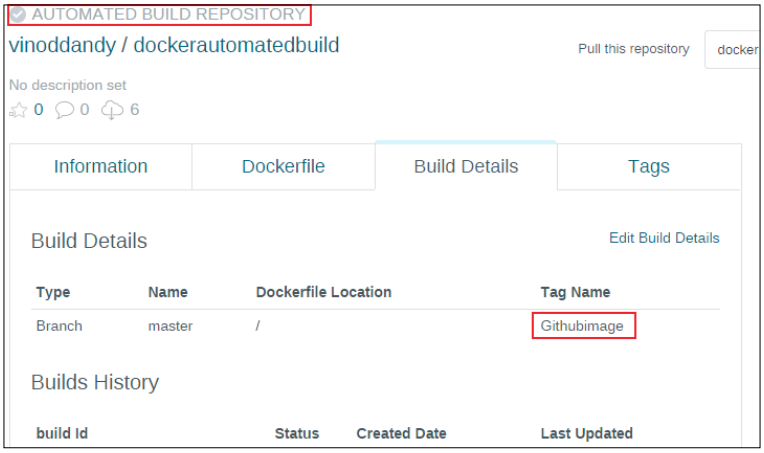
6. We choose the default branch and update the tag with `Githubimage`. Also, we will keep the location as its default value, which is the root of our Docker Hub, as shown in the following screenshot:



7. Finally, we will click on **Create Repository**, as shown in the preceding screenshot:



8. Click on **build details** to track your build status, as shown in the preceding screenshot. It will lead you to the following screenshot:



So, whenever the `Dockerfile` is updated in GitHub, the automated build gets triggered, and a new image will be stored in the Docker Hub Registry. We can always check the build history. We can change the `Dockerfile` on the local machine and push to GitHub. Then, we can see the automated build link of the Docker Hub at `https://registry.hub.docker.com/u/vinoddandy/dockerautomatedbuild/builds_history/82194/`.

Private repositories on the Docker Hub

The Docker Hub provides both a public and private repository. The public repository is free to users and private is a paid service. The plans with private repositories are available in different sizes, such as a micro, small, medium, or large subscription.

Docker has published their public repository code to open source at `https://github.com/docker/docker-registry`.

Normally, enterprises will not like to keep their Docker images either in a Docker public or private repository. They prefer to keep, maintain, and support their own repository. Hence, Docker also provides the option for enterprises to create and install their own repository.

Let's create a repository in the local machine using the registry image provided by Docker. We will run the registry container on the local machine, using the registry image from Docker:

```
$ sudo docker run -p 5000:5000 -d registry
768fb5bcbe3a5a774f4996f0758151b1e9917dec21aedf386c5742d44beafa41
```

In the automated build section, we built the `vinoddandy/dockerfileforhub` image. Let's tag the image ID `224affbf9a65` to our locally created `registry` image. This tagging of the image is needed for unique identification inside the local repository. This image `registry` may have multiple variants in the repository, so this `tag` will help you identify the particular image:

```
$ sudo docker tag
224affbf9a65localhost:5000/vinoddandy/dockerfileimageforhub
```

Once the tagging is done, push this image to a new registry using the `docker push` command:

```
$ sudo docker push localhost:5000/vinoddandy/dockerfile
imageforhub
```

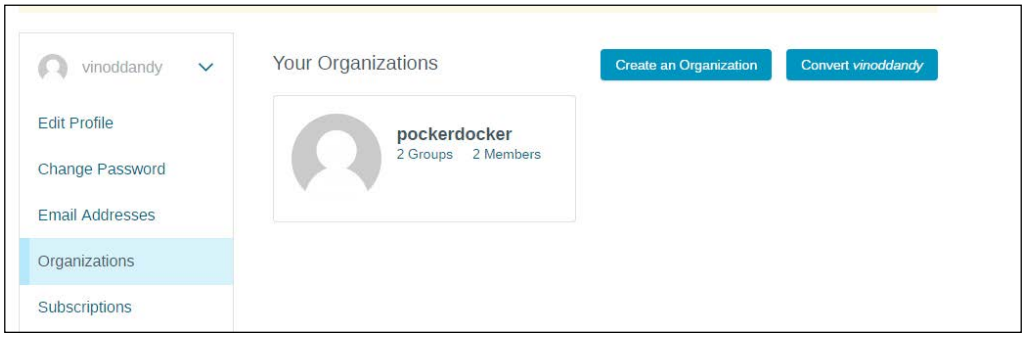
```
The push refers to a repository
[localhost:5000/vinoddandy/dockerfileimageforhub
] (len: 1)
Sending image list
Pushing repository localhost:5000/vinoddandy/dockerfileimageforhub (1
tags)
511136ea3c5a: Image successfully pushed
d497ad3926c8: Image successfully pushed
--------------------------------------------------
224affbf9a65: Image successfully pushed
Pushing tag for rev [224affbf9a65] on
{http://localhost:5000/v1/repositories/vin
oddandy/dockerfileimageforhub/tags/latest}
ubuntu@ip-172-31-21-44:~$
```

Now, the new image is available in the local repository. You can now retrieve this image from the local registry and run the container. This task is left for you to complete.

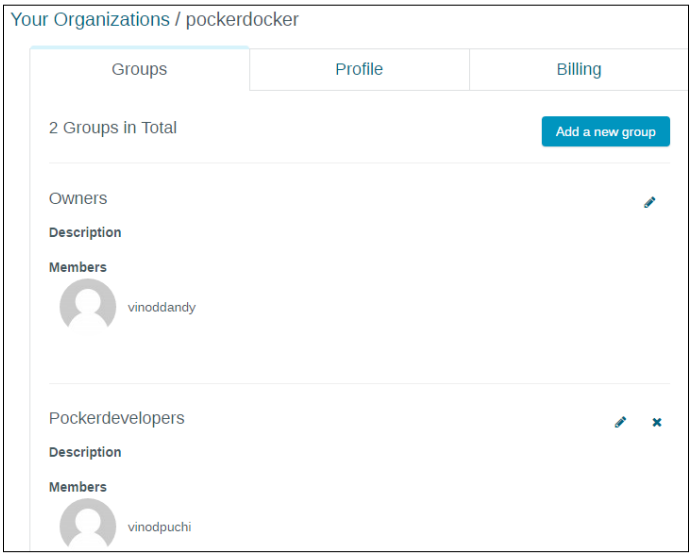
Organizations and teams on the Docker Hub

One of the useful aspects of private repositories is that you can share them only with members of your organization or team. The Docker Hub lets you create organizations, where you can collaborate with your colleagues and manage private repositories. You can learn how to create and manage an organization.

The first step is to create an organization on the Docker Hub, as shown in the following screenshot:



Inside your organization, you can add more organizations, and then add members to it:



The members of your organization and group can collaborate with the organization and teams. This feature would be more useful in case of a private repository.

The REST APIs for the Docker Hub

The Docker Hub provides a REST API to integrate the Hub capabilities through programs. The REST API is supported for both user as well as repository management.

User management supports the following features:

- **User Login**: This is used for user login to the Docker Hub:

```
GET /v1/users
```

```
$ curl --raw -L --user vinoddandy:password
https://index.docker.io/v1/users
4
"OK"
0
$
```

- **User Register**: This is used for registration of a new user:

  `POST /v1/users`

- **Update user**: This is used to update the user's password and e-mail:

  `PUT /v1/users/(usename)/`

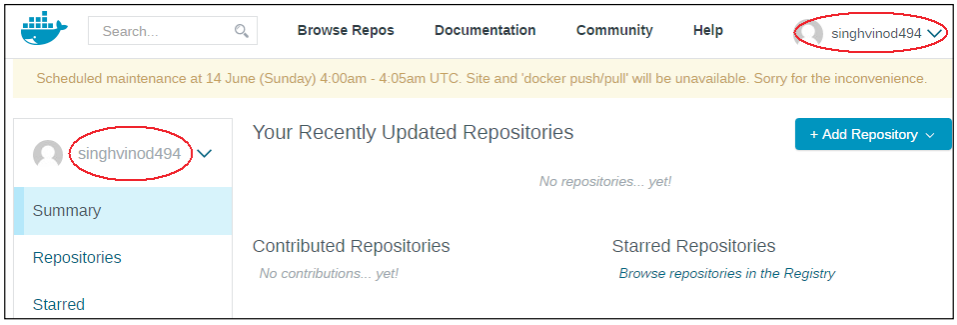
Repository management supports the following features:

- **Create a user repository**: This creates a user repository:

  `PUT /v1/repositories/(namespace)/(repo_name)/`

  ```
  $ curl --raw -L -X POST --post301 -H "Accept:application/json"
  -H "Content-Type: application/json" --data-ascii '{"email":
  "singh_vinod@yahoo.com", "password": "password", "username":
  "singhvinod494" }' https://index.docker.io/v1/users
  e
  "User created"
  0
  ```

  After you create repositories, your repositories will be listed here, as shown in this screenshot:



- **Delete a user repository**: This deletes a user repository:

  `DELETE /v1/repositories/(namespace)/(repo_name)/`

- **Create a library repository**: This creates the library repository, and it is available only to Docker administrators:

  `PUT /v1/repositories/(repo_name)/`

- **Delete a library repository**: This deletes the library repository, and it is available only to Docker administrators:

  ```
  DELETE /v1/repositories/(repo_name)/
  ```

- **Update user repository images**: This updates the images of a user's repository:

  ```
  PUT /v1/repositories/(namespace)/(repo_name)/images
  ```

- **List user repository images**: This lists the images of a user's repository:

  ```
  GET /v1/repositories/(namespace)/(repo_name)/images
  ```

- **Update library repository images**: This updates the images of a library repository:

  ```
  PUT /v1/repositories/(repo_name)/images
  ```

- **List library repository images**: This lists the images of a library repository:

  ```
  GET /v1/repositories/(repo_name)/images
  ```

- **Authorize a token for a library repository**: This authorizes a token for a library repository:

  ```
  PUT /v1/repositories/(repo_name)/auth
  ```

- **Authorize a token for a user repository**: This authorizes a token for a user's repository:

  ```
  PUT /v1/repositories/(namespace)/(repo_name)/auth
  ```

Summary of Module 1 Chapter 5

In this chapter, we looked deep into setting up automated builds. We took a look at how to set up your own Docker Hub Registry.

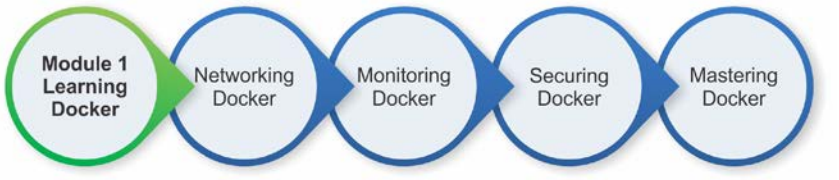
The, we saw how to push images to the Docker Hub. Docker images are the most prominent building blocks used for deriving real-world Docker containers that can be exposed as a service over any network. Developers can find and check images for their unique capabilities, and use them accordingly for their own purposes in bringing up highly usable, publicly discoverable, network-accessible, and cognitively composable containers. All the crafted images need to be put in a public registry repository. In this chapter, we clearly explained how to publish images in a repository. We also talked about the trusted repositories and their distinct characteristics. Finally, we demonstrated how the REST API for the repository can be leveraged to push in and play around with Docker images and user management programmatically.

Ankita Thakur

Your Course Guide

The Docker images need to be stored in a public, controlled, and network-accessible location to be readily found and leveraged by worldwide software engineers and system administrators. The Docker Hub is being touted as the best-in-class method for centrally aggregating, curating, and managing Docker images, originating from Docker enthusiasts (internal as well as external). However, enterprises cannot afford to keep their Docker images in a public domain, and hence, the next chapter is dedicated to expose the steps needed for image deployment and management in private IT infrastructures.

Your Progress through the Course So Far



6
Running Your Private Docker Infrastructure

In *Chapter 4*, *Publishing Images,* we discussed Docker images and clearly understood that Docker containers are the runtime implementations of Docker images. Docker images and containers are in plenty these days, as the containerization paradigm has taken the IT domain by storm. Therefore, there is a need for worldwide enterprises to keep their Docker images in their own private infrastructure for security considerations. So, the concept of deploying a Docker Hub to our own infrastructure has emerged and evolved. Docker Hubs are paramount and pertinent to register and then deposit the growing array of Docker images. Primarily, a Docker Hub is specially made to centralize and centrally manage information on:

- User accounts
- Checksums of the images
- Public namespaces

This chapter is developed with a focus on providing all the relevant information to enable you and Docker container crafters to design, populate, and run their own private Docker Hubs in their own backyards. This chapter covers the following important topics:

- The Docker registry and index
- Docker registry use cases
- Run your own index and registry
- Push the image to a newly created registry

The Docker registry and index

Typically, a Docker Hub consists of a Docker index and registry. Docker clients can connect and interact with the Docker Hubs over a network. The registry has the following characteristics:

- It stores the images and graphs for a set of repositories
- It does not have user accounts data
- It has no notion of user accounts or authorization
- It delegates the authentication and authorization to the Docker Hub Authentication service
- It supports different storage backends (S3, cloud files, local filesystem, and so on)
- It doesn't have a local database
- It has a source code associated with it

The advanced features of the Docker registry include `bugsnag`, `new relic`, and `cors`. The `bugsnag` feature detects and diagnoses crashes in applications, `new relic` encapsulates the registry and monitors performance, and `cors` can be enabled to share resources outside our own registry domain. It is recommended that you deploy the registry to production environments using a proxy, such as nginx. You can also run the Docker registry directly on Ubuntu and Red Hat Linux-based systems.

Currently, the firm in charge of developing the Docker platform has released the Docker registry as an open source service on GitHub at `https://github.com/docker/docker-registry`. It is important to note that the Docker index is only a recommendation and nothing has been released by Docker as an open source project at the time of writing this book. In this chapter, we will start with a use case of the Docker registry, and then start with the actual deployment of the index elements and the Docker registry from GitHub.

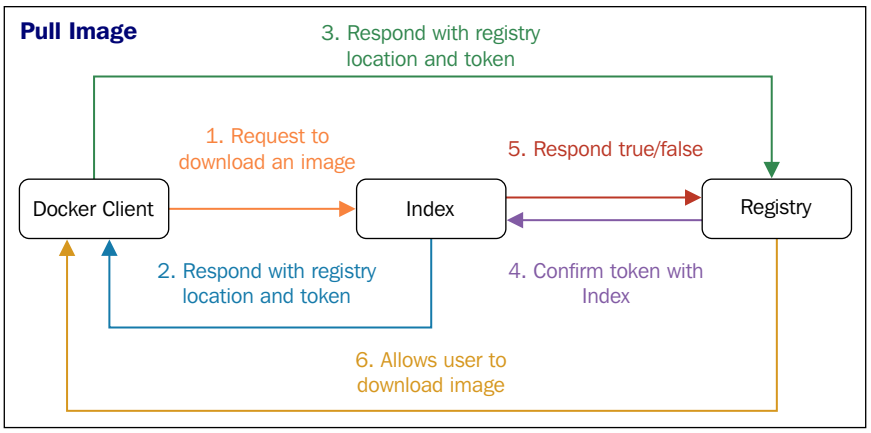
Docker registry use cases

The following are the use cases of the Docker registry:

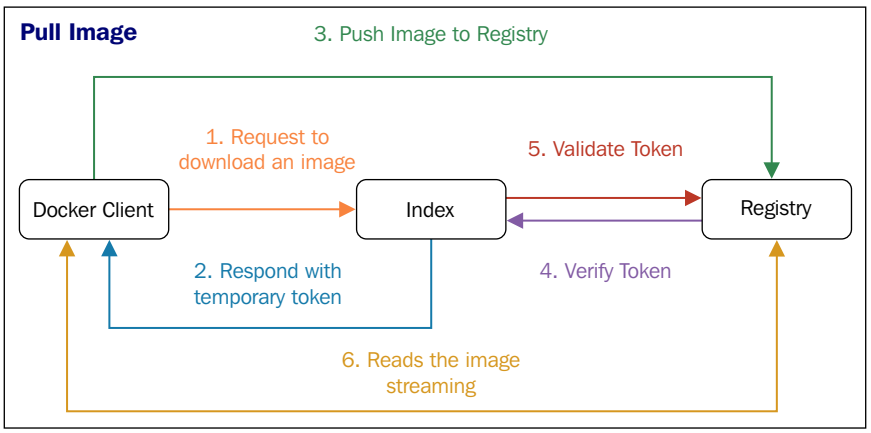
1. Pull or download an image
2. Push an image
3. Delete an image

We will now go through each of these use cases in detail:

1. **Pull or download an image**: The user requests an image using the Docker client from the index, the index, in turn responds back to the user with the registry details. Then, the Docker client will directly request the registry to get the required image. The registry authenticates the user with an index internally. As shown in the following diagram, image pulling is accomplished with the collaboration of the client, index, and registry modules:



2. **Push an image**: A user requests to push the image, gets the registry information from the index, and then pushes the image directly to the registry. The registry authenticates the user with the index and, finally, responds to the user. The control flow is illustrated in the following diagram:



3. **Delete an image**: The user can also request to delete an image from the repository.

   The user has the option to use the registry with or without the index. Use of the registry, without the index, is best suited for storing private images.

Run your own index and registry

In this section, we will perform the following steps to run our own index and registry, and finally, push the image:

1. Deployment of the index components and the registry from GitHub.
2. Configuration of nginx with the Docker Registry.
3. Set up SSL on the web server for secure communication.

Step 1 – Deployment of the index components and the registry from GitHub

The index components include `apache-utils` and `ngnix` for password authentication and the SSL feature for HTTPS support. The user must note that the current version of the Docker registry supports only HTTP to connect to the registry. So, it is mandatory for the user to deploy and use **Secure Sockets Layer** (**SSL**) to secure the data. SSL creates an encrypted connection between a web server and the client's web browser that allows private data to be transmitted without the issues of eavesdropping, data tampering, or message forgery. This is a proven way of securing the data using SSL certificates that is widely accepted.

The Docker registry is a Python application, and we can install Python on the local Ubuntu machine from `https://github.com/docker/docker-registry`, using the following command:

```
$ sudo apt-get -y install build-essential python-dev \
       libevent-dev python-pip liblzma-dev swig libssl-dev
```

Now, install the Docker registry:

```
$ sudo pip install docker-registry
```

This will update the Docker registry in the Python package and update the configuration file in the following path:

```
$ cd /usr/local/lib/python2.7/dist-
packages/config/
```

Copy the `config_sample.yml` file to `config.yml`:

```
$ sudo cp config_sample.yml config.yml
```

Docker, by default, saves its data in the `/tmp` directory, which can create problems because the `/tmp` folder is cleared on reboot on many Linux systems. Let's create a permanent folder to store our data:

```
$ sudo mkdir /var/docker-registry
```

Let's update our preceding `config.yml` file for this updated path for the following two locations. The updated code for the first location will look like this:

```
sqlalchemy_index_database:
    _env:SQLALCHEMY_INDEX_DATABASE:sqlite:////var/docker-
registry/docker-registry.db
```

The following is the code for the second location:

```
local: &local
    storage: local
    storage_path: _env:STORAGE_PATH:/var/docker-registry/registry
```

The other default configuration of the `config.yml` file works fine.

Now, let's start the Docker registry using `gunicorn`. Gunicorn, also known as Green Unicorn, is a Python **Web Server Gateway Interface** (**WSGI**) HTTP server for Linux systems:

```
$ sudo gunicorn --access-logfile - --debug -k gevent -b \
       0.0.0.0:5000 -w 1 docker_registry.wsgi:application
```

```
01/Dec/2014:04:59:23 +0000 WARNING: Cache storage disabled!
01/Dec/2014:04:59:23 +0000 WARNING: LRU cache disabled!
01/Dec/2014:04:59:23 +0000 DEBUG: Will return docker-
registry.drivers.file.Storage
```

Now, the Docker registry is up and running as a process on the user's local machine.

We can stop this process using *Ctrl* + *C*.

We can start a Linux service as follows:

1. Make a directory for the `docker-registry` tool:
   ```
   $ sudo mkdir -p /var/log/docker-registry
   ```
2. Create and update the file for the Docker registry configuration:
   ```
   $ sudo vi /etc/init/docker-registry.conf
   ```
3. Update the following content in the file:
   ```
   description "Docker Registry"
   start on runlevel [2345]
   stop on runlevel [016]
   respawn
   respawn limit 10 5
   script
   exec gunicorn --access-logfile /var/log/docker-
   registry/access.log --error-logfile /var/log/docker-
   registry/server.log -k gevent --max-requests 100 --
   graceful-timeout 3600 -t 3600 -b localhost:5000 -w 8
   docker_registry.wsgi:application
   end script
   ```
4. After saving the file, run the Docker registry service:
   ```
   $ sudo service docker-registry start
   docker-registry start/running, process 25760
   ```
5. Now secure this registry using `apache-utils`, by enabling the password protected feature, as shown here:
   ```
   $ sudo apt-get -y install nginx apache2-utils
   ```
6. The user creates a login ID and password to access the Docker registry:
   ```
   $ sudo htpasswd -c /etc/nginx/docker-registry.htpasswd vinod1
   ```
7. Enter the new password when prompted. At this point, we have the login ID and password to access the Docker registry.

Step 2 – Configuration of nginx with the Docker registry

Next, we need to tell nginx to use that authentication file (created in step 6 and step 7 of the previous section) to forward requests to our Docker registry.

We need to create the nginx configuration file. To do this, we need to follow these steps:

1. We create the ngnix configuration file by running the following command:

   ```
   $ sudo vi /etc/nginx/sites-available/docker-registry
   ```

   Update the file with the following content:

```
upstream docker-registry {
 server localhost:5000;
}
server {
 listen 8080;
 server_name my.docker.registry.com;
 # ssl on;
 # ssl_certificate /etc/ssl/certs/docker-registry;
 # ssl_certificate_key /etc/ssl/private/docker-registry;
 proxy_set_header Host       $http_host;   # required for
Docker client sake
 proxy_set_header X-Real-IP  $remote_addr; # pass on real
client IP
 client_max_body_size 0; # disable any limits to avoid HTTP
413 for large image uploads
 # required to avoid HTTP 411: see Issue #1486
(https://github.com/dotcloud/docker/issues/1486)
 chunked_transfer_encoding on;
 location / {
     # let Nginx know about our auth file
     auth_basic              "Restricted";
     auth_basic_user_file    docker-registry.htpasswd;
     proxy_pass http://docker-registry;
 } location /_ping {
     auth_basic off;
     proxy_pass http://docker-registry;
 }   location /v1/_ping {
     auth_basic off;
     proxy_pass http://docker-registry;
 }
}
```

2. Make the soft link and restart the ngnix service:

```
$ sudo ln -s /etc/nginx/sites-available/docker-registry  \
        /etc/nginx/sites-enabled/docker-registry
$ sudo service nginx restart
```

3. Let's check whether everything works fine. Run the following command, and we should get this output:

```
$ sudo curl localhost:5000
"\"docker-registry server\""
```

Great! So now we have the Docker registry running. Now, we have to check whether nginx worked as we expected it to. To do this, run the following command:

```
$ curl localhost:8080
```

This time, we will get an unauthorized message:

```
<html>
<head><title>401 Authorization Required</title></head>
<body bgcolor="white">
<center><h1>401 Authorization Required</h1></center>
<hr><center>nginx/1.4.6 (Ubuntu)</center>
</body>
</html>
```

Let's log in using the password created earlier:

```
$ curl vinod1:vinod1@localhost:8080
"\"docker-registry server\""ubuntu@ip-172-31-21-44:~$
```

This confirms that your Docker registry is password protected.

Step 3 – Set up SSL on the web server for secure communication

This is the final step to set up SSL on a local machine, which hosts the web server for the encryption of data. We create the following file:

```
$sudo vi /etc/nginx/sites-available/docker-registry
```

Update the file with the following content:

```
server {
 listen 8080;
```

```
    server_name mydomain.com;
    ssl on;
    ssl_certificate /etc/ssl/certs/docker-registry;
    ssl_certificate_key /etc/ssl/private/docker-registry;
```

Note that my Ubuntu machine is available on the Internet with the name `mydomain.com` and SSL is set up with the path for a certificate and key.

Let's sign the certificate as follows:

```
$ sudo mkdir ~/certs
$ sudo cd ~/certs
```

The root key is generated using `openssl`, using the following command:

```
$ sudo openssl genrsa -out devdockerCA.key 2048
Generating RSA private key, 2048 bit long modulus
..........+++
.....................+++
e is 65537 (0x10001)
```

Now we have the root key, let's generate a root certificate (enter whatever you'd like to at the Command Prompt):

```
$ sudo openssl req -x509 -new -nodes -key devdockerCA.key -days  \
         10000 -out devdockerCA.crt
```

Then, generate a key for our server:

```
$ sudo openssl genrsa -out dev-docker-registry.com.key 2048
```

Now, we have to make a certificate signing request. Once we run the signing command, ensure that `Common Name` is our server name. This is mandatory and any deviation will result in an error:

```
$ sudo openssl req -new -key dev-docker-registry.com.key -out \
   dev-docker-registry.com.csr
```

Here, `Common Name` looks like `mydomain.com`. This is an Ubuntu VM running on AWS.

The output of the preceding command is shown as follows:

```
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
```

```
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:mydomain.com
Email Address []:
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

The `challenge password` input is kept blank, and the user is also free to populate this. Then, we need to sign the certificate request, by running the following command:

```
$ sudo openssl x509 -req -in dev-docker-registry.com.csr -CA  \
   devdockerCA.crt -CAkey devdockerCA.key -CAcreateserial -out \
    dev-docker-registry.com.crt -days 10000
```

Now that we've generated all the files we need for our certificate to work, we need to copy these files to the correct places.

First, copy the certificate and key to the paths where nginx is expecting them to be:

```
$ sudo cp dev-docker-registry.com.crt /etc/ssl/certs/docker-registry
$ sudo chmod 777 /etc/ssl/certs/docker-registry
$ sudo cp dev-docker-registry.com.key /etc/ssl/private/docker-registry
$ sudo chmod 777 /etc/ssl/private/docker-registry
```

Note that we have created self-signed certificates, and they are signed by any known certificate authority, so we need to inform the registry that this is a legitimate certificate:

```
$ sudo mkdir /usr/local/share/ca-certificates/docker-dev-cert
$ sudo cp devdockerCA.crt /usr/local/share/ca-certificates/docker-
dev-cert
$ sudo update-ca-certificates
Updating certificates in /etc/ssl/certs... 1 added, 0 removed; done.
Running hooks in /etc/ca-certificates/updated....done.
ubuntu@ip-172-31-21-44:~/certs$
```

Let's restart nginx to reload the configuration and SSL keys:

```
$ sudo service nginx restart
```

Now, we will test the SSL certificate to check whether it works fine. Since `mydomain.com` is not an Internet address, add the entry in `/etc/hosts` file:

```
172.31.24.44 mydomain.com
```

Now run the following command:

```
$ sudo curl https://vinod1:vinod1@mydomain.com:8080
"\"docker-registry server\""ubuntu@ip-172-31-21-44:~$
```

So if all went well, you should see something like this:

```
"docker-registry server"
```

Push the image to the newly created Docker registry

Finally, push the image to the Docker registry. So, let's create an image on the local Ubuntu machine:

```
$ sudo docker run -t -i ubuntu /bin/bash
root@9593c56f9e70:/# echo "TEST" >/mydockerimage
root@9593c56f9e70:/# exit
$ sudo docker commit $(sudo docker ps -lq) vinod-image
e17b685ee6987bb0cd01b89d9edf81a9fc0a7ad565a7e85650c41fc7e5c0cf9e
```

Let's log in to the Docker registry created locally on the Ubuntu machine:

```
$ sudo docker --insecure-registry=mydomain.com:8080 \
    login https://mydomain.com:8080
Username: vinod1
Password:
```

```
Email: vinod.puchi@gmail.com
Login Succeeded
```

Tag the image before pushing it to the registry:

```
$ sudo docker tag vinod-image mydomain.com:8080/vinod-image
```

Finally, use the `push` command to upload the image:

```
$ sudo docker push \
mydomain.com:8080/vinod-image
The push refers to a repository [mydomain.com
:8080/vinod-image] (len: 1)
Sending image list
Pushing repository mydomain.com:8080/vi
nod-image (1 tags)
511136ea3c5a: Image successfully pushed
5bc37dc2dfba: Image successfully pushed
--------------------------------------------------
e17b685ee698: Image successfully pushed
Pushing tag for rev [e17b685ee698] on {https://mydomain.com
:8080/v1/repositories/vinod-image/tags/latest}
$
```

Now, remove the image from the local disk and `pull` it from the Docker registry:

```
$ sudo docker pull mydomain.com:8080/vinod-image
Pulling repository mydomain.com:8080/vi
nod-image
e17b685ee698: Pulling image (latest) from mydomain.com
17b685ee698: Download complete
dc07507cef42: Download complete
86ce37374f40: Download complete
Status: Downloaded newer image for mydomain.com:8080/vinod-image:latest
$
```

Ankita Thakur

Your Course Guide

Your Coding Challenge

Here are some practice questions for you to check whether you have understood the concepts:

- What are the few recommendations, which we discussed in this chapter, on how to avoid the common pitfalls?
- What are the three different formats of the -v option that we discussed?

Ankita Thakur

Your Course Guide

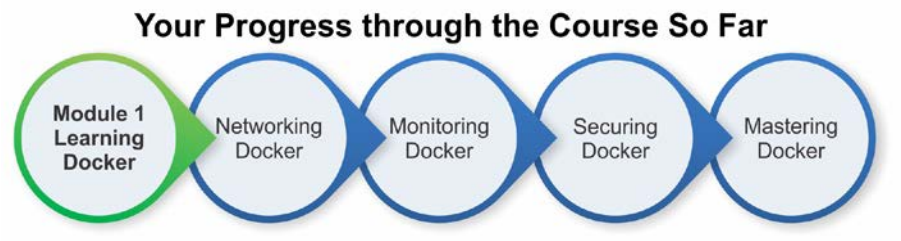
Summary of Module 1 Chapter 6

The Docker engine allows every value-adding software solution to be containerized, indexed, registered, and stocked. Docker is turning out to be a great tool for systematically developing, shipping, deploying, and running containers everywhere.

While docker.io lets you upload your Docker creations to their registry for free, anything you upload there is publicly discoverable and accessible. Innovators and companies aren't keen on this and therefore, insist on for private Docker Hubs.

In this chapter, we explained all the steps, syntaxes, and semantics for you in an easy-to-understand manner. We saw how to retrieve images to generate Docker containers and described how to push our images to the Docker registry in a secure manner in order to be found and used by authenticated developers. The authentication and authorization mechanisms, a major part of the whole process, have been explained in detail. Precisely speaking, this chapter is conceived and concretized as a guide for setting up your own Docker Hubs. As world organizations are showing exemplary interest in having containerized clouds, private container hubs are becoming more essential.

In the next chapter, we will look at securing Docker with third-party tools and learn which third-party tools, beyond those offered by Docker, are out there to help secure your environments to help keep your application(s) secure when running on Docker.


Your Progress through the Course So Far
Module 1 Learning Docker
Networking Docker
Monitoring Docker
Securing Docker
Mastering Docker

7
Running Services in a Container

We've reached so far, brick by brick, laying a strong and stimulating foundation on the fast-evolving Docker technology. We talked about the important building blocks of the highly usable and reusable Docker images. Further on, you can read the easy-to-employ techniques and tips on how to store and share Docker images through a well-designed storage framework. As usual, images will have to go through a series of verifications, validations, and refinements constantly in order to make them more right and relevant for the aspiring development community. In this chapter, we are going to take our learning to the next level by describing the steps towards creating a small web server, run the same inside a container, and connect to the web server from the external world.

In this chapter, we will cover the following topics:

- Container networking
- **Container as a Service** (**CaaS**)—building, running, exposing, and connecting to container services
- Publishing and retrieving container ports
- Binding a container to a specific IP address
- Auto-generating the Docker host port
- Port binding using the `EXPOSE` and `-P` options

A brief overview of container networking

Like any computing node, the Docker containers need to be networked, in order to be found and accessible by other containers and clients. In a network, generally, any node is being identified through its IP address. Besides, the IP address is a unique mechanism for any client to reach out to the services offered by any server node. Docker internally uses Linux capabilities to provide network connectivity to containers. In this section, we are going to learn about the container's IP address assignment and the procedure to retrieve the container's IP address.

The Docker engine seamlessly selects and assigns an IP address to a container with no intervention from the user, when it gets launched. Well, you might be puzzled on how Docker selects an IP address for a container, and this puzzle is answered in two parts, which is as follows:

1. During the installation, Docker creates a virtual interface with the name `docker0` on the Docker host. It also selects a private IP address range, and assigns an address from the selected range to the `docker0` virtual interface. This selected IP address is always outside the range of the Docker host IP address in order to avoid an IP address conflict.
2. Later, when we spin up a container, the Docker engine selects an unused IP address from the IP address range selected for the `docker0` virtual interface. Then, the engine assigns this IP address to the freshly spun container.

Docker, by default, selects the IP address `172.17.42.1/16`, or one of the IP addresses that is within the range `172.17.0.0` to `172.17.255.255`. Docker will select a different private IP address range if there is a direct conflict with the `172.17.x.x` addresses. Perhaps, the good old `ifconfig` (the command to display the details of the network interfaces) comes in handy here to find out the IP address assigned to the virtual interface. Let's just run `ifconfig` with `docker0` as an argument, as follows:

```
$ ifconfig docker0
```

The second line of the output will show the assigned IP address and its netmask:

```
inet addr:172.17.42.1  Bcast:0.0.0.0  Mask:255.255.0.0
```

Apparently, from the preceding text, `172.17.42.1` is the IP address assigned to the `docker0` virtual interface. The IP address `172.17.42.1` is one of the addresses in the private IP address range from `172.17.0.0` to `172.17.255.255`.

It's now imperative that we learn how to find the IP address assigned to a container. The container should be launched in an interactive mode using the `-i` option. Of course, we can easily find the IP by running the `ifconfig` command within the container, as shown here:

```
$ sudo docker run -i -t ubuntu:14.04 /bin/bash
root@4b0b567b6019:/# ifconfig
```

The `ifconfig` command will display the details of all the interfaces in the Docker container, as follows:

```
eth0      Link encap:Ethernet  HWaddr e6:38:dd:23:aa:3f
          inet addr:172.17.0.12  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::e438:ddff:fe23:aa3f/64 Scope:Link
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:6 errors:0 dropped:2 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:488 (488.0 B)  TX bytes:578 (578.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Evidently, the preceding output of the `ifconfig` command shows that the Docker engine has virtualized two network interfaces for the container, which are as follows:

- The first one is an `eth0` (Ethernet) interface for which the Docker engine assigned the IP address `172.17.0.12`. Obviously, this address also falls within the same IP address range of the `docker0` virtual interface. Besides, the address assigned to the `eth0` interface is used for intra-container communication and host-to-container communication.
- The second interface is the `lo` (Loopback) interface for which the Docker engine assigned the loopback address `127.0.0.1`. The loopback interface is used for local communication within a container.

Easy, isn't it? However, the retrieval of an IP address gets complicated when the container is launched in the detached mode, using the `-d` option in the `docker run` subcommand. The primary reason for this complication in the detached mode is that there is no shell prompt to run the `ifconfig` command. Fortunately, Docker provides a `docker inspect` subcommand, which is as handy as a Swiss army knife, and allow us to dive deep into the low-level details of the Docker container or image. The `docker inspect` subcommand generates the requested details in the JSON array format.

Here is a sample run of the `docker inspect` subcommand on the interactive container that we previously launched. The `4b0b567b6019` container ID is taken from the prompt of the container:

```
$ sudo docker inspect 4b0b567b6019
```

This command generates quite a lot of information about the container. Here, we show some excerpts of the container's network configuration from the output of the `docker inspect` subcommand:

```
"NetworkSettings": {
        "Bridge": "docker0",
        "Gateway": "172.17.42.1",
        "IPAddress": "172.17.0.12",
        "IPPrefixLen": 16,
        "PortMapping": null,
        "Ports": {}
    },
```

Here, the network configuration lists out the following details:

- **Bridge**: This is the bridge interface to which the container is bound
- **Gatewa**y: This is the gateway address of the container, which is the address of the bridge interface as well
- **IPAddres**s: This is the IP address assigned to the container
- **IPPrefixLen**: This is the IP prefix length, another way of representing the subnet mask
- **PortMapping**: This is the port mapping field, which is now being deprecated, and its value is always null
- **Ports**: This is the ports field that will enumerate all the port binds, which is introduced later in this chapter

There is no doubt that the `docker inspect` subcommand is quite convenient for finding the minute details of a container or an image. However, it's a tiresome job to go through the intimidating details and to find the right information that we are keenly looking for. Perhaps, you can narrow it down to the right information, using the `grep` command. Or even better, the `docker inspect` subcommand, which helps you pick the right field from the JSON array using the `--format` option of the `docker inspect` subcommand.

Notably, in the following example, we use the `--format` option of the `docker inspect` subcommand to retrieve just the IP address of the container. The IP address is accessible through the `.NetworkSettings.IPAddress` field of the JSON array:

```
$ sudo docker inspect \
  --format='{{.NetworkSettings.IPAddress}}' 4b0b567b6019
172.17.0.12
```

Envisaging the Container as a Service

We laid a good foundation of the fundamentals of the Docker technology.
In this section, we are going to focus on crafting an image with the HTTP service, launch the HTTP service inside the container using the crafted image, and then, demonstrate the connectivity to the HTTP service running inside the container.

Building an HTTP server image

In this section, we are going to craft a Docker image in order to install `Apache2` on top of the `Ubuntu 14.04` base image, and configure a `Apache HTTP Server` to run as an executable, using the `ENTRYPOINT` instruction.

In *Chapter 3*, *Building Images*, we illustrated the concept of the Dockerfile to craft an `Apache2` image on top of the `Ubuntu 14.04` base image. Here, in this example, we are going to extend this Dockerfile by setting the `Apache` log path and setting `Apache2` as the default execution application, using the `ENTRYPOINT` instruction. The following is a detailed explanation of the content of `Dockerfile`.

We are going to build an image using `ubuntu:14.04` as the base image, using the `FROM` instruction, as shown in the `Dockerfile` snippet:

```
##########################################
# Dockerfile to build an apache2 image
##########################################
```

```
# Base image is Ubuntu
FROM ubuntu:14.04
```

Set authors' detail using MAINTAINER instruction

```
# Author: Dr. Peter
MAINTAINER Dr. Peter <peterindia@gmail.com>
```

Using one `RUN` instruction, we will synchronize the `apt` repository source list, install the `apache2` package, and then clean the retrieved files:

```
# Install apache2 package
RUN apt-get update && \
     apt-get install -y apache2 && \
     apt-get clean
```

Set the Apache log directory path using the `ENV` instruction:

```
# Set the log directory PATH
ENV APACHE_LOG_DIR /var/log/apache2
```

Now, the final instruction is to launch the `apache2` server using the `ENTRYPOINT` instruction:

```
# Launch apache2 server in the foreground
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

In the preceding line, you might be surprised to see the `FOREGROUND` argument. This is one of the key differences between the traditional and the container paradigm. In the traditional paradigm, the server applications are usually launched in the background either as a service or a daemon because the host system is a general-purpose system. However, in the container paradigm, it is imperative to launch an application in the foreground because the images are crafted for a sole purpose.

Having prescribed the image building instruction in the `Dockerfile`, now let's move to the next logical step of building the image using the `docker build` subcommand by naming the image as `apache2`, as shown here:

```
$ sudo docker build -t apache2 .
```

Let's now do a quick verification of the images using the `docker images` subcommand:

```
$ sudo docker images
```

As we have seen in the previous chapters, the `docker images` command displays the details of all the images in the Docker host. However, in order to illustrate precisely the images created using the `docker build` subcommand, we highlight the details of `apache2:latest` (the target image) and `ubuntu:14.04` (the base image) from the complete image list, as shown in the following output snippet:

```
apache2             latest              d5526cd1a645        About a
minute ago   232.6 MB
ubuntu              14.04               5506de2b643b        5 days
ago           197.8 MB
```

Having built the HTTP server image, now let's move on to the next session to learn how to run the HTTP service.

Running the HTTP server Image as a Service

In this section, we are going to launch a container using the Apache HTTP server image, we crafted in the previous section. Here, we launch the container in the detached mode (similar to a Unix daemon process) using the `-d` option of the `docker run` subcommand:

```
$ sudo docker run -d apache2
9d4d3566e55c0b8829086e9be2040751017989a47b5411c9c4f170ab865afcef
```

Having launched the container, let's run the `docker logs` subcommand to see whether our Docker container generates any output on its `STDIN` (standard input) or `STDERR` (standard error):

```
$ sudo docker logs \
9d4d3566e55c0b8829086e9be2040751017989a47b5411c9c4f170ab865afcef
```

As we have not fully configured the Apache HTTP server; you will find the following warning, as the output of the `docker logs` subcommand:

```
AH00558: apache2: Could not reliably determine the server's fully
qualified domain name, using 172.17.0.13. Set the 'ServerName'
directive globally to suppress this message
```

From the preceding warning message, it is quite evident that the IP address assigned to this container is `172.17.0.13`.

Connecting to the HTTP service

In the preceding section, from the warning message, we found out that the IP address of the container is `172.17.0.13`. On a fully configured HTTP server container, no such warning is available, so let's still run the `docker inspect` subcommand to retrieve the IP address using the container ID:

```
$ sudo docker inspect \
--format='{{.NetworkSettings.IPAddress}}' \
9d4d3566e55c0b8829086e9be2040751017989a47b5411c9c4f170ab865afcef
172.17.0.13
```

Having found the IP address of the container as `172.17.0.13`, let's quickly run a web request on this IP address from the shell prompt of the Docker host, using the `wget` command. Here, we choose to run the `wget` command with `-qO-` in order to run in the quiet mode and also display the retrieved HTML file on the screen:

```
$ wget -qO - 172.17.0.13
```

Here, we are showcasing just the first five lines of the retrieved HTML file:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <!--
    Modified from the Debian original for Ubuntu
    Last updated: 2014-03-19
```

Awesome, isn't it? We got our first service running in a container, and we are able to reach out to our service from our Docker host.

Furthermore, on a plain vanilla Docker installation, the service offered by one container is accessible by any other container within the Docker host. You can go ahead, launch a new Ubuntu container in the interactive mode, install the `wget` package using `apt-get`, and run the same `wget -qO - 172.17.0.13` command, as we did in the Docker host. Of course, you will see the same output.

Exposing container services

So far, we have successfully launched an HTTP service and accessed the service from the Docker host as well as another container within the same Docker host. Furthermore, as demonstrated in the *Build images from containers* section of *Chapter 4*, *Handling Docker Containers*, the container is able to successfully install the `wget` package by making a connection to the publicly available apt repository over the Internet. Nonetheless, the outside world cannot access the service offered by a container by default. At the outset, this might seem like a limitation in the Docker technology. However, the fact of the matter is, the containers are isolated from the outside world by design.

Docker achieves network isolation for the containers by the IP address assignment criteria, as enumerated:

1. Assign a private IP address to the container, which is not reachable from an external network.
2. Assign an IP address to the container outside the host's IP network.

Consequently, the Docker container is not reachable, even from the systems that are connected to the same IP network as the Docker host. This assignment scheme also provides protection from an IP address conflict that might otherwise arise.

Now, you might wonder how to make the services run inside a container that is accessible to the outside world, in other words, exposing container services. Well, Docker bridges this connectivity gap in a classy manner by leveraging the Linux `iptables` functionality under the hood.

At the frontend, Docker provides two different building blocks to bridge this connectivity gap for its users. One of the building blocks is to bind the container port using the `-p` (publish a container's port to the host interface) option of the `docker run` subcommand. Another alternative is to use the combination of the `EXPOSE` Dockerfile instruction and the `-P` (publish all exposed ports to the host interfaces) option of the `docker run` subcommand.

Publishing container ports – the -p option

Docker enables you to publish a service offered inside a container by binding the container's port to the host interface. The `-p` option of the `docker run` subcommand enables you to bind a container port to a user-specified or auto-generated port of the Docker host. Thus, any communication destined for the IP address and the port of the Docker host would be forwarded to the port of the container. The `-p` option, actually, supports the following four formats of arguments:

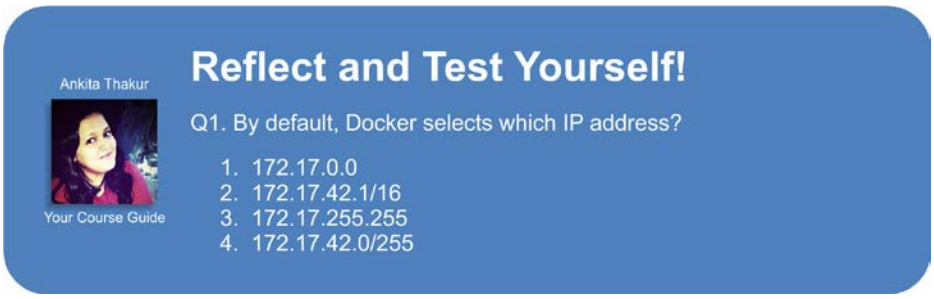
- `<hostPort>:<containerPort>`
- `<containerPort>`
- `<ip>:<hostPort>:<containerPort>`
- `<ip>::<containerPort>`

Here, `<ip>` is the IP address of the Docker host, `<hostPort>` is the Docker host port number, and `<containerPort>` is the port number of the container. Here, in this section, we present you with the `-p <hostPort>:<containerPort>` format and introduce other formats in the succeeding sections.

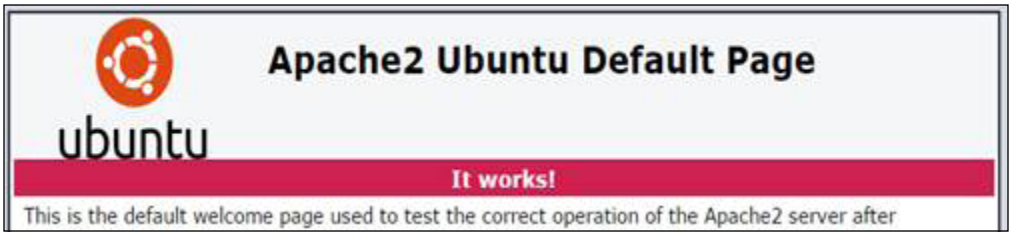
In order to understand the port binding process better, let's reuse the `apache2` HTTP server image that we crafted previously, and spin up a container using a `-p` option of the `docker run` subcommand. The port `80` is the published port of the HTTP service, and as the default behavior, our `apache2` HTTP server is also available on port `80`. Here, in order to demonstrate this capability, we are going to bind port `80` of the container to port `80` of the Docker host, using the `-p <hostPort>:<containerPort>` option of the `docker run` subcommand, as shown in the following command:

```
$ sudo docker run -d -p 80:80 apache2
baddba8afa98725ec85ad953557cd0614b4d0254f45436f9cb440f3f9eeae134
```

Now that we have successfully launched the container, we can connect to our HTTP server using any web browser from any external system (provided it has network connectivity) to reach our Docker host. So far, we have not added any web pages to our `apache2` HTTP server image.



Hence, when we connect from a web browser, we will get the following screen, which is nothing but the default page that comes along with the `Ubuntu Apache2` package:



Network Address Translation for containers

In the previous section, we saw how a `-p 80:80` option did the magic, didn't it? Well, in reality, under the hood, the Docker engine achieves this seamless connectivity by automatically configuring the **Network Address Translation** (**NAT**) rule in the Linux `iptables` configuration files.

To illustrate the automatic configuration of the NAT rule in Linux `iptables`, let's query the Docker hosts `iptables` for its NAT entries, as follows:

```
$ sudo iptables -t nat -L -n
```

The ensuing text is an excerpt from the `iptables` NAT entry, which is automatically added by the Docker engine:

```
Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tcp
dpt:80 to:172.17.0.14:80
```

From the preceding excerpt, it is quite evident that the Docker engine has effectively added a `DNAT` rule. The following are the details of the `DNAT` rule:

- The `tcp` keyword signifies that this `DNAT` rule applies only to the TCP transport protocol.
- The first `0.0.0.0/0` address is a meta IP address of the source address. This address indicates that the connection can originate from any IP address.

- The second `0.0.0.0/0` address is a meta IP address of the destination address on the Docker host. This address indicates that the connection could be made to any valid IP address in the Docker host.
- Finally, `dpt:80 to:172.17.0.14:80` is the forwarding instruction used to forward any TCP activity on port `80` of the Docker host to the IP address `172.17.0.17`, the IP address of our container and port `80`.

  Therefore, any TCP packet that the Docker host receives on port `80` will be forwarded to port `80` of the container.

Retrieving the container port

The Docker engine provides at least three different options to retrieve the containers port binding details. Here, let's first explore the options, and then, move on to dissect the retrieved information. The options are as follows:

- The `docker ps` subcommand always displays the port binding details of a container, as shown here:

```
$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAMES
baddba8afa98        apache2:latest      "/usr/sbin/apache2ct
26 seconds ago      Up 25 seconds       0.0.0.0:80->80/tcp
furious_carson
```

- The `docker inspect` subcommand is another alternative; however, you have to skim through quite a lot of details. Run the following command:

```
$ sudo docker inspect baddba8afa98
```

  The `docker inspect` subcommand displays the port binding related information in three JSON objects, as shown here:

  - The `ExposedPorts` object enumerates all ports that are exposed through the `EXPOSE` instruction in `Dockerfile`, as well as the container ports that are mapped using the `-p` option in the `docker run` subcommand. Since we didn't add the `EXPOSE` instruction in our `Dockerfile`, what we have is just the container port that was mapped using `-p80:80` as an argument to the `docker run` subcommand:

```
"ExposedPorts": {
            "80/tcp": {}
        },
```

- The `PortBindings` object is part of the `HostConfig` object, and this object lists out all the port binding done through the `-p` option in the `docker run` subcommand. This object will never list the ports exposed through the `EXPOSE` instruction in the `Dockerfile`:

```
"PortBindings": {
            "80/tcp": [
                {
                    "HostIp": "",
                    "HostPort": "80"
                }
            ]
        },
```

- The `Ports` object of the `NetworkSettings` object has the same level of detail, as the preceding `PortBindings` object. However, this object encompasses all ports that are exposed through the `EXPOSE` instruction in `Dockerfile`, as well as the container ports that are mapped using the `-p` option in the `docker run` subcommand:

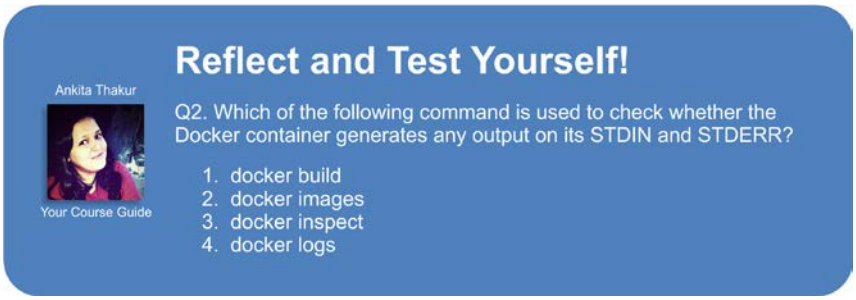
```
"NetworkSettings": {
        "Bridge": "docker0",
        "Gateway": "172.17.42.1",
        "IPAddress": "172.17.0.14",
        "IPPrefixLen": 16,
        "PortMapping": null,
        "Ports": {
            "80/tcp": [
                {
                    "HostIp": "0.0.0.0",
                    "HostPort": "80"
                }
            ]
        }
    },
```

Of course, the specific port field can be filtered using the `--format` option of the `docker inspect` subcommand.

- The `docker port` subcommand enables you to retrieve the port binding on the Docker host by specifying the container's port number:

```
$ sudo docker port baddba8afa98 80
0.0.0.0:80
```

Evidently, in all the preceding output excerpts, the information that stands out is the IP address `0.0.0.0` and the port number `80`. The IP address `0.0.0.0` is a meta address, which represents all the IP addresses configured on the Docker host. In effect, the containers port `80` is bound to all the valid IP addresses on the Docker host. Therefore, the HTTP service is accessible through any of the valid IP addresses configured on the Docker host.



Binding a container to a specific IP address

Until now, with the method that we have learnt, the containers always get bound to all the IP addresses configured in the Docker host. However, you may want to offer different services on different IP addresses. In other words, a specific IP address and port would be configured to offer a particular service. We can achieve this in Docker using the `-p <ip>:<hostPort>:<containerPort>` option of the `docker run` subcommand, as shown in the following example:

```
$ sudo docker run -d -p 198.51.100.73:80:80 apache2
92f107537bebd48e8917ea4f4788bf3f57064c8c996fc23ea0fd8ea49b4f3335
```

Here, the IP address must be a valid IP address on the Docker host. If the specified IP address is not a valid IP address on the Docker host, the container launch will fail with an error message, as follows:

```
2014/11/09 10:22:10 Error response from daemon: Cannot start
container
99db8d30b284c0a0826d68044c42c370875d2c3cad0b87001b858ba78e9de53b:
Error starting userland proxy: listen tcp 198.51.100.73:80: bind:
cannot assign requested address
```

Now, let's quickly review the port mapping as well as the NAT entry for the preceding example.

The following text is an excerpt from the output of the `docker ps` subcommand that shows the details of this container:

```
92f107537beb        apache2:latest      "/usr/sbin/apache2ct   About
a minute ago   Up About a minute   198.51.100.73:80->80/tcp
boring_ptolemy
```

The following text is an excerpt from the output of the `iptables -n nat -L -n` command that shows the `DNAT` entry created for this container:

```
DNAT    tcp -- 0.0.0.0/0      198.51.100.73     tcp dpt:80
to:172.17.0.15:80
```

After reviewing both the output of the `docker run` subcommand and the `DNAT` entry of `iptables`, you will realize how elegantly the Docker engine has configured the service offered by the container on the IP address `198.51.100.73` and port `80` of the Docker host.

Auto-generating the Docker host port

The Docker containers are innately lightweight and due to their lightweight nature, you can run multiple containers with the same, or different services on a single Docker host. Particularly, auto scaling of the same service across several containers based on demand, is the need of IT infrastructure today. Here, in this section, you will be informed about the challenge in spinning up multiple containers with the same service, and also Docker's way of addressing this challenge.

Earlier in this chapter, we launched a container using `apache2 http server` by binding it to port `80` of the Docker host. Now, if we attempt to launch one more container with the same port `80` binding, the container would fail to start with an error message, as you can see in the following example:

```
$ sudo docker run -d -p 80:80 apache2

6f01f485ab3ce81d45dc6369316659aed17eb341e9ad0229f66060a8ba4a2d0e

2014/11/03 23:28:07 Error response from daemon: Cannot start
container
6f01f485ab3ce81d45dc6369316659aed17eb341e9ad0229f66060a8ba4a2d0e:
Bind for 0.0.0.0:80 failed: port is already allocated
```

Obviously, in the preceding example, the container failed to start because the previous container is already mapped to `0.0.0.0` (all the IP addresses of the Docker host) and port `80`. In the TCP/IP communication model, the combination of the IP address, port, and the Transport Protocols (TCP, UDP, and so on) has to be unique.

We could have overcome this issue by manually choosing the Docker host port number (for instance, `-p 81:80` or `-p 8081:80`). Though this is an excellent solution, it does not perform well to auto-scaling scenarios. Instead, if we give the control to Docker, it would auto-generate the port number on the Docker host. This port number generation is achieved by underspecifying the Docker host port number, using the `-p <containerPort>` option of the `docker run` subcommand, as shown in the following example:

```
$ sudo docker run -d -p 80 apache2
ea3e0d1b18cff40ffcddd2bf077647dc94bceffad967b86c1a343bd33187d7a8
```

Having successfully started the new container with the auto-generated port, let's review the port mapping as well as the NAT entry for the preceding example:

- The following text is an excerpt from the output of the `docker ps` subcommand that shows the details of this container:

  ```
  ea3e0d1b18cf        apache2:latest      "/usr/sbin/apache2ct
  5 minutes ago       Up 5 minutes        0.0.0.0:49158->80/tcp
  nostalgic_morse
  ```

- The following text is an excerpt from the output of the `iptables -n nat -L -n` command that shows the `DNAT` entry created for this container:

  ```
  DNAT    tcp -- 0.0.0.0/0      0.0.0.0/0      tcp dpt:49158
  to:172.17.0.18:80
  ```

After reviewing both the output of the `docker run` subcommand and the `DNAT` entry of `iptables`, what stands out is the port number `49158`. The port number `49158` is niftily auto-generated by the Docker engine on the Docker host, with the help of the underlying operating system. Besides, the meta IP address `0.0.0.0` implies that the service offered by the container is accessible from outside, through any of the valid IP addresses configured on the Docker host.

You may have a use case where you want to auto-generate the port number. However, if you still want to restrict the service to a particular IP address of the Docker host, you can use the `-p <IP>::<containerPort>` option of the `docker run` subcommand, as shown in the following example:

```
$ sudo docker run -d -p 198.51.100.73::80 apache2
6b5de258b3b82da0290f29946436d7ae307c8b72f22239956e453356532ec2a7
```

In the preceding two scenarios, the Docker engine auto-generated the port number on the Docker host and exposed it to the outside world. The general norm for network communication is to expose any service through a predefined port number so that anybody can know the IP address, and the port number can easily access the offered service. Whereas, here, the port numbers are auto-generated and as a result, the outside world cannot directly reach the offered service. So, the primary purpose of this method of container creation is to achieve auto-scaling, and the container created in this fashion would be interfaced with a proxy or load balance service on a predefined port.

Port binding using EXPOSE and the -P option

So far, we have discussed the four distinct methods to publish a service running inside a container to the outside world. In all these four methods, the port binding decision is taken during the container launch time, and the image has no information about the ports on which the service is being offered. It has worked well so far because the image is being built by us, and we are pretty much aware of the port in which the service is being offered. However, in the case of third-party images, the port usage inside a container has to be published unambiguously. Besides, if we build images for third-party consumption or even for our own use, it is a good practice to explicitly state the ports in which the container offers its service. Perhaps, the image builders could ship a readme document along with the image. However, it is even better to embed the port details in the image itself so that you can easily find the port details from the image both manually as well as through automated scripts.

The Docker technology allows us to embed the port information using the `EXPOSE` instruction in the `Dockerfile`, which we introduced in *Chapter 3*, *Building Images*. Here, let's edit the `Dockerfile` we used to build the `apache2` HTTP server image earlier in this chapter, and add an `EXPOSE` instruction, as shown in the following code. The default port for the HTTP service is port `80`, hence port `80` is exposed:

```
##########################################
# Dockerfile to build an apache2 image
##########################################
# Base image is Ubuntu
FROM ubuntu:14.04
# Author: Dr. Peter
MAINTAINER Dr. Peter <peterindia@gmail.com>
# Install apache2 package
RUN apt-get update && \
     apt-get install -y apache2 && \
     apt-get clean
# Set the log directory PATH
ENV APACHE_LOG_DIR /var/log/apache2
```

```
# Expose port 80
EXPOSE 80
# Launch apache2 server in the foreground
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Now that we have added the `EXPOSE` instruction to our `Dockerfile`, let's move to the next step of building the image using the `docker build` command. Here, let's reuse the image name `apache2`, as shown here:

```
$ sudo docker build -t apache2 .
```

Having successfully built the image, let's inspect the image to verify the effects of the `EXPOSE` instruction to the image. As we learnt earlier, we can resort to the `docker inspect` subcommand, as shown here:

```
$ sudo docker inspect apache2
```

On close review of the output generated by the preceding command, you will realize that Docker stores the exposed port information in the `ExposedPorts` field of the `Config` object. The following is an excerpt to show how the exposed port information is being displayed:

```
"ExposedPorts": {
            "80/tcp": {}
        },
```

Alternatively, you can apply the format option to the `docker inspect` subcommand in order to narrow down the output to very specific information. In this case, the `ExposedPorts` field of the `Config` object is shown in the following example:

```
$ sudo docker inspect --format='{{.Config.ExposedPorts}}' \
    apache2
map[80/tcp:map[]]
```

To resume our discussion on the `EXPOSE` instruction, we can now spin up containers using an `apache2` image, that we just crafted. Yet, the `EXPOSE` instruction by itself cannot create port binding on the Docker host. In order to create port binding for the port declared using the `EXPOSE` instruction, the Docker engine provides a `-P` option in the `docker run` subcommand.

In the following example, a container is launched from the `apache2` image, which was rebuilt earlier. Here, the `-d` option is used to launch the container in the detached mode, and the `-P` option is used to create the port binding in the Docker host for all the ports declared, using the `EXPOSE` instruction in the `Dockerfile`:

```
$ sudo docker run -d -P apache2
fdb1c8d68226c384ab4f84882714fec206a73fd8c12ab57981fbd874e3fa9074
```

Now that we have started the new container with the image that was created using the `EXPOSE` instruction, like the previous containers, let's review the port mapping as well as the NAT entry for the preceding example:

- The following text is an excerpt from the output of the `docker ps` subcommand that shows the details of this container:

```
ea3e0d1b18cf        apache2:latest      "/usr/sbin/apache2ct
5 minutes ago       Up 5 minutes        0.0.0.0:49159->80/tcp
nostalgic_morse
```

- The following text is an excerpt from the output of the `iptables -t nat -L -n` command that shows the `DNAT` entry created for this container:

```
DNAT    tcp -- 0.0.0.0/0      0.0.0.0/0      tcp dpt:49159
to:172.17.0.19:80
```

The `-P` option of the `docker run` subcommand does not take any additional arguments, such as an IP address or a port number; consequently, fine tuning of the port binding is not possible, such as the `-p` option of the `docker run` subcommand. You can always resort to the `-p` option of the `docker run` subcommand if fine tuning of the port binding is critical to you.

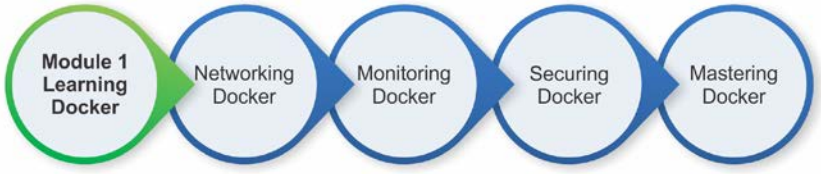
Summary of Module 1 Chapter 7

Ankita Thakur

Your Course Guide

Containers do not deliver anything in an isolated or solo way substantially. They need to be systematically built and provided with a network interface along with a port number. This leads to a standardized exposition of containers to the outside world, facilitating other hosts or containers to find, bind, and leverage their unique capabilities on any network. Thus, network-accessibility is paramount for containers to get noticed and utilized in innumerable ways. This chapter is dedicated to showcasing how containers are being designed and deployed as a service, and how the aspect of container networking comes in handy by precisely and profusely empowering the peculiar world of container services as the days unfold. In the forthcoming chapters, we will deal and dwell at length on the various capabilities of Docker containers in software-intensive IT environments.

Your Progress through the Course So Far



8
Sharing Data with Containers

Do one thing at a time and do it well, is one of the successful mantras of the information technology (IT) sector for quite a long time now. This widely used tenet fits nicely with the building and exposing of Docker containers too and is being prescribed as one of the best practices to avail the originally envisaged benefits of the Docker-inspired containerization paradigm. That is, inscribe a single application along with its direct dependencies and libraries inside a Docker container in order to ensure the container's independence, self-sufficiency, and maneuverability. Let's see why containers are that important:

- **The temporal nature of containers**: The container typically lives as long as the application lives and vice versa. However, this has some negative implications for the application data. It is natural that applications go through a variety of changes in order to accommodate both businesses, as well as technical changes, even in their production environments. There are other causes, such as application malfunction, version changes, application maintenance, and so on, for applications to be updated and upgraded consistently. In the case of a general-purpose computing model, even when an application dies for any reason, the persistent data associated with this application would be preserved in the filesystem. However, in the case of the container paradigm, the application upgrades are usually performed by crafting a new container with the newer version of the application, by discarding the old one. Similarly, when an application malfunctions, a new container needs to be launched and the old one has to be discarded. To sum it up, containers are temporal in nature.

- **The need for business continuity**: In the container landscape, the complete execution environment, including its data files are usually bundled and encapsulated inside the container. For any reason, when a container gets discarded, the application data files also perish along with the container. However, in order to provide a seamless service, these application data files must be preserved outside the container and passed on to the container that will be continuing with the service. Some application data files, such as the log files, need to be accessed outside the container for various posterior-analyses. The Docker technology addresses this file persistence issue very innovatively through a new building block called data volume.

In this chapter, we will cover the following topics:

- Data volume
- Sharing host data
- Sharing data between containers
- The avoidable common pitfalls
- Data volume containers
- Data volume backups

The data volume

The data volume is the fundamental building block of data sharing in the Docker environment. Before getting into the details of data sharing, it is imperative to gain a good understanding of the Data Volume concept. Until now, all the files that we created in an image or a container are part and parcel of the Union filesystem. However, the data volume is part of the Docker host filesystem, and it simply gets mounted inside the container.

A data volume can be inscribed in a Docker image using the `VOLUME` instruction of the `Dockerfile`. Also, it can be prescribed during the launch of a container using the `-v` option of the `docker run` subcommand. Here, in the following example, the implication of the `VOLUME` instruction in the `Dockerfile` is illustrated in detail, in the following steps:

1. Create a very simple `Dockerfile` with the instruction of the base image (`ubuntu:14.04`) and the data volume (`/MountPointDemo`):

```
FROM ubuntu:14.04
VOLUME /MountPointDemo
```

2. Build the image with the name `mount-point-demo` using the `docker build` subcommand:

```
$ sudo docker build -t mount-point-demo .
```

3. Having built the image, let's quickly inspect the image for our data volume using the `docker inspect` subcommand:

```
$ sudo docker inspect mount-point-demo
[{
    "Architecture": "amd64",
... TRUNCATED OUTPUT ...
        "Volumes": {
            "/MountPointDemo": {}
        },
... TRUNCATED OUTPUT ...
```

   Evidently, in the preceding output, the data volume is inscribed in the image itself.

4. Now, let's launch an interactive container using the `docker run` subcommand from the earlier crafted image, as shown in the following command:

```
$ sudo docker run --rm -it mount-point-demo
```

   From the container's prompt, let's check the presence of the data volume using the `ls -ld` command:

```
root@8d22f73b5b46:/# ls -ld /MountPointDemo
drwxr-xr-x 2 root root 4096 Nov 18 19:22 /MountPointDemo
```

   As mentioned earlier, the data volume is part of the Docker host filesystem and it gets mounted, as shown in the following command:

```
root@8d22f73b5b46:/# mount
... TRUNCATED OUTPUT ...
/dev/disk/by-uuid/721cedbd-57b1-4bbd-9488-ec3930862cf5 on
/MountPointDemo type ext3
(rw,noatime,nobarrier,errors=remount-ro,data=ordered)
... TRUNCATED OUTPUT ...
```

5. In this section, we inspected the image to find out about the data volume declaration in the image. Now that we have launched the container, let's inspect the container's data volume using the `docker inspect` subcommand with the container ID as its argument in a different terminal. We created a few containers previously and for this purpose, let's take the container ID `8d22f73b5b46` directly from the container's prompt:

```
$ sudo docker inspect 8d22f73b5b46
... TRUNCATED OUTPUT ...
    "Volumes": {
        "/MountPointDemo":
"/var/lib/docker/vfs/dir/737e0355c5d81c96a99d41d1b9f540c2a2120
00661633ceea46f2c298a45f128"
    },
    "VolumesRW": {
        "/MountPointDemo": true
    }
}
```

Apparently, here, the data volume is mapped to a directory in the Docker host, and the directory is mounted in read-write mode. This directory is created by the Docker engine automatically during the container launch time.

So far, we have seen the implication of the `VOLUME` instruction in the `Dockerfile`, and how Docker manages the data volume. Like the `VOLUME` instruction of the `Dockerfile`, we can use the `-v <container mount point path>` option of the `docker run` subcommand, as shown in the following command:

```
$ sudo docker run -v /MountPointDemo -it ubuntu:14.04
```

Having launched the container, we encourage you to try the `ls -ld / MountPointDemo` and `mount` commands in the newly launched container, and then also, inspect the container, as shown in the preceding step, step 5.

In both the scenarios described here, the Docker engine automatically creates the directory under `/var/lib/docker/vfs/` and mounts it to the container. When a container is removed using the `docker rm` subcommand, the Docker engine does not remove the directory that was automatically created during the container launch time. This behavior is innately designed to preserve the state of the container's application that was stored in the directory. If you want to remove the directory that was automatically created by the Docker engine, you can do so while removing the container by providing a `-v` option to the `docker rm` subcommand, on an already stopped container:

```
$ sudo docker rm -v 8d22f73b5b46
```

If the container is still running, then you can remove the container as well as the auto-generated directory by adding a `-f` option to the previous command:

```
$ sudo docker rm -fv 8d22f73b5b46
```

We have taken you through the techniques and tips to auto-generate a directory in the Docker host and mount it to the data volume in the container. However, with the `-v` option of the `docker run` subcommand, a user-defined directory can be mounted to the data volume. In such cases, the Docker engine would not auto-generate any directory.

The system-generated directory has a caveat of directory leak. In other words, if you forget to delete the system-generated directories, you may face some unwanted issues. For further information, you can read the *Avoiding common pitfalls* section in this chapter.

Sharing host data

Earlier, we described the steps towards creating a data volume in a Docker image using the `VOLUME` instruction in the `Dockerfile`. However, Docker does not provide any mechanism to mount the host directory or file during the build time in order to ensure the Docker images are portable. The only provision Docker provides is to mount the host directory or file to a container's data volume during the container's launch time. Docker exposes the host directory or file mounting facility through the `-v` option of the `docker run` subcommand. The `-v` option has three different formats enumerated as follows:

1. `-v <container mount path>`
2. `-v <host path>/<container mount path>`
3. `-v <host path>/<container mount path>:<read write mode>`

The `<host path>` is an absolute path in the Docker host, `<container mount path>` is an absolute path in the container filesystem, and `<read write mode>` can be either read-only (`ro`) or read-write (`rw`) mode. The first `-v <container mount path>` format has already been explained in the *Data Volume* section in this chapter, as a method to create a mount point during the container launch time. The second and third options enable us to mount a file or directory from the Docker host to the container mount point.

We would like to dig deeper to gain a better understanding of the host's data sharing through a couple of examples. In the first example, we will demonstrate how to share a directory between the Docker host and the container, and in the second example, we will demonstrate file sharing.

Here, in the first example, we mount a directory from the Docker host to a container, perform a few basic file operations on the container, and verify these operations from the Docker host, as detailed in the following steps:

1. First, let's launch an interactive container with the `-v` option of the `docker run` subcommand to mount `/tmp/hostdir` of the Docker host directory to `/MountPoint` of the container:

```
$ sudo docker run -v /tmp/hostdir:/MountPoint \
                  -it ubuntu:14.04
```

If `/tmp/hostdir` is not found on the Docker host, the Docker engine will create the directory itself. However, the problem is that the system-generated directory cannot be deleted using the `-v` option of the `docker rm` subcommand.

2. Having successfully launched the container, we can check the presence of `/MountPoint` using the `ls` command:

```
root@4a018d99c133:/# ls -ld /MountPoint
drwxr-xr-x 2 root root 4096 Nov 23 18:28 /MountPoint
```

3. Now, we can proceed to checking the mount details using the `mount` command:

```
root@4a018d99c133:/# mount
... TRUNCATED OUTPUT ...
/dev/disk/by-uuid/721cedbd-57b1-4bbd-9488-ec3930862cf5 on
/MountPoint type ext3 (rw,noatime,nobarrier,errors=remount-
ro,data=ordered)
... TRUNCATED OUTPUT ...
```

4. Here, we are going to validate `/MountPoint`, change to the `/MountPoint` directory using the `cd` command, create a few files using the `touch` command, and list the files using the `ls` command, as shown in the following script:

```
root@4a018d99c133:/# cd /MountPoint/
root@4a018d99c133:/MountPoint# touch {a,b,c}
root@4a018d99c133:/MountPoint# ls -l
total 0
-rw-r--r-- 1 root root 0 Nov 23 18:39 a
-rw-r--r-- 1 root root 0 Nov 23 18:39 b
-rw-r--r-- 1 root root 0 Nov 23 18:39 c
```

5. It might be worth the effort to verify the files in the `/tmp/hostdir` Docker host directory using the `ls` command on a new terminal, as our container is running in an interactive mode on the existing terminal:

```
$ sudo  ls -l /tmp/hostdir/
total 0
-rw-r--r-- 1 root root 0 Nov 23 12:39 a
-rw-r--r-- 1 root root 0 Nov 23 12:39 b
-rw-r--r-- 1 root root 0 Nov 23 12:39 c
```

   Here, we can see the same set of files, as we can see in step 4. However, you might have noticed the difference in the time stamp of the files. This time difference is due to the time-zone difference between the Docker host and the container.

6. Finally, let's run the `docker inspect` subcommand with the container ID `4a018d99c133` as an argument to see whether the directory mapping is set up between the Docker host and the container mount point, as shown in the following command:

```
$ sudo docker inspect \
                    --format={{.Volumes}} 4a018d99c133
map[/MountPoint:/tmp/hostdir]
```

   Apparently, in the preceding output of the `docker inspect` subcommand, the `/tmp/hostdir` directory of the Docker host is mounted on the `/MountPoint` mount point of the container.

For the second example, we can mount a file from the Docker host to a container, update the file from the container, and verify these operations from the Docker host, as detailed in the following steps:

1. In order to mount a file from the Docker host to the container, the file must preexist in the Docker host. Otherwise, the Docker engine will create a new directory with the specified name, and mount it as a directory. We can start by creating a file on the Docker host using the `touch` command:

```
$ touch /tmp/hostfile.txt
```

2. Launch an interactive container with the `-v` option of the `docker run` subcommand to mount the `/tmp/hostfile.txt` Docker host file to the container as `/tmp/mntfile.txt`:

```
$ sudo docker run -v /tmp/hostfile.txt:/mountedfile.txt \
                  -it ubuntu:14.04
```

3. Having successfully launched the container, now let's check the presence of `/mountedfile.txt` using the `ls` command:

```
root@d23a15527eeb:/# ls -l /mountedfile.txt
-rw-rw-r-- 1 1000 1000 0 Nov 23 19:33 /mountedfile.txt
```

4. Then, proceed to check the mount details using the `mount` command:

```
root@d23a15527eeb:/# mount
... TRUNCATED OUTPUT ...
/dev/disk/by-uuid/721cedbd-57b1-4bbd-9488-ec3930862cf5 on
/mountedfile.txt type ext3
(rw,noatime,nobarrier,errors=remount-ro,data=ordered)
... TRUNCATED OUTPUT ...
```

5. Then, update some text to `/mountedfile.txt` using the `echo` command:

```
root@d23a15527eeb:/# echo "Writing from Container" \
           > mountedfile.txt
```

6. Meanwhile, switch to a different terminal in the Docker host, and print the `/tmp/hostfile.txt` Docker host file using the `cat` command:

```
$ cat /tmp/hostfile.txt
Writing from Container
```

7. Finally, run the `docker inspect` subcommand with the container ID `d23a15527eeb` as it's argument to see the file mapping between the Docker host and the container mount point:

```
$ sudo docker inspect \
                 --format={{.Volumes}} d23a15527eeb
map[/mountedfile.txt:/tmp/hostfile.txt]
```

   From the preceding output, it is evident that the `/tmp/hostfile.txt` file from the Docker host is mounted as `/mountedfile.txt` inside the container.

In the case of file sharing between the Docker host and container, the file must exist before launching the container. However, in the case of directory sharing, if the directory does not exist in the Docker host, then the Docker engine would create a new directory in the Docker host, as explained earlier.

The practicality of host data sharing

In the previous chapter, we launched an `HTTP` service in a Docker container. However, if you remember correctly, the log file for the `HTTP` service is still inside the container, and it cannot be accessed directly from the Docker host. Here, in this section, we elucidate the procedure of accessing the log files from the Docker host in a step-by-step manner:

1. Let's begin with launching an Apache2 HTTP service container by mounting the `/var/log/myhttpd` directory of the Docker host to the `/var/log/apache2` directory of the container, using the `-v` option of the `docker run` subcommand. In this example, we are leveraging the `apache2` image, which we had built in the previous chapter, by invoking the following command:

   ```
   $ sudo docker run -d -p 80:80 \
          -v /var/log/myhttpd:/var/log/apache2 apache2
   9c2f0c0b126f21887efaa35a1432ba7092b69e0c6d
   523ffd50684e27eeab37ac
   ```

   If you recall the `Dockerfile` in *Chapter 7*, *Running Services in a Container*, the `APACHE_LOG_DIR` environment variable is set to the `/var/log/apache2` directory, using the `ENV` instruction. This would make the Apache2 HTTP service route all log messages to the `/var/log/apache2` data volume.

2. Once the container is launched, we can change the directory to `/var/log/myhttpd` on the Docker host:

   ```
   $ cd /var/log/myhttpd
   ```

3. Perhaps, a quick check of the files present in the `/var/log/myhttpd` directory is appropriate here:

   ```
   $ ls -1
   access.log
   error.log
   other_vhosts_access.log
   ```

   Here, the `access.log` contains all the access requests handled by the Apache2 HTTP server. The `error.log` is a very important log file, where our HTTP server records the errors it encounters while processing any HTTP requests. The `other_vhosts_access.log` file is the virtual host log, which will always be empty in our case.

4. We can display the content of all the log files in the `/var/log/myhttpd` directory using the `tail` command with the `-f` option:

```
$ tail -f *.log
==> access.log <==

==> error.log <==
AH00558: apache2: Could not reliably determine the server's
fully qualified domain name, using 172.17.0.17. Set the
'ServerName' directive globally to suppress this message
[Thu Nov 20 17:45:35.619648 2014] [mpm_event:notice] [pid
16:tid 140572055459712] AH00489: Apache/2.4.7 (Ubuntu)
configured -- resuming normal operations
[Thu Nov 20 17:45:35.619877 2014] [core:notice] [pid 16:tid
140572055459712] AH00094: Command line: '/usr/sbin/apache2 -D
FOREGROUND'
==> other_vhosts_access.log <==
```

   The `tail -f` command will run continuously and display the content of the files, as soon as they get updated. Here, both `access.log` and `other_vhosts_access.log` are empty, and there are a few error messages on the `error.log` file. Apparently, these error logs are generated by the HTTP service running inside the container. The logs are then stocked in the Docker host directory, which is mounted during the launch time of the container.

5. As we continue to run `tail -f *`, let's connect to the HTTP service from a web browser running inside the container, and observe the log files:

```
==> access.log <==
111.111.172.18 - - [20/Nov/2014:17:53:38 +0000] "GET /
HTTP/1.1" 200 3594 "-" "Mozilla/5.0 (Windows NT 6.1; WOW64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.65
Safari/537.36"
111.111.172.18 - - [20/Nov/2014:17:53:39 +0000] "GET
/icons/ubuntu-logo.png HTTP/1.1" 200 3688
"http://111.71.123.110/" "Mozilla/5.0 (Windows NT 6.1; WOW64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.65
Safari/537.36"
111.111.172.18 - - [20/Nov/2014:17:54:21 +0000] "GET
/favicon.ico HTTP/1.1" 404 504 "-" "Mozilla/5.0 (Windows NT
6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/39.0.2171.65 Safari/537.36"
```

   The HTTP service updates the `access.log` file, which we can manipulate from the host directory mounted through the `-v` option of the `docker run` subcommand.

Sharing data between containers

In the previous section, we learnt how seamlessly the Docker engine enables data sharing between the Docker host and the container. Even though it is a very effective solution, it tightly couples the container to the host filesystem. These directories might leave a nasty footprint because the user has to manually remove them once their purpose is met. So, the Docker's prescription to solve this issue is to create data-only containers as a base container, and then mount the Data Volume of that container to other containers using the `--volume-from` option of the `docker run` subcommand.

Data-only containers

The prime responsibility of a data-only container is to preserve the data. Creating a data-only container is very similar to the method illustrated in the data volume section. In addition, the containers are named explicitly for other containers to mount the data volume using the container's name. The container's data volumes are accessible from other containers even when the data-only containers are in the stopped state. The data-only containers can be created in two ways, as follows:

- During the container's launch time by configuring the data volume and container's name.
- The data volume can also be inscribed with `Dockerfile` during the image-building time, and later the container can be named during the container's launch time.

In the following example, we are launching a data-only container by configuring the container launch with the `-v` and `--name` options of the `docker run` subcommand, as shown here:

```
$ sudo docker run --name datavol \
                  -v /DataMount \
                  busybox:latest /bin/true
```

Here, the container is launched from the `busybox` image, which is widely used for its smaller footprint. Here, we choose to execute the `/bin/true` command because we don't intend to do any operations on the container. Therefore, we named the container `datavol` using the `--name` option and created a new `/DataMount` data volume using the `-v` option of the `docker run` subcommand. The `/bin/true` command exits immediately with the exit status `0`, which in turn will stop the container and continue to be in the stopped state.

Mounting data volume from other containers

The Docker engine provides a nifty interface to mount (share) the data volume from one container to another. Docker makes this interface available through the `--volumes-from` option of the `docker run` subcommand. The `--volumes-from` option takes a container name or container ID as its input and automatically mounts all the data volumes available on the specified container. Docker allows you to mount multiple containers with the data volume using the `--volumes-from` option multiple times.

Here is a practical example that demonstrates how to mount the data volume from another container and showcases the data volume mount process, step by step.

1. We begin with launching an interactive Ubuntu container by mounting the data volume from the data-only container (`datavol`), which we launched in the previous section, as explained here:

```
$ sudo docker run –it \
                  --volumes-from datavol \
                  ubuntu:latest /bin/bash
```

2. Now from the container's prompt, let's verify the data volume mounts using the `mount` command:

```
root@e09979cacec8:/# mount
. . . TRUNCATED OUTPUT . . .
/dev/disk/by-uuid/32a56fe0-7053-4901-ae7e-24afe5942e91 on
/DataMount type ext3 (rw,noatime,nobarrier,errors=remount-
ro,data=ordered)
. . . TRUNCATED OUTPUT . . .
```

   Here, we successfully mounted the data volume from the `datavol` data-only container.

3. Next, we need to inspect the data volume of this container from another terminal using the `docker inspect` subcommand:

```
$ sudo docker inspect  e09979cacec8
. . . TRUNCATED OUTPUT . . .
  "Volumes": {
    "/DataMount": "/var/lib/docker/vfs/dir/62f5a3314999e5aaf485fc6
92ae07b3cbfacb
ca9815d8071f519c1a836c0f01e"
```

```
},
  "VolumesRW": {
    "/DataMount": true
  }
}
```

Evidently, the data volume from the `datavol` data-only container is mounted as if they were mounted directly on to this container.

We can mount a data volume from another container and also showcase the mount points. We can make the mounted data volume work by sharing data between containers using the data volume, as demonstrated here:

1. Let's reuse the container that we launched in the previous example and create a `/DataMount/testfile` file in the data volume `/DataMount` by writing some text to the file, as shown here:

   ```
   root@e09979cacec8:/# echo \
              "Data Sharing between Container" > \
                 /DataMount/testfile
   ```

2. Just spin off a container to display the text that we wrote in the previous step, using the `cat` command:

   ```
   $ sudo docker run --rm \
                    --volumes-from datavol \
                    busybox:latest cat /DataMount/testfile
   ```

   The following is the typical output of the preceding command:

   ```
   Data Sharing between Container
   ```

Evidently, the preceding output `Data Sharing between Container` of our newly containerized `cat` command is the text that we have written in `/DataMount/testfile` of the `datavol` container in step 1.

Cool, isn't it? You can share data seamlessly between containers by sharing the data volumes. Here, in this example, we used data-only containers as the base container for data sharing. However, Docker allows us to share any type of data volumes and to mount data volumes one after another, as depicted here:

```
$ sudo docker run --name vol1 --volumes-from datavol \
          busybox:latest /bin/true
$ sudo docker run --name vol2 --volumes-from vol1 \
          busybox:latest /bin/true
```

Here, in the `vol1` container, we can mount the data volume from the `datavol` container. Then, in the `vol2` container, we mounted the data volume from the `vol1` container, which is originally from the `datavol` container.

The practicality of data sharing between containers

Earlier, in this chapter, we learnt the mechanism of accessing the log files of the Apache2 HTTP service from the Docker host. Although it was fairly convenient to share data by mounting the Docker host directory to a container, later we came to realise that data can be shared between containers by just using the data volumes. So here, we are bringing in a twist to the method of the Apache2 HTTP service log handling by sharing data between containers. To share log files between containers, we will spin off the following containers as enlisted in the following steps:

1. First, a data-only container that would expose the data volume to other containers.
2. Then, an Apache2 HTTP service container leveraging the data volume of the data-only container.
3. A container to view the log files generated by our Apache2 HTTP service.

NOTE: If you are running any HTTP service on the port number `80` of your Docker host machine, pick any other unused port number for the following example. If not, first stop the HTTP service, then proceed with the example in order to avoid any port conflict.

Now, we meticulously walk you through the steps to craft the respective images and launch the containers to view the log files, as illustrated here:

1. Here, we begin with crafting a `Dockerfile` with the `/var/log/apache2` data volume using the `VOLUME` instruction. The `/var/log/apache2` data volume is a direct mapping to `APACHE_LOG_DIR`, the environment variable set in the `Dockerfile` in *Chapter 7*, *Running Services in a Container*, using the `ENV` instruction:

```
########################################################
# Dockerfile to build a LOG Volume for Apache2 Service
########################################################
```

```
# Base image is BusyBox
FROM busybox:latest
# Author: Dr. Peter
MAINTAINER Dr. Peter <peterindia@gmail.com>
# Create a data volume at /var/log/apache2, which is
# same as the log directory PATH set for the apache image
VOLUME /var/log/apache2
# Execute command true
CMD ["/bin/true"]
```

Since this `Dockerfile` is crafted to launch data-only containers, the default execution command is set to `/bin/true`.

2. We will continue to build a Docker image with the name `apache2log` from the preceding `Dockerfile` using `docker build`, as presented here:

```
$ sudo docker build -t apache2log .
Sending build context to Docker daemon  2.56 kB
Sending build context to Docker daemon
Step 0 : FROM busybox:latest
... TRUNCATED OUTPUT ...
```

3. Launch a data-only container from the `apache2log` image using the `docker run` subcommand and name the resulting container `log_vol`, using the `--name` option:

```
$ sudo docker run --name log_vol apache2log
```

Acting on the preceding command, the container will create a data volume in `/var/log/apache2` and move it to a stop state.

4. Meanwhile, you can run the `docker ps` subcommand with the `-a` option to verify the container's state:

```
$ sudo docker ps -a
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS                      PORTS
NAMES
40332e5fa0ae        apache2log:latest   "/bin/true"
2 minutes ago      Exited (0) 2 minutes ago
log_vol
```

As per the output, the container exits with the exit value `0`.

5. Launch the Apache2 HTTP service using the `docker run` subcommand. Here, we are reusing the `apache2` image we crafted in *Chapter 7*, *Running Services in a Container*. In this container, we will mount the `/var/log/apache2` data volume from `log_vol`, the data-only container that we launched in step 3, using the `--volumes-from` option:

```
$ sudo docker run -d -p 80:80 \
                  --volumes-from log_vol \
                  apache2
7dfbf87e341c320a12c1baae14bff2840e64afcd082dda3094e7cb0a002
3cf42
```

   With the successful launch of the Apache2 HTTP service with the `/var/log/apache2` data volume mounted from `log_vol`, we can access the log files using transient containers.

6. Here, we are listing the files stored by the Apache2 HTTP service using a transient container. This transient container is spun off by mounting the `/var/log/apache2` data volume from `log_vol`, and the files in `/var/log/apache2` are listed using the `ls` command. Further, the `--rm` option of the `docker run` subcommand is used to remove the container once it is done executing the `ls` command:

```
$  sudo docker run --rm \
                   --volumes-from log_vol
                   busybox:latest ls -l /var/log/apache2
total 4
-rw-r--r--    1 root     root             0 Dec  5 15:27
access.log
-rw-r--r--    1 root     root           461 Dec  5 15:27
error.log
-rw-r--r--    1 root     root             0 Dec  5 15:27
other_vhosts_access.log
```

7. Finally, the error log produced by the Apache2 HTTP service is accessed using the `tail` command, as highlighted in the following command:

```
$ sudo docker run  --rm  \
                   --volumes-from log_vol \
                   ubuntu:14.04 \
                   tail /var/log/apache2/error.log
```

```
AH00558: apache2: Could not reliably determine the server's
fully qualified domain name, using 172.17.0.24. Set the
'ServerName' directive globally to suppress this message

[Fri Dec 05 17:28:12.358034 2014] [mpm_event:notice] [pid
18:tid 140689145714560] AH00489: Apache/2.4.7 (Ubuntu)
configured -- resuming normal operations

[Fri Dec 05 17:28:12.358306 2014] [core:notice] [pid 18:tid
140689145714560] AH00094: Command line: '/usr/sbin/apache2 -D
FOREGROUND'
```

Avoiding common pitfalls

Till now, we discussed how effectively data volumes can be used to share data between the Docker host and the containers as well as between containers. Data sharing using data volumes is turning out to be a very powerful and essential tool in the Docker paradigm. However, it does carry a few pitfalls that are to be carefully identified and eliminated. In this section, we make an attempt to list out a few common issues associated with data sharing and the ways and means to overcome them.

Directory leaks

Earlier in the data volume section, we learnt that the Docker engine automatically creates directories based on the `VOLUME` instruction in `Dockerfile` as well as the `-v` option of the `docker run` subcommand. We also understood that the Docker engine does not automatically delete these auto-generated directories in order to preserve the state of the application(s) run inside the container. We can force Docker to remove these directories using the `-v` option of the `docker rm` subcommand. This process of manual deletion poses two major challenges enumerated as follows:

1. **Undeleted directories:** There could be scenarios where you may intentionally or unintentionally choose not to remove the generated directory while removing the container.

2. **Third-party images:** Quite often, we leverage third-party Docker images that could have been built with the `VOLUME` instruction. Likewise, we might also have our own Docker images with `VOLUME` inscribed in it. When we launch containers using such Docker images, the Docker engine will auto-generate the prescribed directories. Since we are not aware of the data volume creation, we may not call the `docker rm` subcommand with the -v option to delete the auto-generated directory.

In the previously mentioned scenarios, once the associated container is removed, there is no direct way to identify the directories whose containers were removed. Here are a few recommendations on how to avoid this pitfall:

- Always inspect the Docker images using the `docker inspect` subcommand and check whether any data volume is inscribed in the image or not.
- Always run the `docker rm` subcommand with the `-v` option to remove any data volume (directory) created for the container. Even if the data volume is shared by multiple containers, it is still safe to run the `docker rm` subcommand with the `-v` option because the directory associated with the data volume will be deleted only when the last container sharing that data volume is removed.
- For any reason, if you choose to preserve the auto-generated directory, you must keep a clear record so that you can remove them at a later point of time.
- Implement an audit framework that will audit and find out the directories that do not have any container association.

The undesirable effect of data volume

As mentioned earlier, Docker enables us to etch data volumes in a Docker image using the `VOLUME` instruction during the build time. Nonetheless, the data volumes should never be used to store any data during the build time, otherwise it will result in an unwanted effect.

In this section, we will demonstrate the undesirable effect of using the data volume during the build time by crafting a `Dockerfile`, and then showcase the implication by building this `Dockerfile`:

The following are the details of `Dockerfile`:

1. Build the image using `Ubuntu 14.04` as the base image:

```
# Use Ubuntu as the base image
FROM ubuntu:14.04
```

2. Create a `/MountPointDemo` data volume using the `VOLUME` instruction:

   ```
   VOLUME /MountPointDemo
   ```

3. Create a file in the `/MountPointDemo` data volume using the `RUN` instruction:

   ```
   RUN date > /MountPointDemo/date.txt
   ```

4. Display the file in the `/MountPointDemo` data volume using the `RUN` instruction:

   ```
   RUN cat /MountPointDemo/date.txt
   ```

   Proceed to build an image from this `Dockerfile` using the `docker build` subcommand, as shown here:

```
$ sudo docker build -t testvol .
Sending build context to Docker daemon  2.56 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
 ---> 9bd07e480c5b
Step 1 : VOLUME /MountPointDemo
 ---> Using cache
 ---> e8b1799d4969
Step 2 : RUN date > /MountPointDemo/date.txt
 ---> Using cache
 ---> 8267e251a984
Step 3 : RUN cat /MountPointDemo/date.txt
 ---> Running in a3e40444de2e
cat: /MountPointDemo/date.txt: No such file or directory
2014/12/07 11:32:36 The command [/bin/sh -c cat
/MountPointDemo/date.txt] returned a non-zero code: 1
```

In the preceding output of the `docker build` subcommand, you would have noticed that the build fails at step 3 because it cannot find the file created in step 2. Apparently, the file that was created in step 2 vanishes when it reaches step 3. This undesirable effect is due to the approach Docker uses to build its images. An understanding of the Docker image-building process would unravel the mystery.

In the build process, for every instruction in a `Dockerfile`, the following steps are followed:

1. Create a new container by translating the `Dockerfile` instruction to an equivalent `docker run` subcommand

2. Commit the newly-created container to an image
3. Repeat step 1 and step 2, by treating the newly-created image as the base image for step 1.

When a container is committed, it saves the container's filesystem and, deliberately, does not save the data volume's filesystem. Therefore, any data stored in the data volume will be lost in this process. So never use a data volume as storage during the build process.

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.

Ankita Thakur

Your Course Guide

Your Coding Challenge

Here are some questions for you to know whether you have understood the concepts:

- What are the characteristics of Docker registry?
- What are the use cases of Docker registry?

Summary of Module 1 Chapter 8

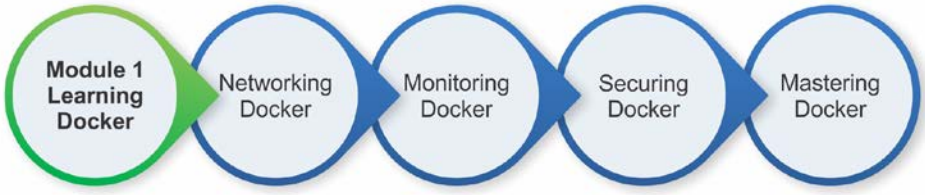
Ankita Thakur

Your Course Guide

For enterprise-scale distributed applications to be distinct in their operations and outputs, data is the most important instrument and ingredient. With IT containerization, the journey takes off in a brisk and bright fashion. IT as well as business software solutions are intelligently containerized through the smart leverage of the Docker engine. However, the original instigation is the need for a faster and flawless realization of application-aware Docker containers, and hence, the data is tightly coupled with the application within the container. However, this closeness brings in some real risks. If the application collapses, then the data is also gone. Also, multiple applications might depend on the same data and hence, data has to be shared across.

In this chapter, we discussed the capabilities of the Docker engine in facilitating the seamless data sharing between the Docker host and container as well as between containers. The data volume is being prescribed as the foundational building block for enabling data sharing among the constituents of the growing Docker ecosystem.

Your Progress through the Course So Far



9
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:

- `config`
- `env`
- `inspect`
- `kill`
- `restart`
- `rm`
- `start`
- `stop`
- `active`
- `ip`
- `ls`
- `scp`
- `ssh`
- `upgrade`
- `url`
- `TLS`

Of these commands, the first eight commands are already covered in *Chapter 2*, *Up and Running*. Let's take a look at the rest.

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
```

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
```

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.

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>
```

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.

Ankita Thakur

Your Course Guide

Reflect and Test Yourself!

Q1. Which of the following is not a Docker machine command?

1. ssh
2. remove
3. stop
4. ls

Summary of Module 1 Chapter 9

Ankita Thakur

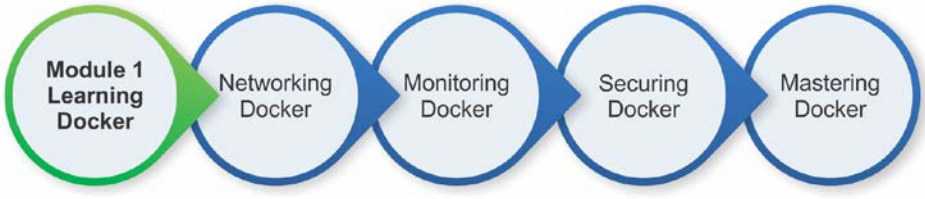
Your Course Guide

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.

Your Progress through the Course So Far



10
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

Linking containers

One of the prominent features of the Docker technology is linking containers. That is, cooperating containers can be linked together to offer complex and business-aware services. The linked containers have a kind of source-recipient relationship, wherein the source container gets linked to the recipient container, and the recipient securely receives a variety of information from the source container. However, the source container would know nothing about the recipients to which it is linked to. Another noteworthy feature of linking containers, in a secured setup, is that the linked containers can communicate using secured tunnels without exposing the ports used for the setup, to the external world.

The Docker engine provides the `--link` option in the `docker run` subcommand to link a source container to a recipient container.

The format of the `--link` option is as follows:

```
--link <container>:<alias>
```

Here, `<container>` is the name of the source container and `<alias>` is the name seen by the recipient container. The name of the container must be unique in a Docker host, whereas alias is very specific and local to the recipient container, and hence, the alias need not be unique to the Docker host. This gives a lot of flexibility towards implementing and incorporating functionalities with a fixed source alias name inside the recipient container.

When two containers are linked together, the Docker engine automatically exports a few environment variables to the recipient container. These environment variables have a well-defined naming convention, where the variables are always prefixed with capitalized form of the alias name. For instance, if `src` is the alias name given to the source container, then the exported environment variables would begin with `SRC_`. Docker exports three categories of environment variables, as enumerated here:

- `NAME`: This is the first category of environment variables. This variable takes the form of `<ALIAS>_NAME`, and it carries the recipient container's hierarchical name as its value. For instance, if the source container's alias is `src` and the recipient container's name is `rec`, then the environment variable and its value would be `SRC_NAME=/rec/src`.
- `ENV`: This is the second category of environment variables. These variables export the environment variables configured in the source container by the `-e` option of the `docker run` subcommand or the `ENV` instruction of Dockerfile. This type of an environment variable takes the form of `<ALIAS>_ENV_<VAR_NAME>`. For instance, if the source container's alias is `src` and the variable name is `SAMPLE`, then the environment variable would be `SRC_ENV_SAMPLE`.
- `PORT`: This is the final and third category of environment variables that is used to export the connectivity details of the source container to the recipient. Docker creates a bunch of variables for each port exposed by the source container through the `-p` option of the `docker run` subcommand or the `EXPOSE` instruction of the Dockerfile.

These variables take the form:

```
*<ALIAS>_PORT_<port>_<protocol>
```

This form is used to share the source's IP address, port, and protocol as an URL. For example, if the source container's alias is `src`, the exposed port is `8080`, the protocol is `tcp`, and the IP address is `172.17.0.2`, then the environment variable and its value would be `SRC_PORT_8080_TCP=tcp://172.17.0.2:8080`. This URL further splits into the following three environment variables:

- `<ALIAS>_PORT_<port>_<protocol>_ADDR`: This form carries the IP address part of the URL (for example: `SRC_PORT_8080_TCP_ADDR=172.17.0.2`)

- `<ALIAS>_PORT_<port>_<protocol>_PORT`: This form carries the port part of the URL (for example: `SRC_PORT_8080_TCP_PORT=8080`)
- `<ALIAS>_PORT_<port>_<protocol>_PROTO`: This form carries the protocol part of the URL (for example: `SRC_PORT_8080_TCP_PROTO=tcp`)

In addition to the preceding environment variables, the Docker engine exports one more variable in this category, that is, of the form `<ALIAS>_PORT`, and its value would be the URL of the lowest number of all the exposed ports of the source container. For instance, if the source container's alias is `src`, the exposed port numbers are `7070`, `8080`, and `80`, the protocol is `tcp`, and the IP address is `172.17.0.2`, then the environment variable and its value would be `SRC_PORT=tcp://172.17.0.2:80`.

Docker exports these auto-generated environment variables in a well-structured format so that they can be easily discovered programmatically. Thus, it becomes very easy for the recipient container to discover the information about the source container. In addition, Docker automatically updates the source IP address and its alias as an entry in the recipient's `/etc/hosts` file.

In this chapter, we will take you deep into the mentioned features provided by the Docker engine for container linkage through a bevy of pragmatic examples.

To start with, let's choose a simple container linking example. Here, we will show you how to establish a linkage between two containers, and transfer some basic information from the source container to the recipient container, as illustrated in the following steps:

1. We begin by launching an interactive container that can be used as a source container for linking, using the following command:

   ```
   $ sudo docker run --rm --name example -it busybox:latest
   ```

   The container is named `example` using the `--name` option. In addition, the `--rm` option is used to clean up the container as soon as you exit from the container.

2. Display the `/etc/hosts` entry of the source container using the `cat` command:

   ```
   / # cat /etc/hosts
   172.17.0.3      a02895551686
   127.0.0.1       localhost
   ::1     localhost ip6-localhost ip6-loopback
   fe00::0 ip6-localnet
   ```

```
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Here, the first entry in the `/etc/hosts` file is the source container's IP address (`172.17.0.3`) and its hostname (`a02895551686`).

3. We will continue to display the environment variables of the source container using the `env` command:

```
/ # env
HOSTNAME=a02895551686
SHLVL=1
HOME=/root
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
bin
PWD=/
```

4. Now that we have launched the source container, from another terminal of the same Docker host, let's launch an interactive recipient container by linking it to our source container using the `--link` option of the `docker run` subcommand, as shown here:

```
$ sudo docker run --rm --link example:ex -it busybox:latest
```

Here, the source container named `example` is linked to the recipient container with `ex` as its alias.

5. Let's display the content of the `/etc/hosts` file of the recipient container using the `cat` command:

```
/ # cat /etc/hosts
172.17.0.4      a17e5578b98e
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.3      ex
```

Of course, as always, the first entry of the `/etc/hosts` file is the container's IP address and its hostname. However, the noteworthy entry in the `/etc/hosts` file is the last entry, where the source container's IP address (`172.17.0.3`) and its alias (`ex`) are added automatically.

6. We will continue to display the recipient container's environment variable using the `env` command:

```
/ # env
HOSTNAME=a17e5578b98e
SHLVL=1
HOME=/root
EX_NAME=/berserk_mcclintock/ex
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
bin
PWD=/
```

   Apparently, a new `EX_NAME` environment variable is added automatically to `/berserk_mcclintock/ex`, as its value. Here `EX` is the capitalized form of the alias `ex` and `berserk_mcclintock` is the auto-generated name of the recipient container.

7. As a final step, ping the source container using the widely used `ping` command for two counts, and use the alias name as the ping address:

```
/ # ping -c 2 ex
PING ex (172.17.0.3): 56 data bytes
64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.108 ms
64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.079 ms

--- ex ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.079/0.093/0.108 ms
```

Evidently, the source container's alias `ex` is resolved to the IP address `172.17.0.3`, and the recipient container is able to successfully reach the source. In the case of secured container communication, pinging between containers is not allowed. We have given more details on the aspect of securing containers in *Chapter 11*, *Securing Docker Containers*.

In the preceding example, we could link two containers together, and also, observe how elegantly, networking is enabled between the containers by updating the source container's IP address in the `/etc/hosts` file of the recipient container.

The next example is to demonstrate how container-linking exports the source container's environment variables, which are configured using the `-e` option of the `docker run` subcommand or the `ENV` instruction of `Dockerfile`, to the recipient container. For this purpose, we are going to craft a `Dockerfile` with the `ENV` instruction, build an image, launch a source container using this image, and then launch a recipient container by linking it to the source container:

1. We begin with composing a `Dockerfile` with the `ENV` instruction, as shown here:

   ```
   FROM busybox:latest
   ENV BOOK="Learning Docker" \
       CHAPTER="Orchestrating Containers"
   ```

   Here, we are setting up two environment variables `BOOK` and `CHAPTER`.

2. Proceed with building a Docker image `envex` using the `docker build` subcommand from the preceding `Dockerfile`:

   ```
   $ sudo docker build -t envex .
   ```

3. Now, let's launch an interactive source container with the name `example` using the `envex` image, we just built:

   ```
   $ sudo docker run -it --rm \
                       --name example envex
   ```

4. From the source container prompt, display all the environment variables by invoking the `env` command:

   ```
   / # env
   HOSTNAME=b53bc036725c
   SHLVL=1
   HOME=/root
   TERM=xterm
   PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
   bin
   BOOK=Learning Docker
   CHAPTER=Orchestrating Containers
   PWD=/
   ```

In all the preceding environment variables, both the BOOK and the CHAPTER variables are configured with the ENV instruction of the Dockerfile.

5. As a final step, to illustrate the ENV category of environment variables, launch the recipient container with the env command, as shown here:

```
$ sudo docker run --rm --link example:ex \
                  busybox:latest env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
bin
HOSTNAME=a5e0c07fd643
TERM=xterm
EX_NAME=/stoic_hawking/ex
EX_ENV_BOOK=Learning Docker
EX_ENV_CHAPTER=Orchestrating Containers
HOME=/root
```

This example is also available on GitHub at https://github.com/thedocker/learning-docker/blob/master/chap08/Dockerfile-Env.

Strikingly, in the preceding output, the variables that are prefixed with EX_ are the outcomes of container-linking. The environment variables of interest are EX_ENV_BOOK and EX_ENV_CHAPTER, which were originally set through the Dockerfile as BOOK and CHAPTER but modified to EX_ENV_BOOK and EX_ENV_CHAPTER, as an effect of container-linking. Though the environment variable names get translated, the values stored in these environment variables are preserved as is. We already discussed the EX_NAME variable name in the previous example.

In the preceding example, we could experience how elegantly and effortlessly Docker exports the ENV category variables from the source container to the recipient container. These environment variables are completely decoupled from the source and the recipient, thus the change in the value of these environment variables in one container does not impact the other. To be even more precise, the values the recipient container receives are the values set during the launch time of the source container. Any changes made to the value of these environment variables in the source container after its launch has no effect on the recipient container. It does not matter when the recipient container is launched because the values are being read from the JSON file.

In our final illustration of linking containers, we are going to show you how to take advantage of the Docker feature to share the connectivity details between two containers. In order to share the connectivity details between containers, Docker uses the `PORT` category of environment variables. The following are the steps used to craft two containers and share the connectivity details between them:

1. Craft a `Dockerfile` to expose port `80` and `8080` using the `EXPOSE` instruction, as shown here:

   ```
   FROM busybox:latest
   EXPOSE 8080 80
   ```

2. Proceed to build a Docker image `portex` using the `docker build` subcommand from the `Dockerfile`, we created just now, by running the following command:

   ```
   $ sudo docker build -t portex .
   ```

3. Now, let's launch an interactive source container with the name, `example` using the earlier built image `portex`:

   ```
   $ sudo docker run -it --rm \
                     --name example portex
   ```

4. Now that we have launched the source container, let's continue to create a recipient container on another terminal by linking it to the source container, and invoke the `env` command to display all the environment variables, as shown here:

   ```
   $ sudo docker run --rm --link example:ex \
                     busybox:latest env
   PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
   bin
   HOSTNAME=c378bb55e69c
   TERM=xterm
   EX_PORT=tcp://172.17.0.4:80
   EX_PORT_80_TCP=tcp://172.17.0.4:80
   EX_PORT_80_TCP_ADDR=172.17.0.4
   EX_PORT_80_TCP_PORT=80
   EX_PORT_80_TCP_PROTO=tcp
   EX_PORT_8080_TCP=tcp://172.17.0.4:8080
   EX_PORT_8080_TCP_ADDR=172.17.0.4
   EX_PORT_8080_TCP_PORT=8080
   EX_PORT_8080_TCP_PROTO=tcp
   ```

```
EX_NAME=/prickly_rosalind/ex
HOME=/root
```

This example is also available on GitHub at `https://github.com/thedocker/learning-docker/blob/master/chap08/Dockerfile-Expose`.

From the preceding output of the `env` command, it is quite evident that, the Docker engine exported a bunch of four `PORT` category environment variables for each port that was exposed using the `EXPOSE` instruction in the `Dockerfile`. In addition, Docker also exported another `PORT` category variable `EX_PORT`.

Orchestration of containers

The pioneering concept of orchestration in the IT domain has been there for a long time now. For instance, in the **service computing** (**SC**) arena, the idea of service orchestration has been thriving in an unprecedented manner in order to produce and sustain highly robust and resilient services. Discrete or atomic services do not serve any substantial purpose unless, they are composed together in a particular sequence to derive process-aware composite services. As orchestrated services are more strategically advantageous for businesses in expressing and exposing their unique capabilities in the form of identifiable/discoverable, interoperable, usable, and composable services to the outside world; corporates are showing exemplary interest in having an easily searchable repository of services (atomic as well as composite). This repository, in turn, enables businesses in realizing large-scale data as well as process-intensive applications. It is clear that the multiplicity of services is very pivotal for organizations to grow and glow. This increasingly mandated requirement gets solved using the proven and promising orchestration capabilities, cognitively.

Now, as we are fast heading toward containerized IT environments; application and data containers ought to be smartly composed to realize a host of new generation software services.

However, for producing highly competent orchestrated containers, both purpose-specific as well as agnostic containers need to be meticulously selected and launched in the right sequence in order to create orchestrated containers. The sequence can come from the process (control as well as data) flow diagrams. Doing this complicated and daunting activity manually evokes a series of cynicisms and criticisms. Fortunately, there are orchestration tools in the Docker space that come in handy to build, run, and manage multiple containers to build enterprise-class services. The Docker firm, which has been in charge of producing and promoting the generation and assembling of Docker-inspired containers, has come out with a standardized and simplified orchestration tool (named as `docker-compose`) in order to reduce the workloads of developers as well as system administrators.

The proven composition technique of the service computing paradigm is being replicated here in the raging containerization paradigm in order to reap the originally envisaged benefits of containerization, especially in building powerful application-aware containers.

The **microservice architecture** is an architectural concept that aims to decouple a software solution by decomposing its functionality into a pool of discrete services. This is done by applying the standard principles at the architectural level. The microservice architecture is slowly emerging as a championed way to design and build large-scale IT and business systems. It not only facilitates loose and light coupling and software modularity, but it is also a boon to continuous integration and deployment for the agile world. Any changes being made to one part of the application mandate, has meant massive changes being made to the application. This has been a bane and barrier to the aspect of continuous deployment. Micro services aim to resolve this situation, and hence, the microservice architecture needs light-weight mechanisms, small, independently deployable services to ensure scalability and portability. These requirements can be met using Docker-sponsored containers.

Micro services are being built around business capabilities and can be independently deployed by fully automated deployment machinery. Each micro service can be deployed without interrupting the other micro services, and containers provide an ideal deployment and execution environment for these services along with other noteworthy facilities, such as the reduced time of deployment, isolation management, and a simple life cycle. It is easy to quickly deploy new versions of services inside containers. All of these factors led to the explosion of micro services using the features that Docker had to offer.

As explained, Docker is being posited as the next-generation containerization technology, which provides a proven and potentially sound mechanism to distribute applications in a highly efficient and distributed fashion. The beauty is that developers can tweak the application pieces within the container, while maintaining the overall integrity of the container. This has a bigger impact as the brewing trend is that instead of large monolithic applications hosted on a single physical or virtual server, companies are building smaller, self-defined, easily manageable, and discrete services to be contained inside standardized and automated containers. In short, the raging containerization technology from Docker has come as a boon for the ensuing era of micro services.

Docker was built and sustained to fulfill the elusive goal of *run it once and run it everywhere*. Docker containers are generally isolated at process level, portable across IT environments, and easily repeatable. A single physical host can host multiple containers, and hence, every IT environment is generally stuffed with a variety of Docker containers. The unprecedented growth of containers is to spell out troubles for effective container management. The multiplicity and the associated heterogeneity of containers are used to sharply increase the management complexities of containers. Hence, the technique of orchestration and the flourishing orchestration tools have come as a strategic solace for accelerating the containerization journey in safe waters.

Orchestrating applications that span multiple containers containing micro services has become a major part of the Docker world, via projects, such as Google's Kubernetes or Flocker. Decking is another option used to facilitate the orchestration of Docker containers. Docker's new offering in this area is a set of three orchestration services designed to cover all aspects of the dynamic life cycle of distributed applications from application development to deployment and maintenance. Helios is another Docker orchestration platform used to deploy and manage containers across an entire fleet. In the beginning, `fig` was the most preferred tool for container orchestration. However, in the recent past, the company at the forefront of elevating, the Docker technology has come out with an advanced container orchestration tool (`docker-compose`) to make life easier for developers working with Docker containers, as they move through the container life cycle.

Having realized the significance of having the capability of container orchestration for next generation, business-critical, and containerized workloads, the Docker company purchased the company that originally conceived and concretized the `fig` tool. Then, the Docker company appropriately renamed the tool as `docker-compose` and brought in a good number of enhancements to make the tool more tuned to the varying expectations of container developers and operation teams.

Here is a gist of `docker-compose`, which is being positioned as a futuristic and flexible tool used for defining and running complex applications with Docker. With `docker-compose`, you define your application's components (their containers, configuration, links, volumes, and so on) in a single file, and then, you can spin everything up with a single command, which does everything to get it up and running.

This tool simplifies the container management by providing a set of built-in tools to do a number of jobs that are being performed manually at this point of time. In this section, we supply all the details for using `docker-compose` to perform orchestration of containers in order to have a stream of next-generation distributed applications.

Orchestrate containers using docker-compose

In this section, we will discuss the widely used container orchestration tool `docker-compose`. The `docker-compose` tool is a very simple, yet powerful tool and has been conceived and concretized to facilitate the running of a group of Docker containers. In other words, `docker-compose` is an orchestration framework that lets you define and control a multi-container service. It enables you to create a fast and isolated development environment as well as the ability to orchestrate multiple Docker containers in production. The `docker-compose` tool internally leverages the Docker engine for pulling images, building the images, starting the containers in a correct sequence, and making the right connectivity/linking among the containers/services based on the definition given in the `docker-compose.yml` file.

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:

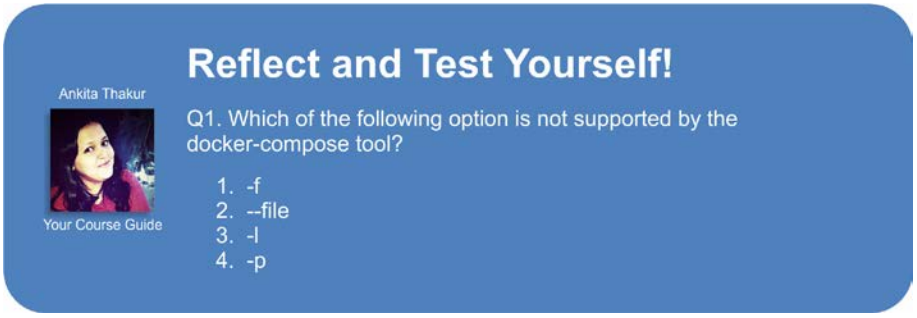
```
node2:
  image:
    scottpgallagher/galeranode
  hostname:
    database
  environment:
    MARIADB_DATABASE: wordpressmu
    MARIADB_USER: replica
```

```
    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 of Module 1 Chapter 10

This chapter has been incorporated in the book in order to provide all the probing and prescribing details on seamlessly orchestrating multiple containers. We extensively discussed the need for container orchestration and the tools that enable us to simplify and streamline the increasingly complicated process of container orchestration. In order to substantiate how orchestration is handy and helpful in crafting enterprise-class containers, and to illustrate the orchestration process, we took the widely followed route of explaining the whole gamut through a simple example. We developed a web application and contained it within a standard container. Similarly, we took a database container, which is a backend for the frontend web application and the database was executed inside another container.

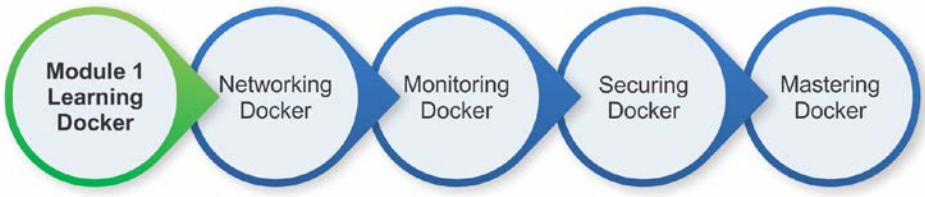
Ankita Thakur

Your Course Guide

We saw how to make the web application container aware of the database, using different technologies through the container-linkage feature of the Docker engine. We used the open source tool (docker-compose) for this purpose.

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.

Your Progress through the Course So Far



11
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

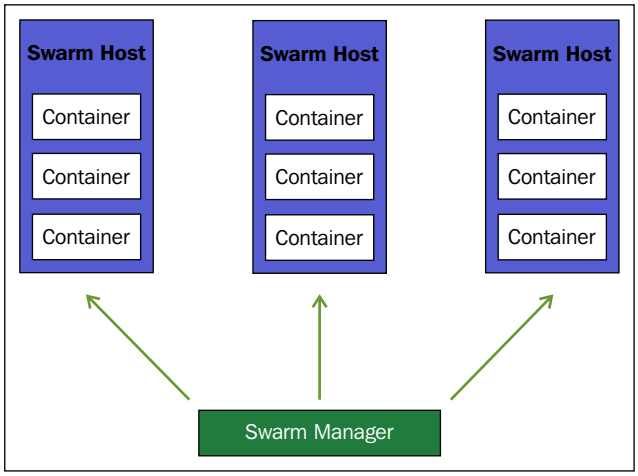
Docker Swarm is the container that runs on each Swarm host. Swarm uses a unique token for each cluster to be able to join the cluster. The Swarm container itself is the one that communicates on behalf of that Docker host to the other Docker hosts that are running Docker Swarm as well as the Docker Swarm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

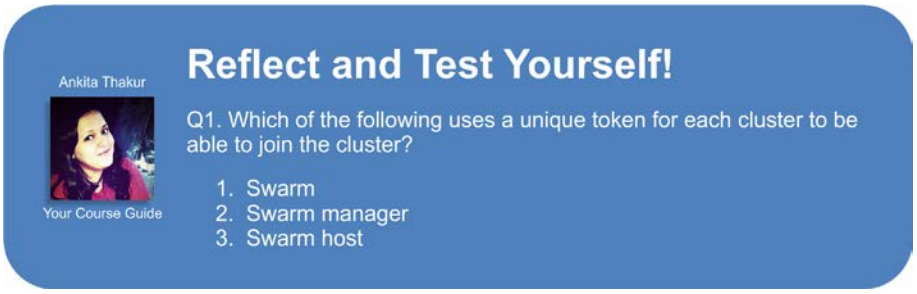
We have seen how we can create a Swarm cluster as well. What this command actually does is it gives us the token that we need to create the cluster and join all the nodes to it. There are no other switches that can be used with this command as we have seen with other commands:

```
$ 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/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 of Module 1 Chapter 11

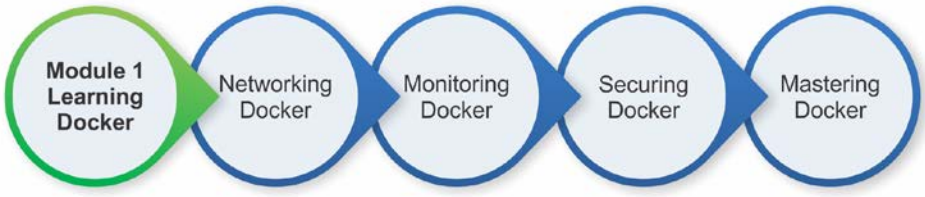
Ankita Thakur

Your Course Guide

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 discuss how Docker facilitates software testing, especially integration testing with a few pragmatic examples.

Your Progress through the Course So Far



12
Testing with Docker

Undoubtedly, the trait of testing has been at the forefront of the software engineering discipline. It is widely accepted that there is a deep and decisive penetration of software into every kind of tangible object in our daily environments these days in order to have plenty of smart, connected, and digitized assets. Also, with a heightened focus on distributed and synchronized software, the complexity of the software design, development, testing and debugging, deployment, and delivery are continuously on the climb. There are means and mechanisms being unearthed to simplify and streamline the much-needed automation of software building and the authentication of software reliability, resiliency, and sustainability. Docker is emerging as an extremely flexible tool to test a wide variety of software applications. In this chapter, we are going to discuss how to effectively leverage the noteworthy Docker advancements for software testing and its unique advantages in accelerating and augmenting testing automation.

The following topics are discussed in this chapter:

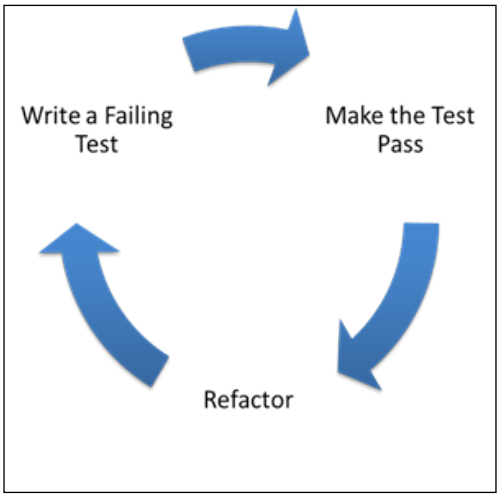
- A brief overview of **test-driven development** (**TDD**)
- Testing your code inside Docker
- Integrating the Docker testing process into Jenkins

The emerging situation is that Docker containers are being leveraged to create development and testing environments that are the exact replicas of the production environment. Containers require less overhead when compared to virtual machines, which have been the primary environments for development, staging, and deployment environments. Let's start with an overview of test-driven development of the next generation software and how the Docker-inspired containerization becomes handy in simplifying the TDD process.

A brief overview of the test-driven development

The long and arduous journey of software development has taken many turns and twists in the past decades, and one of the prominent software engineering techniques is nonetheless the TDD. There are more details and documents on TDD at `http://agiledata.org/essays/tdd.html`.

In a nutshell, the test-driven development, also popularly known as TDD, is a software development practice in which the development cycle begins with writing a test case that would fail, then writes the actual software to make the test pass, and continues to refactor and repeat the cycle till the software reaches the acceptable level. This process is depicted in the following diagram:



Testing your code inside Docker

In this section, we will take you through a journey in which we will show you how TDD is done using stubs, and how Docker can come handy in developing software in the deployment equivalent system. For this purpose, we take a web application use case that has a feature to track the visit count of each of its users. For this example, we use Python as the implementation language and `redis` as the key-value pair database to store the users hit count. Besides, to showcase the testing capability of Docker, we limit our implementation to just two functions: `hit` and `getHit`.

NOTE: All the examples in this chapter use `python3` as the runtime environment. The `ubuntu 14.04` installation comes with `python3` by default. If you don't have `python3` installed on your system, refer to the respective manual to install `python3`.

As per the TDD practice, we start by adding unit test cases for the `hit` and `getHit` functionalities, as depicted in the following code snippet. Here, the test file is named `test_hitcount.py`:

```
import unittest
import hitcount

class HitCountTest (unittest.TestCase):
     def testOneHit(self):
         # increase the hit count for user user1
         hitcount.hit("user1")
         # ensure that the hit count for user1 is just 1
         self.assertEqual(b'1', hitcount.getHit("user1"))

if __name__ == '__main__':
    unittest.main()
```

This example is also available at `https://github.com/thedocker/testing/tree/master/src`.

Here, in the first line, we are importing the `unittest` Python module that provides the necessary framework and functionality to run the unit test and generate a detailed report on the test execution. In the second line, we are importing the `hitcount` Python module, where we are going to implement the hit count functionality. Then, we will continue to add the test code that would test the `hitcount` module's functionality.

Now, run the test suite using the unit test framework of Python, as follows:

```
$ python3 -m unittest
```

The following is the output generated by the unit test framework:

```
E
======================================================================
ERROR: test_hitcount (unittest.loader.ModuleImportFailure)
----------------------------------------------------------------------
Traceback (most recent call last):
...OUTPUT TRUNCATED ...
ImportError: No module named 'hitcount'



----------------------------------------------------------------------
Ran 1 test in 0.001s


FAILED (errors=1)
```

As expected, the test failed with the error message `ImportError: No module named 'hitcount'` because we had not even created the file and hence, it could not import the `hitcount` module.

Now, create a file with the name `hitcount.py` in the same directory as `test_hitcount.py`:

```
$ touch hitcount.py
```

Continue to run the unit test suite:

```
$ python3 -m unittest
```

The following is the output generated by the unit test framework:

```
E
======================================================================
ERROR: testOneHit (test_hitcount.HitCountTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/user/test_hitcount.py", line 10, in testOneHit
    hitcount.hit("peter")
```

```
AttributeError: 'module' object has no attribute 'hit'

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
```

Again the test suite failed like the earlier but with a different error message `AttributeError: 'module' object has no attribute 'hit'`. We are getting this error because we have not implemented the `hit` function yet.

Let's proceed to implement the `hit` and `getHit` functions in `hitcount.py`, as shown here:

```
import redis
# connect to redis server
r = redis.StrictRedis(host='0.0.0.0', port=6379, db=0)

# increase the hit count for the usr
def hit(usr):
    r.incr(usr)

# get the hit count for the usr
   def getHit(usr):
    return (r.get(usr))
```

This example is also available on GitHub at `https://github.com/thedocker/testing/tree/master/src`.

Note: To continue with this example, you must have the `python3` compatible version of package installer (`pip3`).

The following command is used to install `pip3`:

```
$ wget -qO- https://bootstrap.pypa.io/get-pip.py | sudo python3 -
```

In the first line of this program, we are importing the `redis` driver, which is the connectivity driver of the `redis` database. In the following line, we are connecting to the `redis` database, and then we will continue to implement the `hit` and `getHit` function.

The `redis` driver is an optional Python module, so let's proceed to install the `redis` driver using the `pip` installer, which is illustrated as follows:

```
$ sudo pip3 install redis
```

Our `unittest` will still fail even after installing the `redis` driver because we are not running a `redis` database server yet. So, we can either run a `redis` database server to successfully complete our unit testing or take the traditional TDD approach of mocking the `redis` driver. Mocking is a testing approach wherein complex behavior is substituted by predefined or simulated behavior. In our example, to mock the redis driver, we are going to leverage a third-party Python package called `mockredis`. This mock package is available at `https://github.com/locationlabs/mockredis` and the `pip` installer name is `mockredispy`. Let's install this mock using the `pip` installer:

```
$ sudo pip3 install mockredispy
```

Having installed `mockredispy`, the `redis` mock, let's refactor our test code `test_hitcount.py` (which we had written earlier) to use the simulated `redis` functionality provided by the `mockredis` module. This is accomplished by the patch method provided by the `unittest.mock` mocking framework, as shown in the following code:

```
import unittest
from unittest.mock import patch

# Mock for redis
import mockredis
import hitcount

class HitCountTest(unittest.TestCase):


@patch('hitcount.r',mockredis.mock_strict_redis_client(host='0.0.0.0',
port=6379,
db=0))
    def testOneHit(self):
        # increase the hit count for user user1
        hitcount.hit("user1")
        # ensure that the hit count for user1 is just 1
        self.assertEqual(b'1', hitcount.getHit("user1"))

if __name__ == '__main__':
    unittest.main()
```

This example is also available on GitHub at `https://github.com/thedocker/testing/tree/master/src`.

Now, run the test suite again:

```
$ python3 -m unittest
.
----------------------------------------------------------------------
-
Ran 1 test in 0.000s


OK
```

Finally, as we can see in the preceding output, we successfully implemented our visitors count functionality through the test, code, and refactor cycle.

Running the test inside a container

In the previous section, we walked you through the complete cycle of TDD, in which we installed additional Python packages to complete our development. However, in the real world, one might work on multiple projects that might have conflicting libraries and hence, there is a need for the isolation of runtime environments. Before the advent of Docker technology, the Python community used to leverage the `virtualenv` tool to isolate the Python runtime environment. Docker takes this isolation a step further by packaging the OS, the Python tool chain, and the runtime environment. This type of isolation gives a lot of flexibility to the development community to use appropriate software versions and libraries as per the project needs.

Here is the step-by-step procedure to package the test and visitor count implementation of the previous section to a Docker container and perform the test inside the container:

1. Craft a `Dockerfile` to build an image with the `python3` runtime, the `redis` and `mockredispy` packages, both the `test_hitcount.py` test file and the visitors count implementation `hitcount.py`, and finally, launch the unit test:

```
###############################################
# Dockerfile to build the unittest container
###############################################
```

```
# Base image is python
FROM python:latest

# Author: Dr. Peter
MAINTAINER Dr. Peter <peterindia@gmail.com>

# Install redis driver for python and the redis mock
RUN pip install redis && pip install mockredispy

# Copy the test and source to the Docker image
ADD src/ /src/

# Change the working directory to /src/
WORKDIR /src/

# Make unittest as the default execution
ENTRYPOINT python3 -m unittest
```

This example is also available on GitHub at `https://github.com/thedocker/testing/tree/master/src`.

2. Now create a directory called `src` on the directory, where we crafted our `Dockerfile`. Move the `test_hitcount.py` and `hitcount.py` files to the newly created `src` directory.
3. Build the `hit_unittest` Docker image using the `docker build` subcommand:

```
$ sudo docker build -t hit_unittest .
Sending build context to Docker daemon 11.78 kB
Sending build context to Docker daemon
Step 0 : FROM python:latest
 ---> 32b9d937b993
Step 1 : MAINTAINER Dr. Peter <peterindia@gmail.com>
 ---> Using cache
 ---> bf40ee5f5563
Step 2 : RUN pip install redis && pip install mockredispy
 ---> Using cache
 ---> a55f3bdb62b3
```

```
Step 3 : ADD src/ /src/
 ---> 526e13dbf4c3
Removing intermediate container a6d89cbce053
Step 4 : WORKDIR /src/
 ---> Running in 5c180e180a93
 ---> 53d3f4e68f6b
Removing intermediate container 5c180e180a93
Step 5 : ENTRYPOINT python3 -m unittest
 ---> Running in 74d81f4fe817
 ---> 063bfe92eae0
Removing intermediate container 74d81f4fe817
Successfully built 063bfe92eae0
```

4. Now that we have successfully built the image, let's launch our container with the unit testing bundle using the `docker run` subcommand, as illustrated here:

```
$ sudo docker run --rm -it hit_unittest .
----------------------------------------------------------------------
----
Ran 1 test in 0.001s


OK
```

Apparently, the unit test ran successfully with no errors because we already packaged the tested code.

In this approach, for every change, the Docker image is built and then, the container is launched to complete the test.

Using a Docker container as a runtime environment

In the previous section, we built a Docker image to perform the testing. Particularly, in the TDD practice, the unit test cases and the code go through multiple changes. Consequently, the Docker image needs to be built over and over again, which is a daunting job. In this section, we will see an alternative approach in which the Docker container is built with a runtime environment, the development directory is mounted as a volume, and the test is performed inside the container.

During this TDD cycle, if an additional library or update to the existing library is required, then the container will be updated with the required libraries and the updated container will be committed as a new image. This approach gives the isolation and flexibility that any developer would dream of because the runtime and its dependency live within the container, and any misconfigured runtime environment can be discarded and a new runtime environment can be built from a previously working image. This also helps to preserve the sanity of the Docker host from the installation and uninstallation of libraries.

The following example is a step-by-step instruction on how to use the Docker container as a nonpolluting yet very powerful runtime environment:

1. We begin with launching the Python runtime interactive container, using the `docker run` subcommand:

   ```
   $ sudo docker run -it \
             -v /home/peter/src/hitcount:/src \
             python:latest /bin/bash
   ```

   Here, in this example, the `/home/peter/src/hitcount` Docker host directory is earmarked as the placeholder for the source code and test files. This directory is mounted in the container as `/src`.

2. Now, on another terminal of the Docker host, copy both the `test_hitcount.py` test file and the visitors count implementation `hitcount.py` to `/home/peter/src/hitcount directory`.
3. Switch to the Python runtime interactive container terminal, change the current working directory to `/src`, and run the unit test:

   ```
   root@a8219ac7ed8e:~# cd /src
   root@a8219ac7ed8e:/src# python3 -m unittest
   E
   ======================================================================
   ERROR: test_hitcount (unittest.loader.ModuleImportFailure)
   . . . TRUNCATED OUTPUT . . .
     File "/src/test_hitcount.py", line 4, in <module>
       import mockredis
   ImportError: No module named 'mockredis'


   ----------------------------------------------------------------------
   ```

```
Ran 1 test in 0.001s

FAILED (errors=1)
```

Evidently, the test failed because it could not find the `mockredis` Python library.

4. Proceed to install the `mockredispy pip` package because the previous step failed, as it could not find the `mockredis` library in the runtime environment:

```
root@a8219ac7ed8e:/src# pip install mockredispy
```

5. Rerun the Python unit test:

```
root@a8219ac7ed8e:/src# python3 -m unittest
E
======================================================================
ERROR: test_hitcount (unittest.loader.ModuleImportFailure)
. . . TRUNCATED OUTPUT . . .
  File "/src/hitcount.py", line 1, in <module>
    import redis
ImportError: No module named 'redis'


Ran 1 test in 0.001s

FAILED (errors=1)
```

Again, the test failed because the `redis` driver is not yet installed.

6. Continue to install the `redis` driver using the pip installer, as shown here:

```
root@a8219ac7ed8e:/src# pip install redis
```

7. Having successfully installed the `redis` driver, let's once again run the unit test:

```
root@a8219ac7ed8e:/src# python3 -m unittest
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Apparently, this time the unit test passed with no warnings or error messages.

8. Now we have a runtime environment that is good enough to run our test cases. It is better to commit these changes to a Docker image for reuse, using the `docker commit` subcommand:

```
$ sudo docker commit a8219ac7ed8e python_rediswithmock
fcf27247ff5bb240a935ec4ba1bddbd8c90cd79cba66e52b21e1b48f984c7db2
```

From now on, we can use the `python_rediswithmock` image to launch new containers for our TDD.

In this section, we vividly illustrated the approach on how to use the Docker container as a testing environment, and also at the same time, preserve the sanity and sanctity of the Docker host by isolating and limiting the runtime dependency within the container.

Integrating Docker testing into Jenkins

In the previous section, we laid out a stimulating foundation on software testing, how to leverage the Docker technology for the software testing, and the unique benefits of the container technology during the testing phase. In this section, we will introduce you to the steps required to prepare the Jenkins environment for testing with Docker, and then, demonstrate how Jenkins can be extended to integrate and automate testing with Docker using the well-known hit count use case.

Preparing the Jenkins environment

In this section, we will take you through the steps to install `jenkins`, GitHub plugin for Jenkins and `git`, and the revision control tool. These steps are as follows:

1. We begin with adding the Jenkins' trusted PGP public key:

```
$ wget -q -O - \
    https://jenkins-ci.org/debian/jenkins-ci.org.key | \
    sudo apt-key add -
```

Here, we are using `wget` to download the PGP public key, and then we add it to the list of trusted keys using the apt-key tool. Since Ubuntu and Debian share the same software packaging, Jenkins provides a single common package for both Ubuntu and Debian.

2. Add the Debian package location to the `apt` package source list, as follows:

```
$ sudo sh -c \
   'echo deb http://pkg.jenkins-ci.org/debian binary/ > \
    /etc/apt/sources.list.d/jenkins.list'
```

3. Having added the package source, continue to run the `apt-get` command update option to resynchronize the package index from the sources:

```
$ sudo apt-get update
```

4. Now, install `jenkins` using the `apt-get` command install option, as demonstrated here:

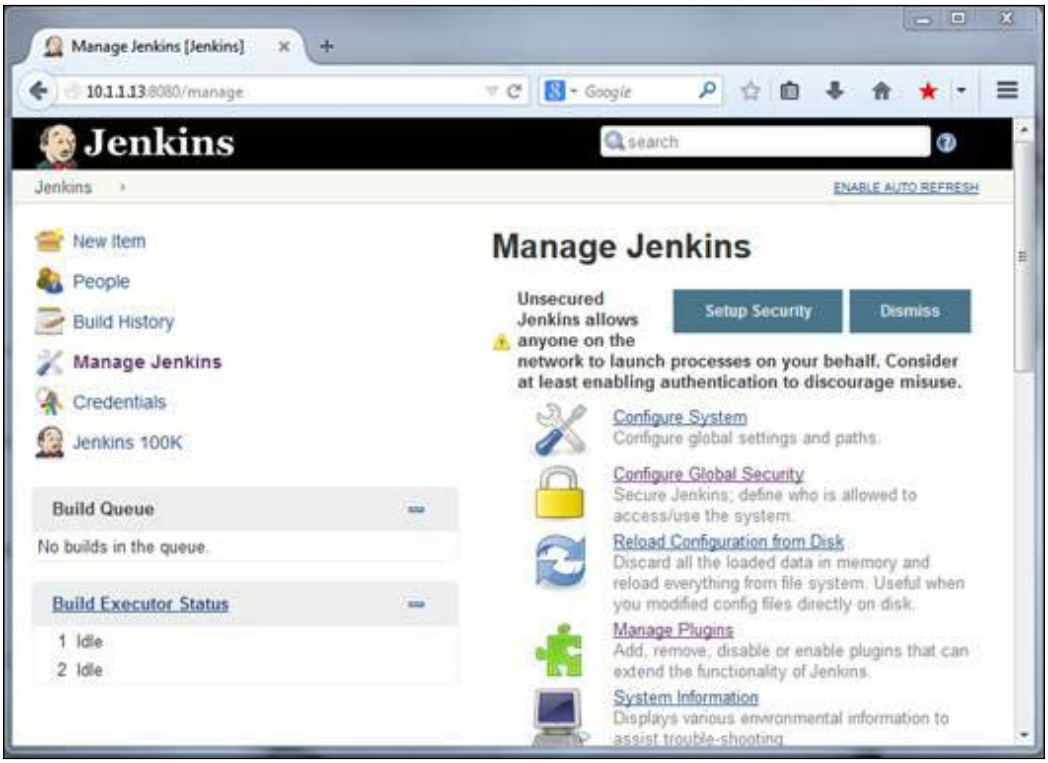
```
$ sudo apt-get install jenkins
```

5. Finally, activate the `jenkins` service using the `service` command:

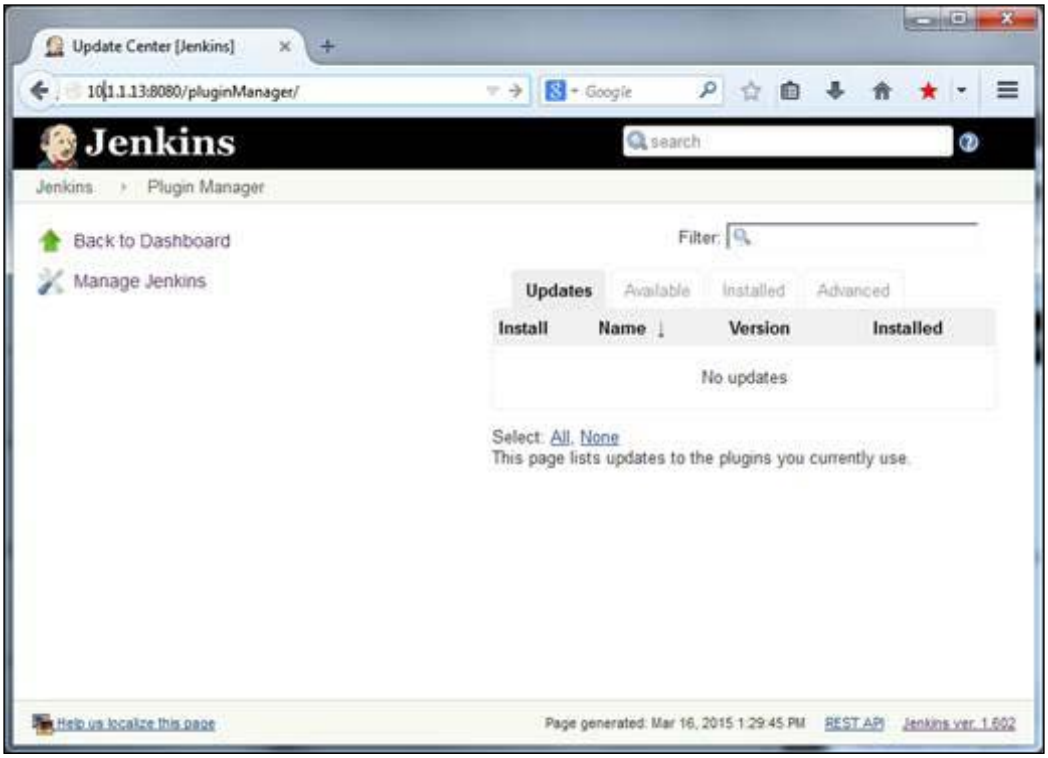
```
$ sudo service jenkins start
```

6. The `jenkins` service can be accessed through any web browsers by specifying the IP address (`10.1.1.13`) of the system in which Jenkins is installed. The default port number for Jenkins is `8080`. The following screenshot is the entry page or **Dashboard** of **Jenkins**:

7. In this example, we are going to use GitHub as the source code repository. Jenkins does not support GitHub by default and hence, we need to install the GitHub plugin. During the installation, sometimes Jenkins does not populate the plugin availability list, and hence, you have to force it to download the list of available plugins. You can do so by performing the following steps:
    1. Select **Manage Jenkins** on the left-hand side of the screen, which will take us to a **Manage Jenkins** page, as shown in the following screenshot:



2. On the **Manage Jenkins** page, select **Manage Plugins** and this will take us to the **Plugin Manager** page, as shown in the following screenshot:

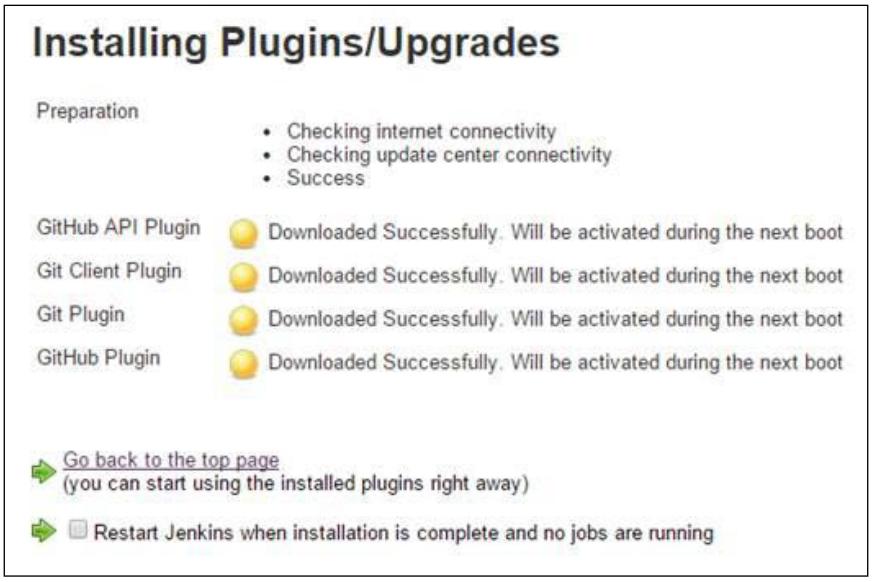


3. Here, on the **Plugin Manager** page, select the **Advanced** tab, go to the bottom of this page, and you will find the **Check now** button in the right-hand side corner of the page. Click on the **Check now** button to start the plugin updates. Alternatively, you can directly go to the **Check now** button on the **Advanced** page by navigating to `http://<jenkins-server>:8080/pluginManager/advanced`, wherein `<jenkins-server>` is the IP address of the system in which Jenkins is installed.

NOTE: If Jenkins does not update the available plugin list, it is most likely a mirror site issue, so modify the **Update Site** field with a working mirror URL.

8. Having updated the available plugin list, let's continue to install the GitHub plugin, as depicted in the following substeps:
    1. Select the **Available** tab in the **Plugin Manager** page, which will list all the available plugins.
    2. Type `GitHub plugin` as the filter, which will list just the GitHub plugin, as shown in the following screenshot:

    3. Select the checkbox, and click on **Download now and install after restart**. You will be taken to a screen that will show you the progress of the plugin installation:



4. After all the plugins have successfully downloaded, go ahead and restart Jenkins using `http://< jenkins-server >:8080/restart`, where `<jenkins-server>` is the IP address of the system in which Jenkins is installed.

9. Ensure that the `git` package is installed, otherwise install the `git` package using the `apt-get` command:

```
$ sudo apt-get install git
```

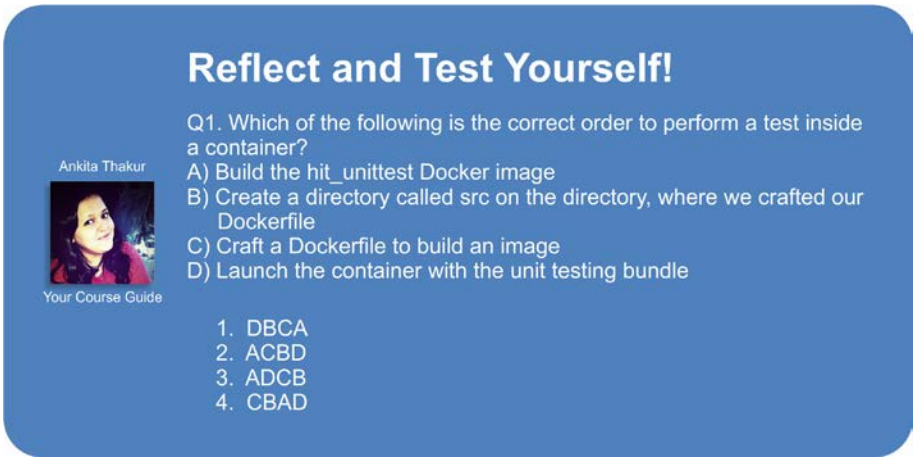
10. So far, we have been running the Docker client using the `sudo` command, but unfortunately, we could not invoke `sudo` inside Jenkins because sometimes it prompts for a password. To overcome the `sudo` password prompt issue, we can make use of the Docker group, wherein any user who is part of the Docker group can invoke the Docker client without using the `sudo` command. Jenkins installation always sets up a user and group called `jenkins` and runs the Jenkins server using that user and group. So, we just need to add the `jenkins` user to the Docker group to get the Docker client working without the `sudo` command:

```
$ sudo gpasswd -a jenkins docker
Adding user jenkins to group docker
```

11. Restart the `jenkins` service for the group change to take effect using the following command:

```
$ sudo service jenkins restart
 * Restarting Jenkins Continuous Integration Server jenkins
[ OK ]
```

We have set up a Jenkins environment that is now capable of automatically pulling the latest source code from the `http://github.com` repository, packaging it as a Docker image, and executing the prescribed test scenarios.

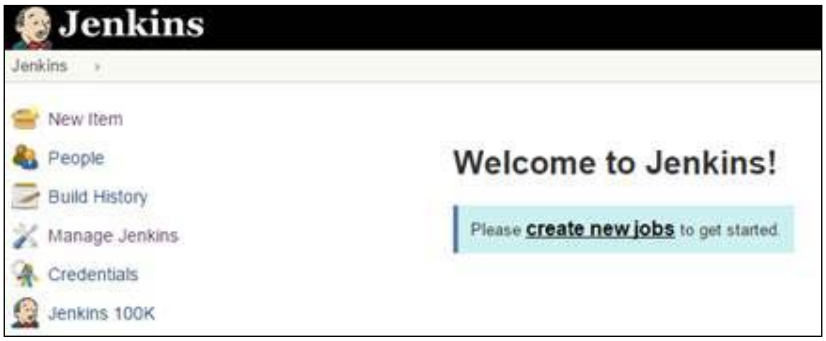


Automating the Docker testing process

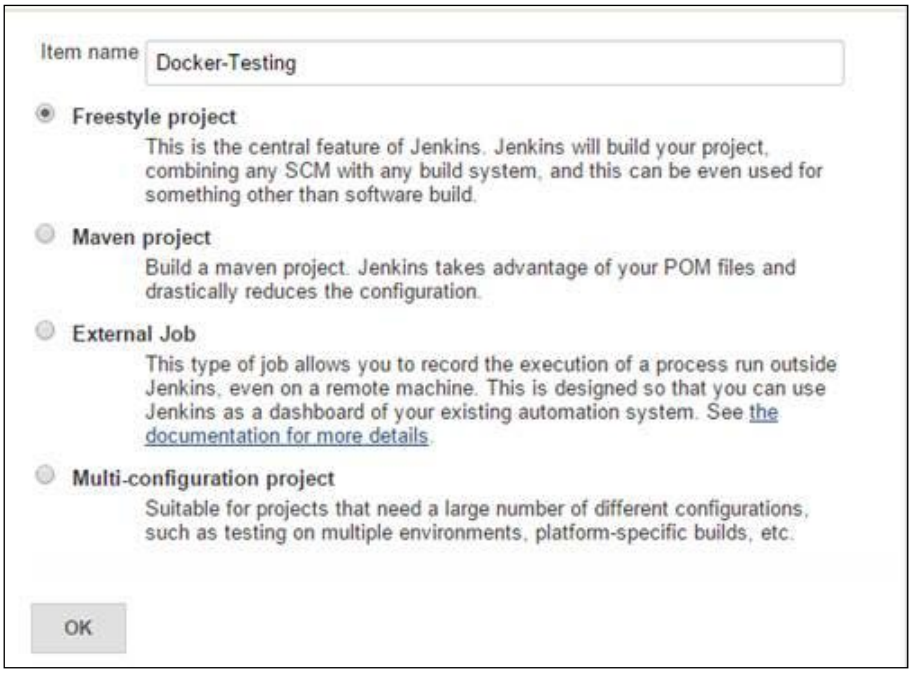
In this section, we will explore how to automate testing using Jenkins and Docker. As mentioned earlier, we are going to use GitHub as our repository. We have already uploaded the `Dockerfile`, `test_hitcount.py`, and `hitcount.py` files of our previous example to GitHub at `https://github.com/thedocker/testing`, which we are to use in the ensuing example. However, we strongly encourage you to set up your own repository at `http://github.com`, using the fork option that you can find at `https://github.com/thedocker/testing`, and substitute this address wherever applicable in the ensuing example.

The following are the detailed steps to automate the Docker testing:

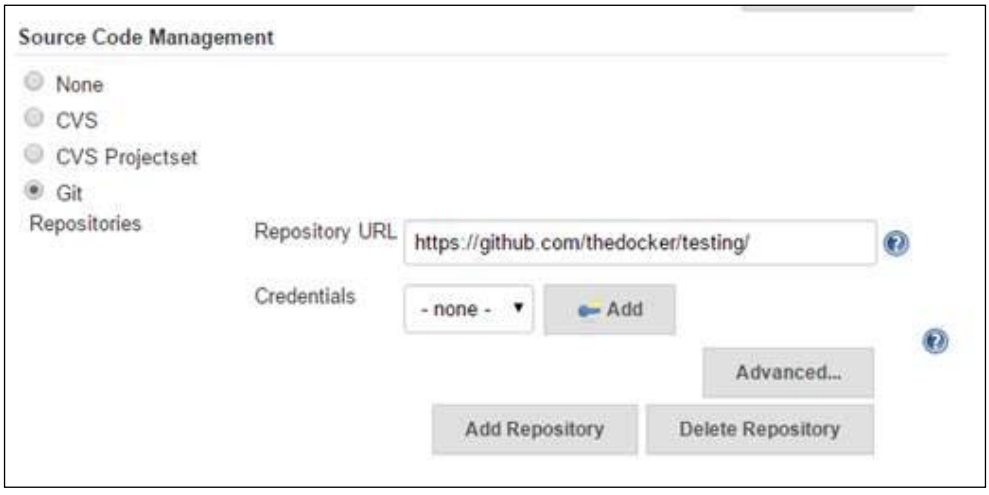
1. Configure Jenkins to trigger a build when a file is modified in the GitHub repository, which is illustrated in the following substeps:
    1. Connect to the Jenkins server again.
    2. Select either **New Item** or **create new jobs**:



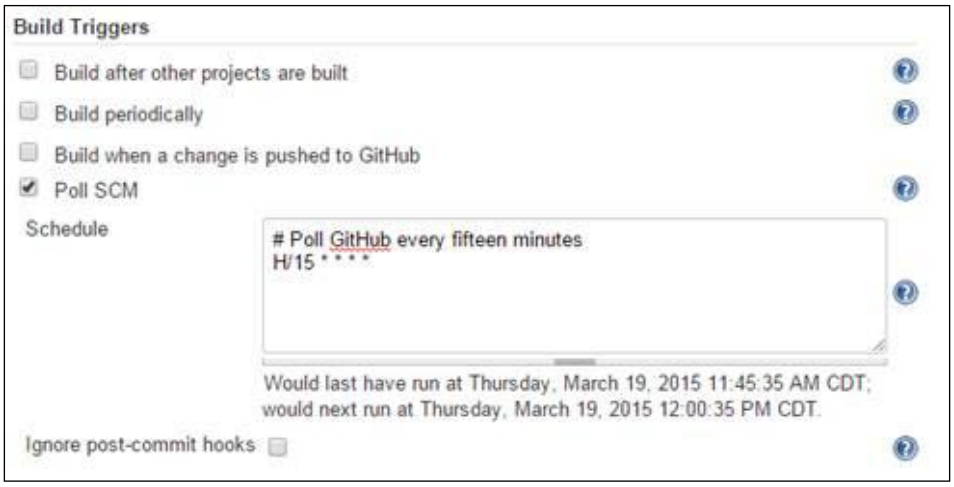
3. In the following screenshot, give a name to the project (for example, `Docker-Testing`), and select the **Freestyle project** radio button:



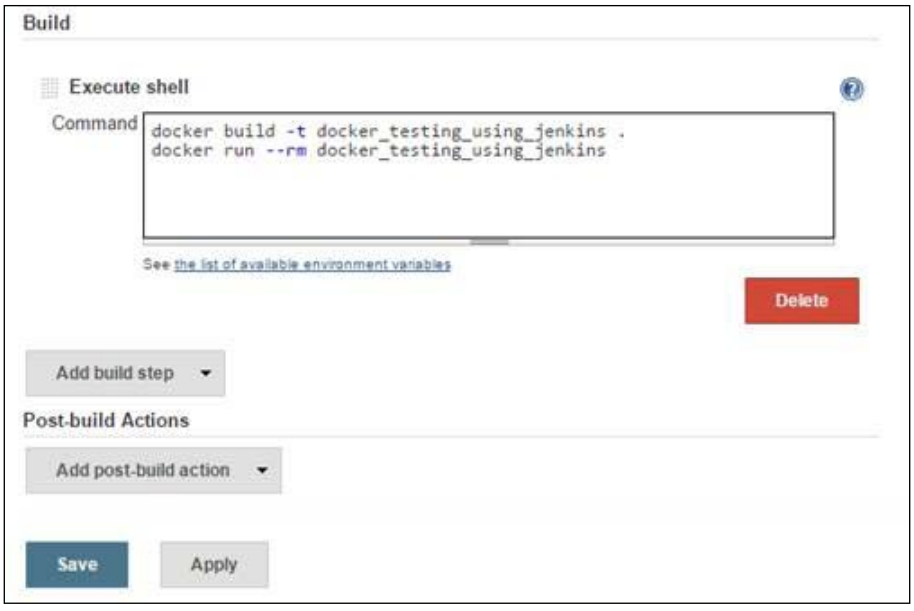
4. In the next screenshot, select the **Git** radio button under **Source Code Management**, and specify the GitHub repository URL in the **Repository URL** text field:



5. Select **Poll SCM** under **Build Triggers** to schedule GitHub polling for every `15` minute interval. Type the following line of code `H/15 * * * *` in the **Schedule** textbox, as shown in the following screenshot. For testing purposes, you can reduce the polling interval:



6. Scroll down the screen a little further, and select the **Add build step** button under **Build**. In the drop-down list, select **Execute shell** and type in the text, as shown in the following screenshot:

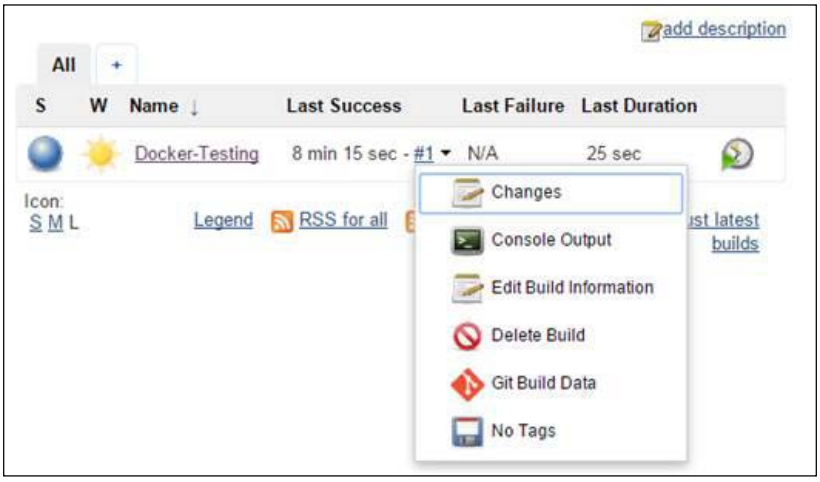


7. Finally, save the configuration by clicking on the **Save** button.

2. Go back to the Jenkins Dashboard, and you can find your test listed on the dashboard:

3. You can either wait for the Jenkins schedule to kick-start the build, or you can click on the clock icon on the right-hand side of the screen to kick-start the build immediately. As soon as the build is done, the Dashboard is updated with the build status as a success or failure, and the build number:

4. If you hover the mouse closer to the build number, you will get a drop-down button with options, such as **Changes**, **Console Output**, and so on, as shown in the following screenshot:



5. The **Console Output** option will show the details highlighted for the build, as follows:

```
Started by user anonymous
Building in workspace /var/lib/jenkins/jobs/Docker-Testing/
workspace
Cloning the remote Git repository
Cloning repository https://github.com/thedocker/testing/
. . . OUTPUT TRUNCATED . . .
+ docker build -t docker_testing_using_jenkins .
Sending build context to Docker daemon 121.9 kB

Sending build context to Docker daemon
Step 0 : FROM python:latest
. . . OUTPUT TRUNCATED . . .
Successfully built ad4be4b451e6
+ docker run --rm docker_testing_using_jenkins
.
----------------------------------------------------------------------
----
Ran 1 test in 0.000s

OK
Finished: SUCCESS
```

6. Evidently, the test failed because of the wrong module name **error_hitcount**, which we deliberately introduced. Now, let's experiment a negative scenario by deliberately introducing a bug in t**est_hitcount.py** and observe the effect on Jenkins build. As we have configured Jenkins, it faithfully polls the GitHub and kick-starts the build:

Apparently, the build failed as we expected.

7. As a final step, open **Console Output** of the failed build:

```
Started by an SCM change
Building in workspace /var/lib/jenkins/jobs/Docker-Testing/
workspace
. . . OUTPUT TRUNCATED . . .
ImportError: No module named 'error_hitcount'



----------------------------------------------------------------------
----
Ran 1 test in 0.001s

FAILED (errors=1)
Build step 'Execute shell' marked build as failure
Finished: FAILURE
```

Evidently, the test failed because of the wrong module name `error_hitcount`, which we deliberately introduced.

Cool, isn't it? We automated our testing using Jenkins and Docker. Besides, we are able to experience the power of testing automation using Jenkins and Docker. In a large-scale project, Jenkins and Docker can be combined together to automate the complete unit testing needs, and thus, to automatically capture any defects and deficiencies introduced by any developers.

Summary of Module 1 Chapter 12

The potential benefits of containerization are being discovered across the breadth and the length of software engineering. Previously, testing sophisticated software systems involved a number of expensive and hard-to-manage server modules and clusters. Considering the costs and complexities involved, most of the software testing is accomplished using mocking procedures and stubs. All of this is going to end for good with the maturity of the Docker technology. The openness and flexibility of Docker enables it to work seamlessly with other technologies to substantially reduce the testing time and complexity.

Ankita Thakur

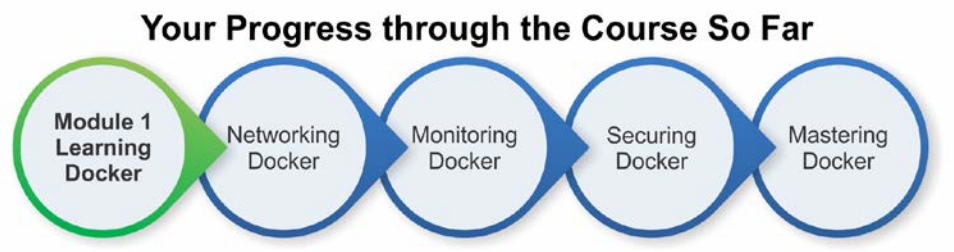
Your Course Guide

For a long time, the leading ways of testing software systems included mocking, dependency, injection, and so on. Usually, these mandate creating many sophisticated abstractions in the code. The current practice for developing and running test cases against an application is actually done on stubs rather than on the full application. That is, with a containerized workflow, it is very much possible to test against real application containers with all the dependencies. The contributions of the Docker paradigm, especially for the testing phenomenon and phase are therefore being carefully expounded and recorded in the recent past. Precisely speaking, the field of software engineering is moving towards smart and sunnier days with In this chapter, we clearly expounded and explained a powerful testing framework for integrated applications using the Docker-inspired containerization paradigm. Increasingly for the agile world, the proven and potential TDD method is being insisted as an efficient software building and sustenance methodology. This chapter has utilized the Python unit test framework to illustrate how the TDD methodology is a pioneering tool for software engineering. The unit test framework is tweaked to be efficiently and elegantly containerized, and the Docker container is seamlessly integrated with Jenkins, which is a modern day deployment tool for continuous delivery, and is part and parcel of the agile programming world, as described in this chapter.

Ankita Thakur

Your Course Guide

The Docker container source code is pre-checked before it enters into the GitHub code repository. The Jenkins tool downloads the code from GitHub and runs the test inside a container. In the next chapter, we shall will dive deep into, and describe the theoretical aspects of, the process isolation through the container technology and various debugging tools and techniques all the innovations in the Docker space.



13
Debugging Containers

Debugging has been an artistic component in the field of software engineering. All kinds of software building blocks individually, as well as collectively, need to go through a stream of deeper and decisive investigations by software development and testing professionals in order to ensure the security and safety of the resulting software applications. As Docker containers are said to be key runtime environments for next generation mission-critical software workloads, it is pertinent and paramount for containers, crafters, and composers to embark on the systematic and sagacious verification and validation of containers.

This chapter has been dedicatedly written to enable technical guys with all the right and relevant information to meticulously debug applications running inside containers and containers themselves. In this chapter, we will take a look at the theoretical aspects of process isolation for processes running as containers. A Docker container runs at a user-level process on host machines and typically has the same isolation level as provided by the operating system. With the release of Docker 1.5, many debugging tools are available, which can be efficiently used to debug your applications. We will also cover the primary Docker debugging tools, such as Docker `exec`, `stats`, `ps`, `top`, `events`, and `logs`. Finally, we will take a look at the `nsenter` tool to log in to containers without running the **Secure Shell** (**SSH**) daemon.

The list of topics that will be covered in the chapter is as follows:

- Process level isolation for Docker containers
- Debugging a containerized application
- Installing and using `nsenter`

Process level isolation for Docker containers

In the virtualization paradigm, the hypervisor emulates computing resources and provides a virtualized environment called a VM to install the operating system and applications on top of it. Whereas, in the case of the container paradigm, a single system (bare metal or virtual machine) is effectively partitioned to run multiple services simultaneously without interfering with each other. These services must be isolated from each other in order to prevent them from stepping on each other's resources or dependency conflict (also known as dependency hell). The Docker container technology essentially achieves process-level isolation by leveraging the Linux kernel constructs, such as namespaces and cgroups, particularly, the namespaces. The Linux kernel provides the following five powerful namespace leavers to isolate the global system resources from each other. These are the **Interprocess Communication** (**IPC**) namespaces used to isolate the interprocess communication resources:

- The network namespace is used to isolate networking resources, such as the network devices, network stack, port number, and so on
- The mount namespace isolates the filesystem mount points
- The PID namespace isolates the process identification number
- The user namespace is used to isolate the user ID and group ID
- The UTS namespace is used to isolate the hostname and the NIS domain name

These namespaces add an additional level of complexity when we have to debug the services running inside the containers, which we will learn more in detail in the next chapter.

In this section, we will discuss how the Docker engine provides process isolation by leveraging the Linux namespaces through a series of practical examples, and one of them is listed here:

1. Start by launching an `ubuntu` container in an interactive mode using the `docker run` subcommand, as shown here:

```
$ sudo docker run -it --rm ubuntu /bin/bash
root@93f5d72c2f21:/#
```

2. Proceed to find the process ID of the preceding container `93f5d72c2f21`, using the `docker inspect` subcommand in a different terminal:

```
$ sudo docker inspect \
         --format "{{ .State.Pid }}" 93f5d72c2f21
2543
```

   Apparently, from the preceding output, the process ID of the container `93f5d72c2f21` is `2543`.

3. Having got the process ID of the container, let's continue to see how the process associated with the container looks in the Docker host, using the `ps` command:

```
$ ps -fp 2543
UID         PID  PPID  C STIME TTY          TIME CMD
root       2543  6810  0 13:46 pts/7    00:00:00 /bin/bash
```

   Amazing, isn't it? We launched a container with `/bin/bash` as its command, and we have the `/bin/bash` process in the Docker host as well.

4. Let's go one step further and display the `/proc/2543/environ` file in the Docker host using the `cat` command:

```
$ sudo cat -v /proc/2543/environ
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
bin^@HOSTNAME=93f5d72c2f21^@TERM=xterm^@HOME=/root^@$
```

   In the preceding output, `HOSTNAME=93f5d72c2f21` stands out from the other environment variables because `93f5d72c2f21` is the container ID, as well as the hostname of the container, which we launched previously.

5. Now, let's get back to the terminal, where we are running our interactive container `93f5d72c2f21`, and list all the processes running inside this container using the `ps` command:

```
root@93f5d72c2f21:/# ps -ef
UID    PID PPID C STIME TTY     TIME CMD
root     1   0 0 18:46 ?    00:00:00 /bin/bash
root    15   1 0 19:30 ?    00:00:00 ps -ef
```

   Surprising, isn't it? Inside the container, the process ID of the `bin/bash` process is `1`, whereas outside the container, in the Docker host, the process ID is `2543`. Besides, the **Parent Process ID** (**PPID**) is `0` (zero).

In the Linux world, every system has just one root process with the PID 1 and PPID 0, which is the root of the complete process tree of that system. The Docker framework cleverly leverages the Linux PID namespace to spin a completely new process tree; thus, the processes running inside a container have no access to the parent process of the Docker host. However, the Docker host has a complete view of the child PID namespace spun by the Docker engine.

The network namespace ensures that all containers have independent network interfaces on the host machine. Also, each container has its own loopback interface. Each container talks to the outside world using its own network interface. You will be surprised to know that the namespace not only has its own routing table, but also has its own iptables, chains, and rules. The author of this chapter is running three containers on his host machine. Here, it is natural to expect three network interfaces for each container. Let's run the `docker ps` command:

```
$ sudo docker ps
41668be6e513        docker-apache2:latest   "/bin/sh -c 'apachec
069e73d4f63c        nginx:latest            "nginx -g '
871da6a6cf43        ubuntu:14.04            "/bin/bash"
```

So here, three are three interfaces, one for each container. Let's get their details by running the following command:

```
$ ifconfig
veth2d99bd3 Link encap:Ethernet  HWaddr 42:b2:cc:a5:d8:f3
          inet6 addr: fe80::40b2:ccff:fea5:d8f3/64 Scope:Link
          UP BROADCAST RUNNING  MTU:9001  Metric:1
veth422c684 Link encap:Ethernet  HWaddr 02:84:ab:68:42:bf
          inet6 addr: fe80::84:abff:fe68:42bf/64 Scope:Link
          UP BROADCAST RUNNING  MTU:9001  Metric:1
vethc359aec Link encap:Ethernet  HWaddr 06:be:35:47:0a:c4
          inet6 addr: fe80::4be:35ff:fe47:ac4/64 Scope:Link
          UP BROADCAST RUNNING  MTU:9001  Metric:1
```

The mount namespace ensures that the mounted filesystem is accessible only to the processes within the same namespace. The container A cannot see the mount points of the container B. If you want to check your mount points, you need to first log in to your container using the `exec` command (described in the next section), and then go to `/proc/mounts`:

```
root@871da6a6cf43:/# cat /proc/mounts
rootfs / rootfs rw 0 0/dev/mapper/docker-202:1-149807
871da6a6cf4320f625d5c96cc24f657b7b231fe89774e09fc771b3684bf405fb /
ext4 rw,relatime,discard,stripe=16,data=ordered 0 0 proc /proc proc
rw,nosuid,nodev,noexec,relatime 0 0
```

Let's run a container with a mount point that runs as the **Storage Area Network** (**SAN**) or **Network Attached Storage** (**NAS**) device and access it by logging into the container. This is given to you as an exercise. I have implemented this in one of my projects at work.

There are other namespaces that these containers/processes can be isolated into, namely, user, IPC, and UTS. The user namespace allows you to have root privileges within the namespace without giving that particular access to processes outside the namespace. Isolating a process with the IPC namespace gives it its own interprocess communication resources, for example, System V IPC and POSIX messages. The UTS namespace isolates the *hostname* of the system.

Docker has implemented this namespace using the `clone` system call. On the host machine, you can inspect the namespace created by Docker for the container (with `pid 3728`):

```
$ sudo ls /proc/3728/ns/
ipc  mnt  net  pid  user  uts
```

In most industrial deployments of Docker, people are extensively using patched Linux kernels to provide specific needs. Also, a few companies have patched their kernels to attach arbitrary processes to the existing namespaces because they feel that this is the most convenient and reliable way to deploy, control, and orchestrate containers.

Control groups

Linux containers rely on control groups (cgroups), which not only track groups of processes, but also expose metrics of the CPU, memory, and block I/O usage. You can access these metrics and obtain network usage metrics as well. Control groups are another important components of Linux containers. Control groups are around for a while and are initially merged in the Linux kernel code 2.6.24. They ensure that each Docker container will get a fixed amount of memory, CPU, and disk I/O, so that any container will not able to bring the host machine down at any point of time under any circumstances. Control groups do not play a role in preventing one container from being accessed, but they are essential to fending off some **Denial of Service** (**DoS**) attacks.

On Ubuntu 14.04, `cgroup` is implemented in the `/sys/fs/cgroup` path. The memory information of Docker is available at the `/sys/fs/cgroup/memory/docker/` path.

Similarly, the CPU details are made available in the `/sys/fs/cgroup/cpu/docker/` path.

Let's find out the maximum limit of memory that can be consumed by the container (`41668be6e513e845150abd2dd95dd574591912a7fda947f6744a0bfdb5cd9a85`).

For this, you can go to the `cgroup` memory path and check for the `memory.max.usage` file:

```
/sys/fs/cgroup/memory/docker/41668be6e513e845150abd2dd95dd574591912a7
fda947f6744a0bfdb5cd9a85
```

```
$ cat memory.max_usage_in_bytes
```

```
13824000
```

So, by default, any container can use up to 13.18 MB memory only.

Similarly, CPU parameters can be found in the following path:

```
/sys/fs/cgroup/cpu/docker/41668be6e513e845150abd2dd95dd574591912a7fda
947f6744a0bfdb5cd9a85
```

Traditionally, Docker runs only one process inside the containers.
So typically, you have seen people running three containers each for PHP, nginx, and MySQL. However, this is a myth. You can run all your three processes inside a single container.

Docker isolates many aspects of the underlying host from an application running in a container without the root privileges. However, this separation is not as strong as that of virtual machines, which run independent OS instances on top of a hypervisor without sharing the kernel with the underlying OS. It's not a good idea to run applications with different security profiles as containers on the same host, but there are security benefits to encapsulate different applications into containerized applications that would otherwise run directly on the same host.

Debugging a containerized application

Computer programs (software) sometimes fail to behave as expected. This is due to faulty code or due to the environmental changes between the development, testing, and deployment systems. Docker container technology eliminates the environmental issues between development, testing, and deployment as much as possible by containerizing all the application dependencies. Nonetheless, there could be still anomalies due to faulty code or variations in the kernel behavior, which needs debugging. Debugging is one of the most complex processes in the software engineering world and it becomes much more complex in the container paradigm because of the isolation techniques. In this section, we are going to learn a few tips and tricks to debug a containerized application using the tools native to Docker, as well as the tools provided by external sources.

Initially, many people in the Docker community individually developed their own debugging tools, but later Docker started supporting native tools, such as `exec`, `top`, `logs`, `events`, and many more. In this section, we will dive deep into the following Docker tools:

- `exec`
- `ps`
- `top`
- `stats`
- `events`
- `logs`

The Docker exec command

The `docker exec` command provided the much-needed help to users, who are deploying their own web servers or other applications running in the background. Now, it is not necessary to log in to run the SSH daemon in the container.

First, run the `docker ps -a` command to get the container ID:

```
$ sudo docker ps -a
b34019e5b5ee        nsinit:latest              "make local"
a245253db38b        training/webapp:latest     "python app.py"
```

Then, run the `docker exec` command to log in to the container.

```
$ sudo docker exec -it a245253db38b bash
root@a245253db38b:/opt/webapp#
```

It is important to note that the `docker exec` command can only access the running containers, so if the container stops functioning, then you need to restart the stopped container in order to proceed. The `docker exec` command spawns a new process in the target container using the Docker API and CLI. So if you run the `pe -aef` command inside the target container, it looks like this:

```
# ps -aef
UID         PID  PPID  C STIME TTY          TIME CMD
root          1     0  0 Mar22 ?        00:00:53 python app.py
root         45     0  0 18:11 ?        00:00:00 bash
root         53    45  0 18:11 ?        00:00:00 ps -aef
```

Here, `python app.y` is the application that is already running in the target container, and the `docker exec` command has added the `bash` process inside the container. If you run `kill -9 pid(45)`, you will be automatically logged out of the container.

If you are an enthusiastic developer, and you want to enhance the `exec` functionality, you can refer to `https://github.com/chris-rock/docker-exec`.

It is recommended that you use the `docker exec` command only for monitoring and diagnostic purposes, and I personally believe in the concept of one process per container, which is one of the best practices widely accentuated.

The Docker ps command

The `docker ps` command, which is available inside the container, is used to see the status of the process. This is similar to the standard `ps` command in the Linux environment and is *not* a `docker ps` command that we run on the Docker host machine.

This command runs inside the Docker container:

```
root@5562f2f29417:/# ps -s
  UID   PID   PENDING   BLOCKED   IGNORED    CAUGHT STAT TTY
TIME COMMAND
    0     1  00000000  00010000  00380004  4b817efb Ss   ?
0:00 /bin/bash
    0    33  00000000  00000000  00000000  73d3fef9 R+   ?
0:00 ps -s
root@5562f2f29417:/# ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  80   0 -  4541 wait   ?        00:00:00
bash
0 R     0    34     1  0  80   0 -  1783 -      ?        00:00:00 ps
root@5562f2f29417:/# ps -t
  PID TTY      STAT   TIME COMMAND
    1 ?        Ss     0:00 /bin/bash
   35 ?        R+     0:00 ps -t
root@5562f2f29417:/# ps -m
  PID TTY          TIME CMD
    1 ?        00:00:00 bash
    - -        00:00:00 -
   36 ?        00:00:00 ps
    - -        00:00:00 -
root@5562f2f29417:/# ps -a
  PID TTY          TIME CMD
   37 ?        00:00:00 ps
```

Use `ps --help <simple|list|output|threads|misc|all>` or `ps --help <s|l|o|t|m|a>` for additional help text.

The Docker top command

You can run the `top` command from the Docker host machine using the following command:

```
docker top [OPTIONS] CONTAINER [ps OPTIONS]
```

This gives a list of the running processes of a container without logging into the container, as follows:

```
$ sudo docker top  a245253db38b
UID                 PID                 PPID                C
STIME               TTY                 TIME                CMD
root                5232                3585                0
Mar22               ?                   00:00:53            python
app.py
$ sudo docker top  a245253db38b  -aef
UID                 PID                 PPID                C
STIME               TTY                 TIME                CMD
root                5232                3585                0
Mar22               ?                   00:00:53            python
app.py
```

The Docker `top` command provides information about the CPU, memory, and swap usage if you run it inside a Docker container:

```
root@a245253db38b:/opt/webapp# top
top - 19:35:03 up 25 days, 15:50,  0 users,  load average: 0.00,
0.01, 0.05
Tasks:   3 total,   1 running,   2 sleeping,   0 stopped,   0 zombie
```

```
Cpu(s):  0.0%us,  0.0%sy,  0.0%ni, 99.9%id,  0.0%wa,  0.0%hi,
0.0%si,  0.0%st
Mem:   1016292k total,   789812k used,   226480k free,    83280k
buffers
Swap:        0k total,        0k used,        0k free,   521972k
cached
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
    1 root      20   0 44780  10m 1280 S  0.0  1.1   0:53.69 python
   62 root      20   0 18040 1944 1492 S  0.0  0.2   0:00.01 bash
   77 root      20   0 17208 1164  948 R  0.0  0.1   0:00.00 top
```

In case you get the error `error - TERM environment variable not set` while running the `top` command inside the container, perform the following steps to resolve it:

Run the `echo $TERM` command. You will get the result as `dumb`. Then, run the following command:

```
$ export TERM=dumb
```

This will resolve the error.

The Docker stats command

The Docker `stats` command provides you with the capability to view the memory, CPU, and the network usage of a container from a Docker host machine, as illustrated here:

```
$ sudo docker stats a245253db38b
CONTAINER           CPU %               MEM USAGE/LIMIT       MEM %
  NET I/O
a245253db38b        0.02%               16.37 MiB/992.5 MiB   1.65%
  3.818 KiB/2.43 KiB
```

You can run the `stats` command to also view the usage for multiple containers:

```
$ sudo docker stats a245253db38b f71b26cee2f1
```

In the latest release of Docker 1.5, Docker provides you access to container statistics *read only* parameters. This will streamline the CPU, memory, network IO, and block IO of your containers. This helps you choose the resource limits and also in profiling. The Docker stats utility provides you with these resource usage details only for running containers. You can get detailed information using the end point APIs at `https://docs.docker.com/reference/api/docker_remote_api_v1.17/#inspect-a-container`.

Docker stats is originally taken from Michael Crosby's code contribution, which can be accessed at `https://github.com/crosbymichael`.

The Docker events command

Docker containers will report the following real-time events: `create`, `destroy`, `die`, `export`, `kill`, `omm`, `pause`, `restart`, `start`, `stop`, and `unpause`. The following are a few examples that illustrate how to use these commands:

```
$ sudo docker pause  a245253db38b
a245253db38b
$ sudo docker ps -a
a245253db38b        training/webapp:latest    "python app.py"
4 days ago          Up 4 days (Paused)       0.0.0.0:5000->5000/tcp
sad_sammet
$ sudo docker unpause  a245253db38b
a245253db38b
$ sudo docker ps -a
a245253db38b        training/webapp:latest    "python app.py"
4 days ago    Up 4 days        0.0.0.0:5000->5000/tcp   sad_sammet
```

The Docker image will also report the untag and delete events.

Using multiple filters will be handled as an `AND` operation; for example, `--filter container= a245253db38b --filter event=start` will display events for the container `a245253db38b` and the event type is start.

Currently, the supported filters are container, event, and image.

The Docker logs command

This command fetches the log of a container without logging into the container. It batch-retrieves logs present at the time of execution. These logs are the output of `STDOUT` and `STDERR`. The general usage is shown in `docker logs [OPTIONS] CONTAINER`.

The `-follow` option will continue to provide the output till the end, `-t` will provide the timestamp, and `--tail= <number of lines>` will show the number of lines of the log messages of your container:

```
$ sudo docker logs a245253db38b
 * Running on http://0.0.0.0:5000/
172.17.42.1 - - [22/Mar/2015 06:04:23] "GET / HTTP/1.1" 200 -
172.17.42.1 - - [24/Mar/2015 13:43:32] "GET / HTTP/1.1" 200 -
$
$ sudo docker logs -t a245253db38b
2015-03-22T05:03:16.866547111Z  * Running on http://0.0.0.0:5000/
2015-03-22T06:04:23.349691099Z 172.17.42.1 - - [22/Mar/2015 06:04:23]
"GET / HTTP/1.1" 200 -
2015-03-24T13:43:32.754295010Z 172.17.42.1 - - [24/Mar/2015 13:43:32]
"GET / HTTP/1.1" 200 -
```

We also used the `docker logs` utility in *Chapter 2*, *Handling Docker Containers* and *Chapter 7*, *Running Services in a Container*, to view the logs of our containers.

Installing and using nsenter

In any commercial Docker deployments, you may use various containers, such as web application, database, and so on. However, you need to access these containers in order to modify the configuration or debug/troubleshoot the issues. A simple solution for this problem is to run an SSH server in each container. It is a not good way to access the machines due to unanticipated security implications. However, if you are working in any one of the world-class IT companies, such as IBM, DELL, HP, and so on, your security compliance guy will never allow you to use SSH to connect to machines.

So, here is the solution. The `nsenter` tool provides you access to log in to your container. Note that `nsenter` will be first deployed as a Docker container only. Using this deployed `nsenter`, you can access your container. Follow these steps:

1. Let's run a simple web application as a container:

   ```
   $ sudo docker run -d -p 5000:5000 training/webapp python
   app.py
   ------------------------
   a245253db38b626b8ac4a05575aa704374d0a3c25a392e0f4f562df92bb98d
   74
   ```

2. Test the web container:

```
$ curl localhost:5000
Hello world!
```

3. Install `nsenter` and run it as a container:

```
$ sudo docker run -v /usr/local/bin:/target jpetazzo/nsenter
```

   Now, `nsenter` is up and running as a container.

4. Use the nsenter container to log in to the container (`a245253db38b`), that we created in step 1.

   Run the following command to get the `PID` value:

```
$ PID=$(sudo docker inspect --format {{.State.Pid}}
a245253db38b)
```

5. Now, access the web container:

```
$ sudo nsenter --target $PID --mount --uts --ipc --net --pid
root@a245253db38b:/#
```

   Then, you can log in and start accessing your container. Accessing your container in this way will make your security and compliance professionals happy, and they will feel relaxed.

Since Docker 1.3, the Docker exec is a supported tool used for logging into containers.

The `nsenter` tool doesn't enter cgroups and therefore evades resource limitations. The potential benefit of this would be debugging and external audit, but for a remote access, `docker exec` is the current recommended approach.

The `nsenter` tool is tested only on Intel 64-bit platforms. You cannot run `nsenter` inside the container that you want to access, and hence, you need to run `nsenter` on host machines only. By running `nsenter` on a host machine, you can access all of the containers of that host machine. Also, you cannot use the running `nsenter` on a particular host, say host A to access the containers on host B.

Your Coding Challenge

Ankita Thakur

Your Course Guide

Let's now test what we've learned so far:

- What does the following commands do?
  - exec
  - ps
  - top
  - stats
  - events
  - logs

Summary of Module 1 Chapter 13

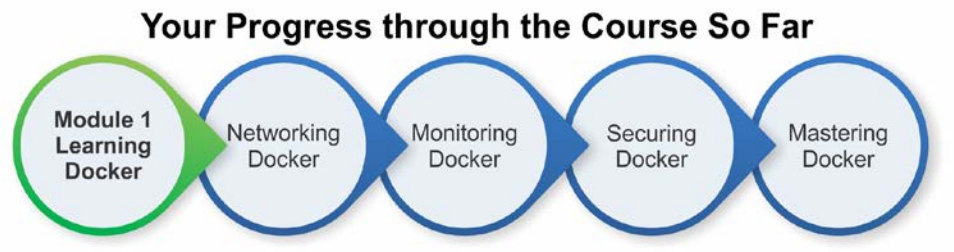
Ankita Thakur

Your Course Guide

Docker provides you with the isolation of containers using the Linux container technology, such as LXC and now libcontainer. Libcontainer is Docker's own implementation in the Go programming language to access the kernel namespace and control groups. This namespace is used for process-level isolation, while control groups is used for restricting the resource usage of running containers. Since the containers run as independent processes directly over the Linux kernel, the generally available (GA) debugging tools are not fit enough to work inside the containers to debug the containerized processes. Docker now provides you with a rich set of tools to effectively debug the container, as well as processes inside the container itself. Docker exec will allow you to log in to the container without running an SSH daemon in the container.

Docker stats provides information about the container's memory and CPU usage. Docker events reports the events, such as create, destroy, kill, and so on. Similarly, Docker logs fetch the logs from the container without logging into the container.

Debugging is the foundation that can be used to strategize other security vulnerabilities and holes. The next chapter is therefore incorporated to expound the plausible security threats of Docker containers and how they can be subdued with a variety of security approaches, automated tools, best practices, key guidelines, and metrics.


Your Progress through the Course So Far
Module 1 Learning Docker
Networking Docker
Monitoring Docker
Securing Docker
Mastering Docker

Course Module 2
Networking Docker

Course Module 1: Learning Docker
Chapter 1: Getting Started with Docker
Chapter 2: Up and Running
Chapter 3: Container Image Storage
Chapter 4: Working with Docker Containers and Images
Chapter 5: Publishing Images
Chapter 6: Running Your Private Docker Infrastructure
Chapter 7: Running Services in a Container
Chapter 8: Sharing Data with Containers
Chapter 9: Docker Machine
Chapter 10: Orchestrating Docker
Chapter 11: Docker Swarm
Chapter 12: Testing with Docker
Chapter 13: Debugging Containers

Course Module 2: Networking Docker
Chapter 1: Docker Networking Primer
Chapter 2: Docker Networking Internals
Chapter 3: Building Your First Docker Network
Chapter 4: Networking in a Docker Cluster
Chapter 5: Next Generation Networking Stack for Docker – libnetwork

Course Module 3: Monitoring Docker
Chapter 1: Introduction to Docker Monitoring
Chapter 2: Using the Built-in Tools
Chapter 3: Advanced Container Resource Analysis
Chapter 4: A Traditional Approach to Monitoring Containers
Chapter 5: Querying with Sysdig
Chapter 6: Exploring Third Party Options
Chapter 7: Collecting Application Logs from within the Container
Chapter 8: What Are the Next Steps?

Course Module 2
Networking Docker

Course Module 4: Securing Docker
Chapter 1: Securing Docker Hosts
Chapter 2: Securing Docker Components
Chapter 3: Securing and Hardening Linux Kernels
Chapter 4: Docker Bench for Security
Chapter 5: Monitoring and Reporting Docker Security Incidents
Chapter 6: Using Docker's Built-in Security Features
Chapter 7: Securing Docker with Third-Party Tools
Chapter 8: Keeping up Security

Course Module 5: Mastering Docker
Chapter 1: Docker in Production
Chapter 2: Shipyard
Chapter 3: Panamax
Chapter 4: Tutum
Chapter 5: Advanced Docker
A Final Run-Through
Reflect and Test Yourself! Answers

Course Module 2

The first module covered a lot of fundamentals and basic topics of Docker. I hope now you're comfortable and confident with Docker and ready to explore more. Aren't you?

Docker is a Linux container implementation that enables the creation of light-weight portable development and production-quality environments. These environments can be updated incrementally. Docker achieves this by leveraging containment principles, such as cgroups and Linux namespaces, along with overlay filesystem-based portable images.

This module aggregates all the latest Docker networking technology and provides great in depth explanation. It will help you learn, create, deploy, and provide administration steps for Docker networking.

Ankita Thakur

Your Course Guide

We'll start with the basics of networking and see how Docker networking works. Then, we'll expose the strengths and weaknesses of the current Docker network implementation and third party landscape. This module helps you understand Docker networking spanning multiple containers over multiple hosts through practical examples. Later on, you'll observe the pitfalls of Docker networking and how to overcome them. You'll learn how Docker networking works for Docker Swarm and Kubernetes. Another important topic that the module will teach you is how to configure networking using Docker's **container network model (CNM)**. Finally, you'll explore OpenvSwitch to connect containers.

So, if you are a Linux administrator who wants to learn Docker networking to ensure the efficient administration of core elements and applications or a DevOps professional who wants to understand inner workings and deployment models and options in connecting containers to each other and to the external world and learn how to leverage this knowledge in building a more secure and portable container based application or service, you'll gain a lot of insights from this module.

1
Docker Networking Primer

Docker is a lightweight container technology that has gathered enormous interest in recent years. It neatly bundles various Linux kernel features and services, such as namespaces, cgroups, SELinux, and AppArmor profiles, over union filesystems such as AUFS and BTRFS in order to make modular images. These images provide a highly configurable virtualized environment for applications and follow a **write once, run anywhere** workflow. An application can be composed of a single process running in a Dcker container or it could be made up of multiple processes running in their own containers and being replicated as the load increases. Therefore, there is a need for powerful networking elements that can support various complex use cases.

In this chapter, you will learn about the essential components of Docker networking and how to build and run simple container examples.

This chapter covers the following topics:

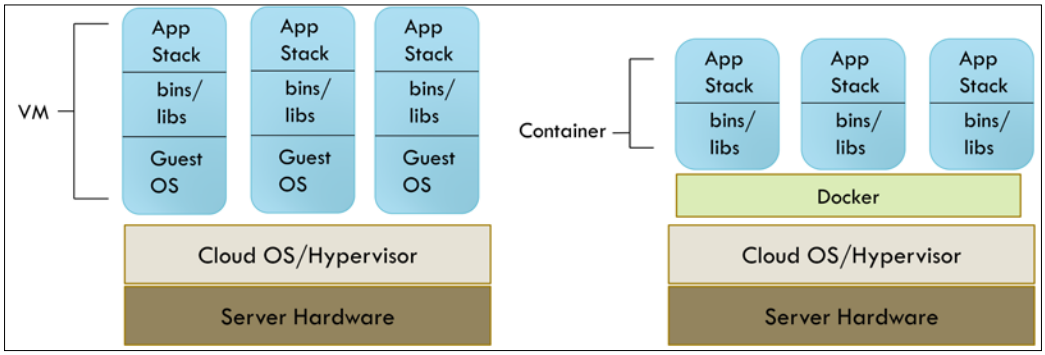
- Networking and Docker
- The `docker0` bridge networking
- Docker OVS networking
- Unix domain networks
- Linking Docker containers
- What's new in Docker networking

Docker is getting a lot of traction in the industry because of its performance-savvy and universal replicability architecture, while providing the following four cornerstones of modern application development:

- Autonomy
- Decentralization
- Parallelism
- Isolation

Furthermore, wide-scale adoption of Thoughtworks's microservices architecture, or **LOSA** (**Lots of Small Applications**), is further bringing potential to Docker technology. As a result, big companies such as Google, VMware, and Microsoft have already ported Docker to their infrastructure, and the momentum is continued by the launch of myriad Docker start-ups, namely Tutum, Flocker, Giantswarm, and so on.

Since Docker containers replicate their behavior anywhere, be it your development machine, a bare metal server, virtual machine, or data center, application designers can focus their attention on development, while operational semantics are left with DevOps. This makes team workflow modular, efficient, and productive. Docker is not to be confused with a **virtual machine** (**VM**), even though they are both virtualization technologies. While Docker shares an OS with providing a sufficient level of isolation and security to applications running in containers, it later completely abstracts away the OS and gives strong isolation and security guarantees. However, Docker's resource footprint is minuscule in comparison to a VM and hence preferred for economy and performance. However, it still cannot completely replace VMs and is therefore complementary to VM technology. The following diagram shows the architecture of VMs and Docker:



Networking and Docker

Each Docker container has its own network stack, and this is due to the Linux kernel NET namespace, where a new NET namespace for each container is instantiated and cannot be seen from outside the container or from other containers.

Docker networking is powered by the following network components and services.

Linux bridges

These are L2/MAC learning switches built into the kernel and are to be used for forwarding.

Open vSwitch

This is an advanced bridge that is programmable and supports tunneling.

NAT

Network address translators are immediate entities that translate IP addresses and ports (SNAT, DNAT, and so on).

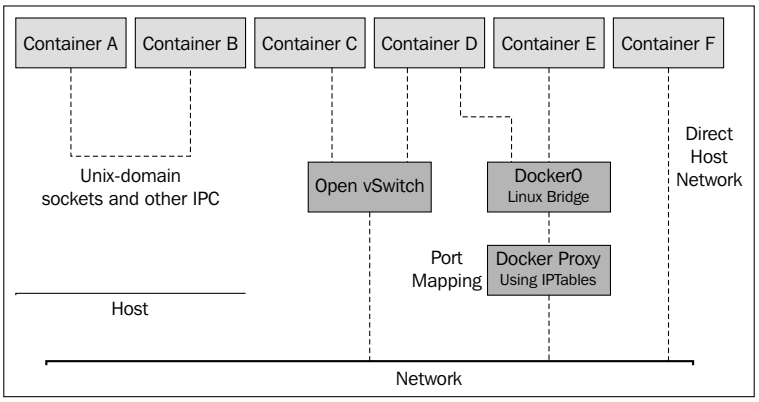
IPtables

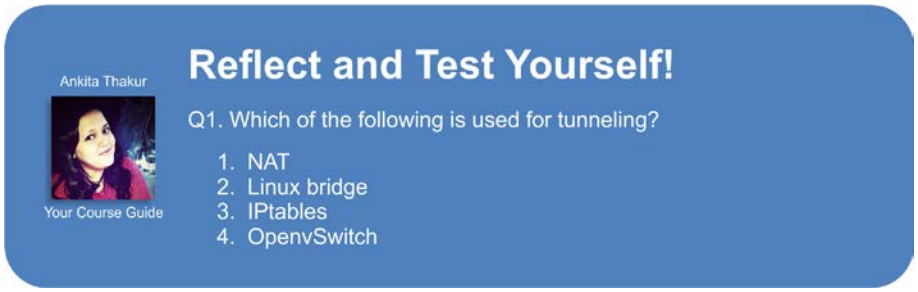
This is a policy engine in the kernel used for managing packet forwarding, firewall, and NAT features.

AppArmor/SELinux

Firewall policies for each application can be defined with these.

Various networking components can be used to work with Docker, providing new ways to access and use Docker-based services. As a result, we see a lot of libraries that follow a different approach to networking. Some of the prominent ones are Docker Compose, Weave, Kubernetes, Pipework, libnetwork, and so on. The following figure depicts the root ideas of Docker networking:





The docker0 bridge

The `docker0` bridge is the heart of default networking. When the Docker service is started, a Linux bridge is created on the host machine. The interfaces on the containers talk to the bridge, and the bridge proxies to the external world. Multiple containers on the same host can talk to each other through the Linux bridge.

`docker0` can be configured via the `--net` flag and has, in general, four modes:

- `--net default`
- `--net=none`
- `--net=container:$container2`
- `--net=host`

The --net default mode

In this mode, the default bridge is used as the bridge for containers to connect to each other.

The --net=none mode

With this mode, the container created is truly isolated and cannot connect to the network.

The --net=container:$container2 mode

With this flag, the container created shares its network namespace with the container called `$container2`.

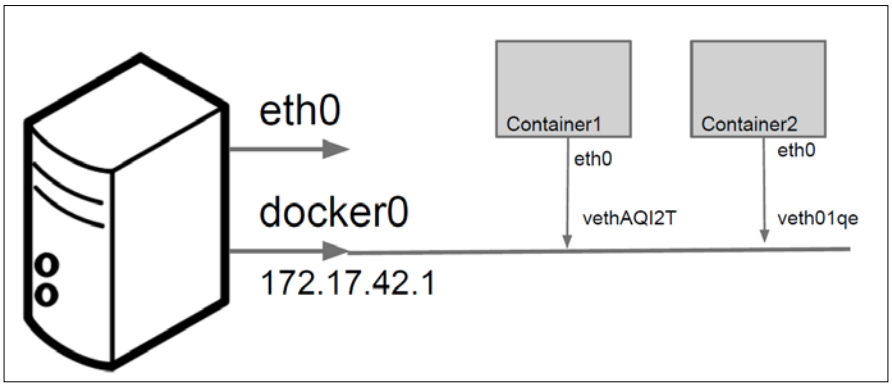
The --net=host mode

With this mode, the container created shares its network namespace with the host.

Port mapping in Docker container

In this section, we look at how container ports are mapped to host ports. This mapping can either be done implicitly by Docker Engine or can be specified.

If we create two containers called **Container1** and **Container2**, both of them are assigned an IP address from a private IP address space and also connected to the **docker0** bridge, as shown in the following figure:



Both the preceding containers will be able to ping each other as well as reach the external world.

For external access, their port will be mapped to a host port.

As mentioned in the previous section, containers use network namespaces. When the first container is created, a new network namespace is created for the container. A vEthernet link is created between the container and the Linux bridge. Traffic sent from `eth0` of the container reaches the bridge through the vEthernet interface and gets switched thereafter. The following code can be used to show a list of Linux bridges:

```
# show linux bridges
$ sudo brctl show
```

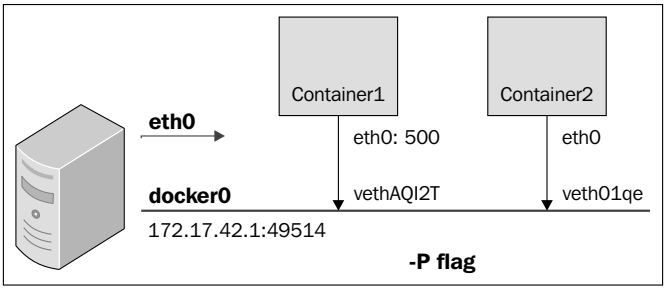
The output will be similar to the one shown as follows, with a bridge name and the `veth` interfaces on the containers it is mapped to:

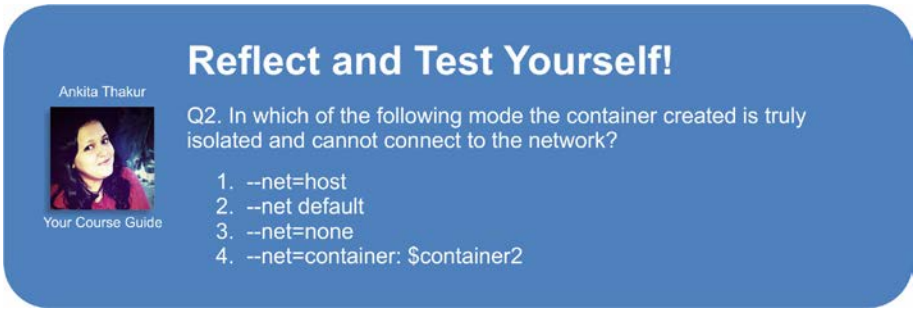
```
bridge name      bridge id         STP enabled     interfaces
docker0       8000.56847afe9799         no          veth44cb727
                                                    veth98c3700
```

How does the container connect to the external world? The `iptables nat` table on the host is used to masquerade all external connections, as shown here:

```
$ sudo iptables -t nat -L -n
...
Chain POSTROUTING (policy ACCEPT) target prot opt
source destination MASQUERADE all -- 172.17.0.0/16
!172.17.0.0/16
...
```

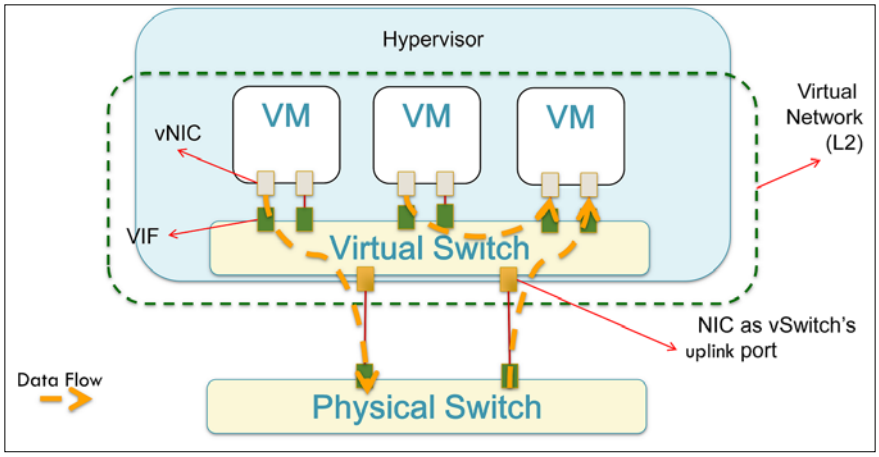
How to reach containers from the outside world? The port mapping is again done using the `iptables nat` option on the host machine.



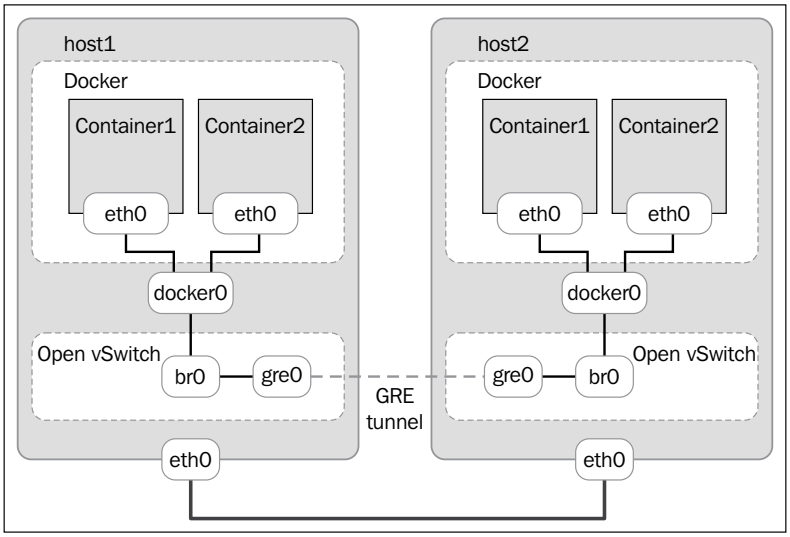


Docker OVS

Open vSwitch is a powerful network abstraction. The following figure shows how OVS interacts with the **VM**s, **Hypervisor**, and the **Physical Switch**. Every **VM** has a **vNIC** associated with it. Every **vNIC** is connected through a **VIF** (also called a **virtual interface**) with the **Virtual Switch**:



OVS uses tunnelling mechanisms such as GRE, VXLAN, or STT to create virtual overlays instead of using physical networking topologies and Ethernet components. The following figure shows how OVS can be configured for the containers to communicate between multiple hosts using GRE tunnels:



Unix domain socket

Within a single host, UNIX IPC mechanisms, especially UNIX domain sockets or pipes, can also be used to communicate between containers:

```
$  docker run  --name c1 -v /var/run/foo:/var/run/foo -d -I -t base
/bin/bash
$  docker run  --name c2 -v /var/run/foo:/var/run/foo -d -I -t base
/bin/bash
```

Apps on `c1` and `c2` can communicate over the following Unix socket address:

```
struct  sockaddr_un address;
address.sun_family = AF_UNIX;
snprintf(address.sun_path, UNIX_PATH_MAX, "/var/run/foo/bar" );
```

C1: Server.c	C2: Client.c

`bind(socket_fd, (struct sockaddr *) &address, sizeof(struct sockaddr_un));`
`listen(socket_fd, 5);`
`while((connection_fd = accept(socket_fd, (struct sockaddr *) &address, &address_length)) > -1)`
`nbytes = read(connection_fd, buffer, 256);`
	`connect(socket_fd, (struct sockaddr *) &address, sizeof(struct sockaddr_un));`
	`write(socket_fd, buffer, nbytes);`

Linking Docker containers

In this section, we introduce the concept of linking two containers. Docker creates a tunnel between the containers, which doesn't need to expose any ports externally on the container. It uses environment variables as one of the mechanisms for passing information from the parent container to the child container.

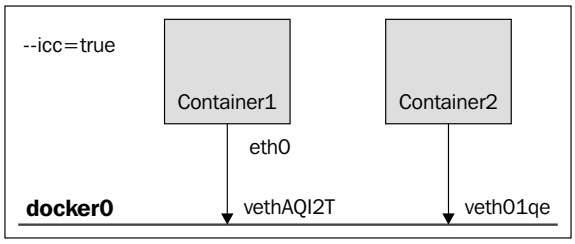
In addition to the environment variable `env`, Docker also adds a host entry for the source container to the `/etc/hosts` file. The following is an example of the host file:

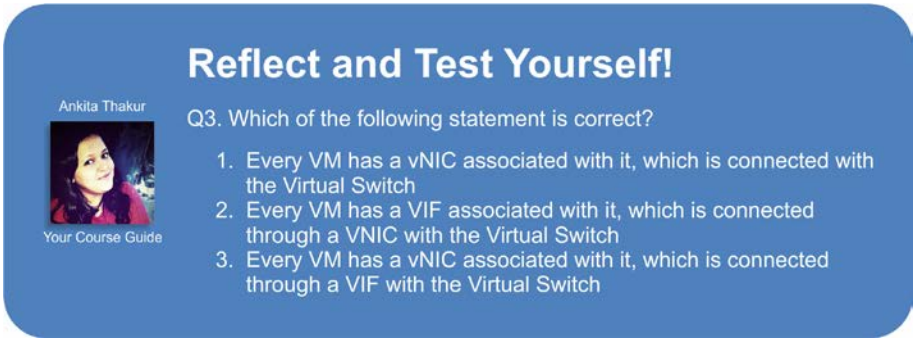
```
$ docker run -t -i --name c2 --rm --link c1:c1alias training/webapp
/bin/bash
root@<container_id>:/opt/webapp# cat /etc/hosts
172.17.0.1  aed84ee21bde
...
172.17.0.2  c1alaias 6e5cdeb2d300 c1
```

There are two entries:

- The first is an entry for the container `c2` that uses the Docker container ID as a host name
- The second entry, `172.17.0.2  c1alaias 6e5cdeb2d300 c1`, uses the `link` alias to reference the IP address of the `c1` container

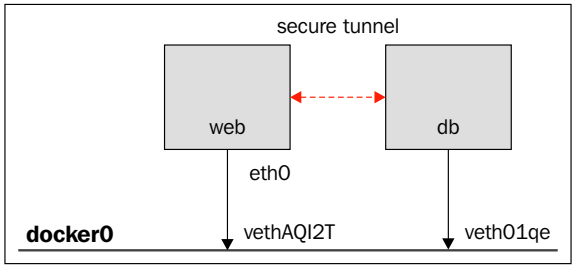
The following figure shows two containers **Container 1** and **Container 2** connected using veth pairs to the `docker0` bridge with `--icc=true`. This means these two containers can access each other through the bridge:





Links

Links provide service discovery for Docker. They allow containers to discover and securely communicate with each other by using the flag `-link name:alias`. Inter-container communication can be disabled with the daemon flag `-icc=false`. With this flag set to `false`, **Container 1** cannot access **Container 2** unless explicitly allowed via a link. This is a huge advantage for securing your containers. When two containers are linked together, Docker creates a parent-child relationship between them, as shown in the following figure:



From the outside, it looks like this:

```
# start the database
$  sudo docker run -dp 3306:3306 --name todomvcdb \
-v /data/mysql:/var/lib/mysql cpswan/todomvc.mysql


# start the app server
$  sudo docker run -dp 4567:4567 --name todomvcapp \
--link todomvcdb:db cpswan/todomvc.sinatra
```

On the inside, it looks like this:

```
$  dburl = ''mysql://root:pa55Word@'' + \
ENV[''DB_PORT_3306_TCP_ADDR'']
+ ''/todomvc''
$  DataMapper.setup(:default, dburl)
```

What's new in Docker networking?

Docker networking is at a very nascent stage, and there are many interesting contributions from the developer community, such as Pipework, Weave, Clocker, and Kubernetes. Each of them reflects a different aspect of Docker networking. We will learn about them in later chapters. Docker, Inc. has also established a new project where networking will be standardized. It is called **libnetwork**.

libnetwork implements the **container network model** (**CNM**), which formalizes the steps required to provide networking for containers while providing an abstraction that can be used to support multiple network drivers. The CNM is built on three main components—sandbox, endpoint, and network.

Sandbox

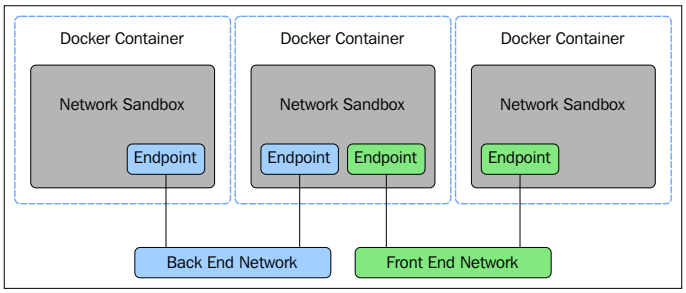
A sandbox contains the configuration of a container's network stack. This includes management of the container's interfaces, routing table, and DNS settings. An implementation of a sandbox could be a Linux network namespace, a FreeBSD jail, or other similar concept. A sandbox may contain many endpoints from multiple networks.

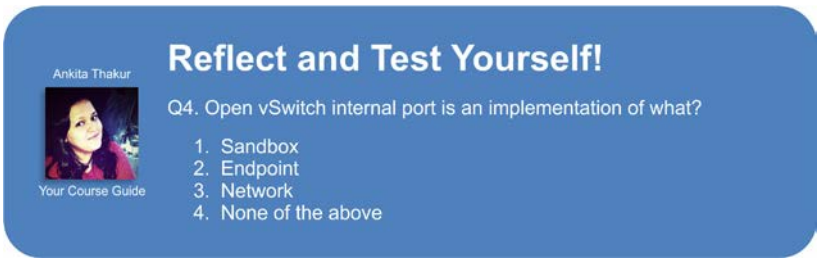
Endpoint

An endpoint connects a sandbox to a network. An implementation of an endpoint could be a veth pair, an Open vSwitch internal port, or something similar. An endpoint can belong to only one network but may only belong to one sandbox.

Network

A network is a group of endpoints that are able to communicate with each other directly. An implementation of a network could be a Linux bridge, a VLAN, and so on. Networks consist of many endpoints, as shown in the following diagram:



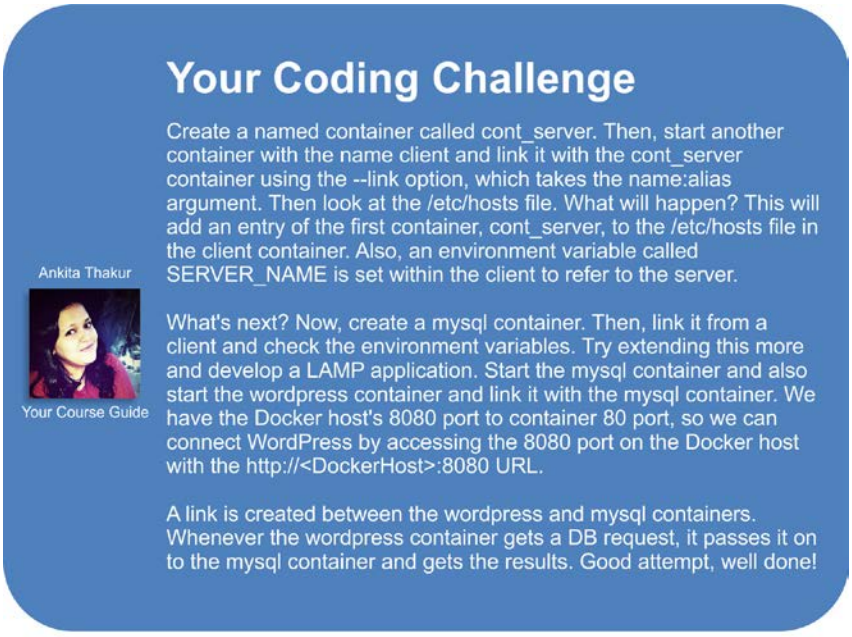


The Docker CNM model

The CNM provides the following contract between networks and containers:

- All containers on the same network can communicate freely with each other
- Multiple networks are the way to segment traffic between containers and should be supported by all drivers
- Multiple endpoints per container are the way to join a container to multiple networks
- An endpoint is added to a network sandbox to provide it with network connectivity

We will discuss the details of how CNM is implemented in *Chapter 6, Next Generation Networking Stack for Docker – libnetwork*.



Summary of Module 2 Chapter 1

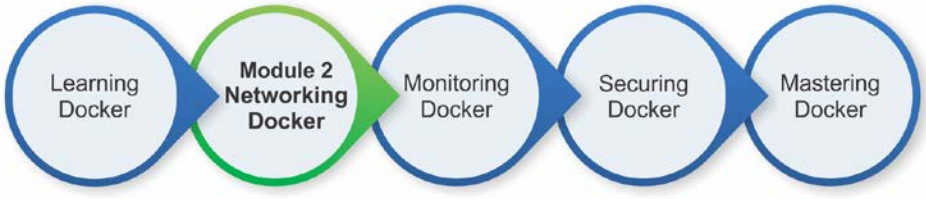
Ankita Thakur

Your Course Guide

In this chapter, we learned about the essential components of Docker networking, which have evolved from coupling simple Docker abstractions and powerful network components such as Linux bridges and Open vSwitch.

We learned how Docker containers can be created with various modes. In the default mode, port mapping helps through the use of iptables NAT rules, allowing traffic arriving at the host to reach containers. Later in the chapter, we covered the basic linking of containers. We also talked about the next generation of Docker networking, which is called libnetwork.

Your Progress through the Course So Far



2
Docker Networking Internals

This chapter discusses the semantics and syntax of Docker networking in detail, exposing strengths and weaknesses of the current Docker network paradigm.

It covers the following topics:

- Configuring the IP stack for Docker
    - IPv4 support
    - Issues with IPv4 address management
    - IPv6 support
- Configuring DNS
    - DNS basics
    - Multicast DNS
- Configuring the Docker bridge
- Overlay networks and underlay networks
    - What are they?
    - How does Docker use them?
    - What are some of their advantages?

Configuring the IP stack for Docker

Docker uses the IP stack to interact with the outside world using TCP or UDP. It supports the IPv4 and IPv6 addressing infrastructures, which are explained in the following subsections.

IPv4 support

By default, Docker provides IPv4 addresses to each container, which are attached to the default `docker0` bridge. The IP address range can be specified while starting the Docker daemon using the `--fixed-cidr` flag, as shown in the following code:

```
$ sudo docker –d --fixed-cidr=192.168.1.0/25
```

We will discuss more about this in the *Configuring the Docker bridge* section.

The Docker daemon can be listed on an IPv4 TCP endpoint in addition to a Unix socket:

```
$ sudo docker -H tcp://127.0.0.1:2375 -H unix:///var/run/docker.sock
-d &
```

IPv6 support

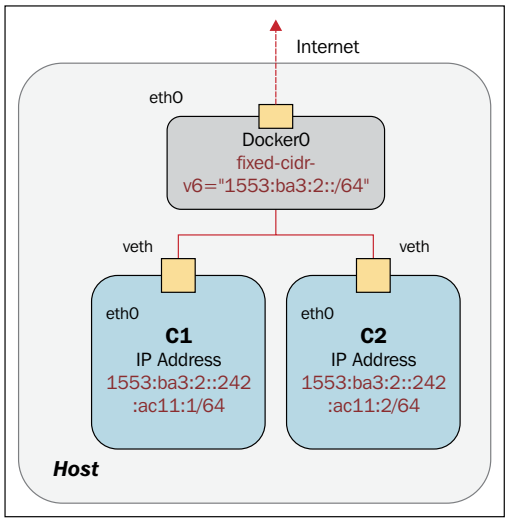
IPv4 and IPv6 can run together; this is called a **dual stack**. This dual stack support is enabled by running the Docker daemon with the `--ipv6` flag. Docker will set up the `docker0` bridge with the IPv6 link-local address `fe80::1`. All packets shared between containers flow through this bridge.

To assign globally routable IPv6 addresses to your containers, you have to specify an IPv6 subnet to pick the addresses from.

The following commands set the IPv6 subnet via the `--fixed-cidr-v6` parameter while starting Docker and also add a new route to the routing table:

```
# docker –d --ipv6 --fixed-cidr-v6="1553:ba3:2::/64"
# docker run -t -i --name c0 ubuntu:latest /bin/bash
```

The following figure shows a Docker bridge configured with an IPv6 address range:



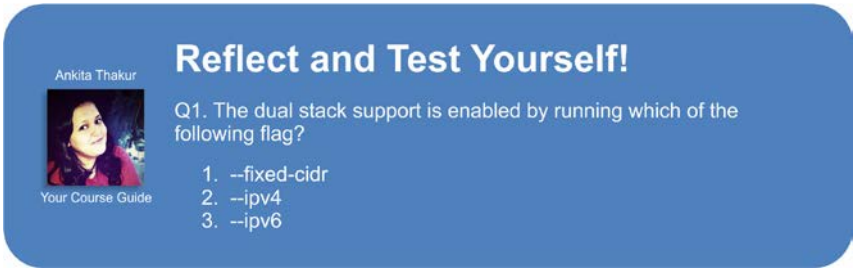
If you check the IP address range using `ifconfig` inside a container, you will notice that the appropriate subnet has been assigned to the `eth0` interface, as shown in the following code:

```
#ifconfig
eth0      Link encap:Ethernet HWaddr 02:42:ac:11:00:01
          inet addr:172.17.0.1  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:1/64 Scope:Link
          inet6 addr: 1553:ba3:2::242:ac11:1/64 Scope:Global
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:738 (738.0 B)  TX bytes:836 (836.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

All the traffic to the `1553:ba3:2::/64` subnet will be routed via the `docker0` interface.

The preceding container is assigned using `fe80::42:acff:fe11:1/64` as the link-local address and `1553:ba3:2::242:ac11:1/64` as the global routable IPv6 address.

Link-local and loopback addresses have link-local scope, which means they are to be used in a directly attached network (link). All other addresses have global (or universal) scope, which means they are globally routable and can be used to connect to addresses with global scope anywhere.



Configuring a DNS server

Docker provides hostname and DNS configurations for each container without us having to build a custom image. It overlays the `/etc` folder inside the container with virtual files, in which it can write new information.

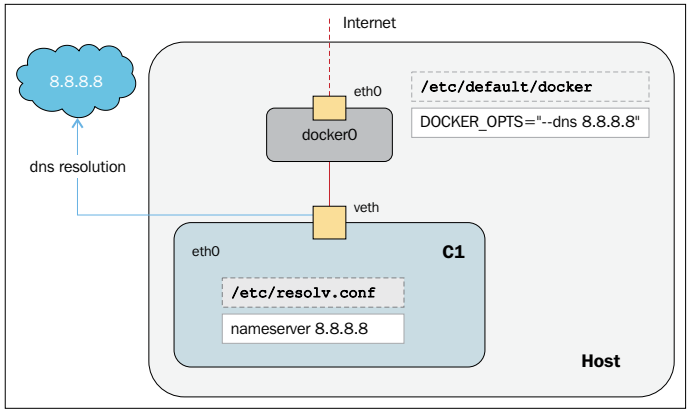
This can be seen by running the `mount` command inside the container. Containers receive the same `resolv.conf` file as that of the host machine when they are created initially. If a host's `resolv.conf` file is modified, this will be reflected in the container's `/resolv.conf` file only when the container is restarted.

In Docker, you can set DNS options in two ways:

- Using `docker run --dns=<ip-address>`
- Adding `DOCKER_OPTS="--dns ip-address"` to the Docker daemon file

You can also specify the search domain using `--dns-search=<DOMAIN>`.

The following figure shows a **nameserver** being configured in a container using the `DOCKER_OPTS` setting in the Docker daemon file:



The main DNS files are as follows:

- `/etc/hostname`
- `/etc/resolv.conf`
- `/etc/hosts`

The following is the command to add a DNS server:

```
# docker run --dns=8.8.8.8 --net="bridge" -t -i  ubuntu:latest
/bin/bash
```

Add hostnames using the following command:

```
#docker run --dns=8.8.8.8 --hostname=docker-vm1  -t -i  ubuntu:latest
/bin/bash
```

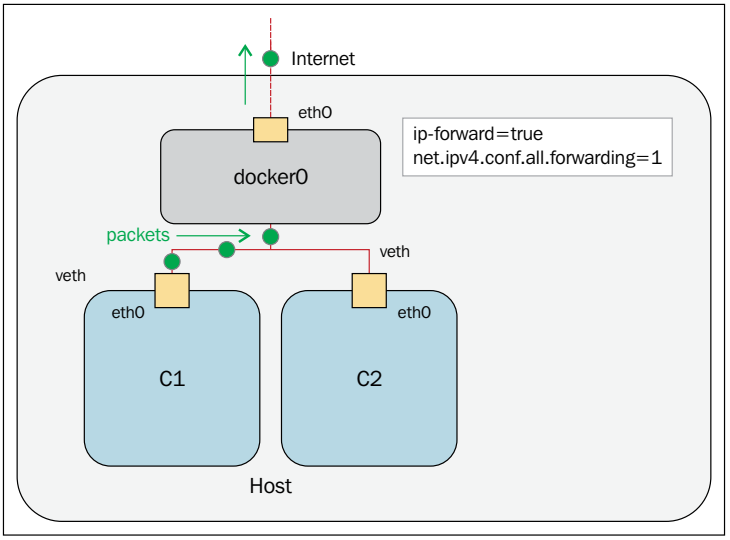
Communication between containers and external networks

Packets can only pass between containers if the `ip_forward` parameter is set to `1`. Usually, you will simply leave the Docker server at its default setting, `--ip-forward=true`, and Docker will set `ip_forward` to `1` for you when the server starts up.

To check the settings or to turn IP forwarding on manually, use these commands:

```
# cat /proc/sys/net/ipv4/ip_forward
0
# echo 1 > /proc/sys/net/ipv4/ip_forward
# cat /proc/sys/net/ipv4/ip_forward
1
```

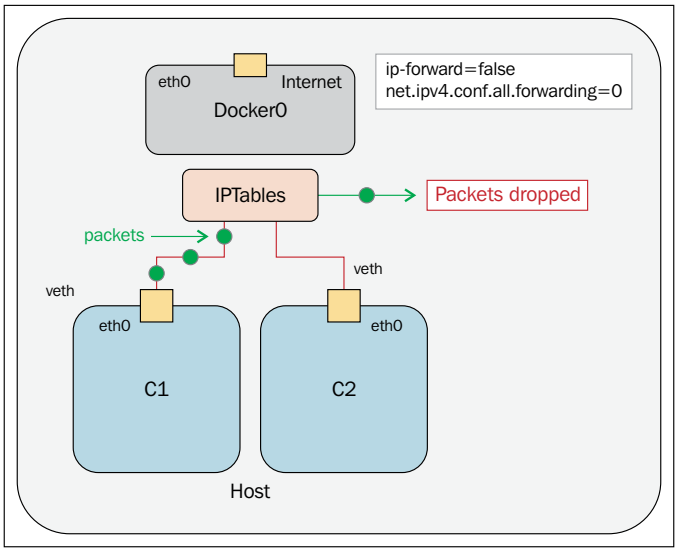
By enabling `ip_forward`, users can make communication possible between containers and the external world; it will also be required for inter-container communication if you are in a multiple-bridge setup. The following figure shows how `ip_forward = false` forwards all the packets to/from the container from/to the external network:



Docker will not delete or modify any pre-existing rules from the Docker filter chain. This allows users to create rules to restrict access to containers.

Docker uses the `docker0` bridge for packet flow between all the containers on a single host. It adds a rule to forward the chain using IPTables in order for the packets to flow between two containers. Setting `--icc=false` will drop all the packets.

When the Docker daemon is configured with both `--icc=false` and `--iptables=true` and `docker run` is invoked with the `--link` option, the Docker server will insert a pair of IPTables accept rules for new containers to connect to the ports exposed by the other containers, which will be the ports that have been mentioned in the exposed lines of its Dockerfile. The following figure shows how `ip_forward = false` drops all the packets to/from the container from/to the external network:



By default, Docker's `forward` rule permits all external IPs. To allow only a specific IP or network to access the containers, insert a negated rule at the top of the Docker filter chain.

For example, using the following command, you can restrict external access such that only the source IP `10.10.10.10` can access the containers:

```
#iptables -I DOCKER -i ext_if ! -s 10.10.10.10 -j DROP
```

Restricting SSH access from one container to another

Following these steps to restrict SSH access from one container to another:

1. Create two containers, `c1` and `c2`.

   For `c1`, use the following command:

   ```
   # docker run -i -t --name c1 ubuntu:latest /bin/bash
   ```

   The output generated is as follows:

   ```
   root@7bc2b6cb1025:/# ifconfig
   eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:05
             inet addr:172.17.0.5  Bcast:0.0.0.0  Mask:255.255.0.0
             inet6 addr: 2001:db8:1::242:ac11:5/64 Scope:Global
             inet6 addr: fe80::42:acff:fe11:5/64 Scope:Link
             UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
             RX packets:7 errors:0 dropped:0 overruns:0 frame:0
             TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:0
             RX bytes:738 (738.0 B)  TX bytes:696 (696.0 B)
   lo        Link encap:Local Loopback
             inet addr:127.0.0.1  Mask:255.0.0.0
             inet6 addr: ::1/128 Scope:Host
             UP LOOPBACK RUNNING  MTU:65536  Metric:1
             RX packets:0 errors:0 dropped:0 overruns:0 frame:0
             TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:0
             RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
   ```

For `c2`, use the following command:

```
# docker run -i -t --name c2 ubuntu:latest /bin/bash
```

The following is the output generated:

```
root@e58a9bf7120b:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:06
          inet addr:172.17.0.6  Bcast:0.0.0.0
Mask:255.255.0.0
          inet6 addr: 2001:db8:1::242:ac11:6/64 Scope:Global
          inet6 addr: fe80::42:acff:fe11:6/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:696 (696.0 B)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

We can test connectivity between the containers using the IP address we've just discovered. Let's see this now using the `ping` tool:

```
root@7bc2b6cb1025:/# ping 172.17.0.6
PING 172.17.0.6 (172.17.0.6) 56(84) bytes of data.
64 bytes from 172.17.0.6: icmp_seq=1 ttl=64 time=0.139 ms
64 bytes from 172.17.0.6: icmp_seq=2 ttl=64 time=0.110 ms
^C
--- 172.17.0.6 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.110/0.124/0.139/0.018 ms
root@7bc2b6cb1025:/#

root@e58a9bf7120b:/# ping 172.17.0.5
```

```
PING 172.17.0.5 (172.17.0.5) 56(84) bytes of data.
64 bytes from 172.17.0.5: icmp_seq=1 ttl=64 time=0.270 ms
64 bytes from 172.17.0.5: icmp_seq=2 ttl=64 time=0.107 ms
^C
--- 172.17.0.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 0.107/0.188/0.270/0.082 ms
root@e58a9bf7120b:/#
```

2. Install `openssh-server` on both the containers:

```
#apt-get install openssh-server
```

3. Enable iptables on the host machine:
    1. Initially, you will be able to SSH from one container to another.
    2. Stop the Docker service and add `DOCKER_OPTS="--icc=false --iptables=true"` to the default Dockerfile of the host machine. This option will enable the iptables firewall and drop all ports between the containers.

       By default, `iptables` is not enabled on the host. Use the following command to enable it:

```
root@ubuntu:~# iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
ctstate RELATED,ESTABLISHED
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
ctstate RELATED,ESTABLISHED
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0


#service docker stop
#vi /etc/default/docker
```

3. Docker Upstart and SysVinit configuration file. Customize the location of the Docker binary (especially for development testing):

```
#DOCKER="/usr/local/bin/docker"
```

4. Use `DOCKER_OPTS` to modify the daemon's startup options:

```
#DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"
#DOCKER_OPTS="--icc=false --iptables=true"
```

5. Restart the Docker service:

```
# service docker start
```

6. Inspect `iptables`:

```
root@ubuntu:~# iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0    ctstate
RELATED, ESTABLISHED
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0   ctstate
RELATED, ESTABLISHED
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
DROP       all  --  0.0.0.0/0            0.0.0.0/0
```

The `DROP` rule has been added to iptables on the host machine, which drops a connection between containers. Now you will be unable to SSH between the containers.

4. We can communicate with or connect containers using the `--link` parameter, with the help of following steps:

1. Create the first container, which will act as the server, `sshserver`:

```
root@ubuntu:~# docker run -i -t -p 2222:22 --name sshserver
ubuntu bash
root@9770be5acbab:/#
```

2. Execute the `iptables` command, and you will find a Docker chain rule added:

```
#root@ubuntu:~# iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
Chain DOCKER (0 references)
target     prot opt source               destination
ACCEPT     tcp  --  0.0.0.0/0            172.17.0.3     tcp
dpt:22
```

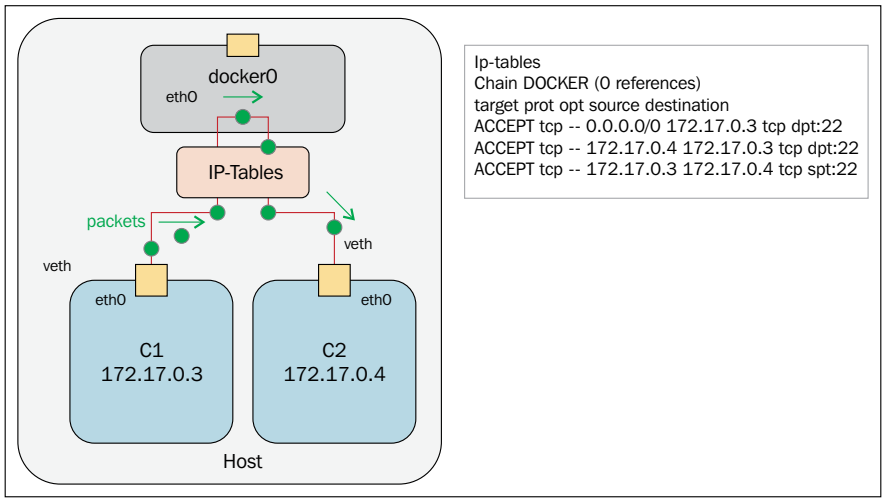
3. Create the second container, which acts like a client, `sshclient`:

```
root@ubuntu:~# docker run -i -t --name sshclient --link
sshserver:sshserver ubuntu bash
root@979d46c5c6a5:/#
```

4. We can see that there are more rules added to the Docker chain rule:

```
root@ubuntu:~# iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
Chain DOCKER (0 references)
target     prot opt source               destination
ACCEPT     tcp  --  0.0.0.0/0            172.17.0.3
tcp dpt:22
ACCEPT     tcp  --  172.17.0.4           172.17.0.3
tcp dpt:22
ACCEPT     tcp  --  172.17.0.3           172.17.0.4
tcp spt:22
root@ubuntu:~#
```

The following image explains communication between the containers using the `--link` flag:



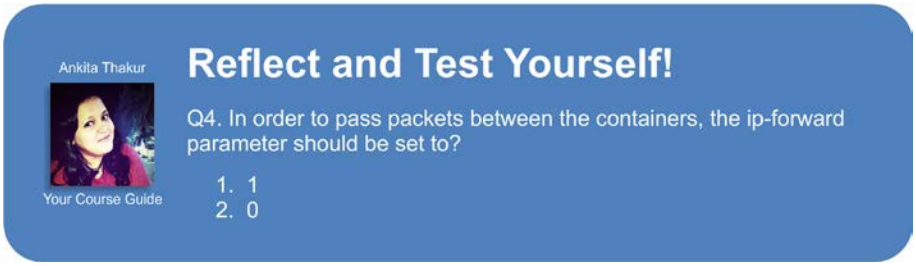
5. You can inspect your linked container with the `docker inspect` command:

```
root@ubuntu:~# docker inspect -f "{{ .HostConfig.Links }}"
sshclient
[/sshserver:/sshclient/sshserver]
```

Now you can successfully ssh into sshserver with its IP.

```
#ssh root@172.17.0.3 -p 22
```

Using the `--link` parameter, Docker creates a secure channel between the containers that doesn't need to expose any ports externally on the containers.



Configuring the Docker bridge

The Docker server creates a bridge called `docker0` by default inside the Linux kernel, and it can pass packets back and forth between other physical or virtual network interfaces so that they behave as a single Ethernet network . Run the following command to find out the list of interfaces in a VM and the IP addresses they are connected to:

```
root@ubuntu:~# ifconfig
docker0   Link encap:Ethernet  HWaddr 56:84:7a:fe:97:99
          inet addr:172.17.42.1  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::5484:7aff:fefe:9799/64 Scope:Link
          inet6 addr: fe80::1/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:11909 errors:0 dropped:0 overruns:0 frame:0
          TX packets:14826 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:516868 (516.8 KB)  TX bytes:46460483 (46.4 MB)
eth0      Link encap:Ethernet  HWaddr 00:0c:29:0d:f4:2c
          inet addr:192.168.186.129  Bcast:192.168.186.255
Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe0d:f42c/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:108865 errors:0 dropped:0 overruns:0 frame:0
          TX packets:31708 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:59902195 (59.9 MB)  TX bytes:3916180 (3.9 MB)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:336 (336.0 B)  TX bytes:336 (336.0 B)
```

Once you have one or more containers up and running, you can confirm that Docker has properly connected them to the `docker0` bridge by running the `brctl` command on the host machine and looking at the `interfaces` column of the output.

Before configuring the `docker0` bridge, install the bridge utilities:

```
# apt-get install bridge-utils
```

Here is a host with two different containers connected:

```
root@ubuntu:~# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.56847afe9799       no              veth21b2e16
                                                        veth7092a45
```

Docker uses the `docker0` bridge settings whenever a container is created. It assigns a new IP address from the range available on the bridge whenever a new container is created, as can be seen here:

```
root@ubuntu:~# docker run -t -i --name container1 ubuntu:latest /bin/bash
root@e54e9312dc04:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:07
          inet addr:172.17.0.7  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: 2001:db8:1::242:ac11:7/64 Scope:Global
          inet6 addr: fe80::42:acff:fe11:7/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:738 (738.0 B)  TX bytes:696 (696.0 B)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
root@e54e9312dc04:/# ip route
default via 172.17.42.1 dev eth0
172.17.0.0/16 dev eth0  proto kernel  scope link  src 172.17.0.7
```

By default, Docker provides a virtual network called `docker0`, which has the IP address `172.17.42.1`. Docker containers have IP addresses in the range of `172.17.0.0/16`.

To change the default settings in Docker, modify the file `/etc/default/docker`.

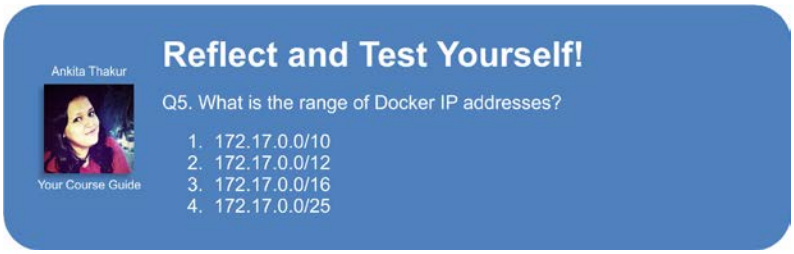
Changing the default bridge from `docker0` to `br0` can be done like this:

```
# sudo service docker stop
# sudo ip link set dev docker0 down
# sudo brctl delbr docker0
# sudo iptables -t nat -F POSTROUTING
# echo 'DOCKER_OPTS="-b=br0"' >> /etc/default/docker
# sudo brctl addbr br0
# sudo ip addr add 192.168.10.1/24 dev br0
# sudo ip link set dev br0 up
# sudo service docker start
```

The following command displays the new bridge name and the IP address range of the Docker service:

```
root@ubuntu:~# ifconfig
br0       Link encap:Ethernet  HWaddr ae:b2:dc:ed:e6:af
          inet addr:192.168.10.1  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::acb2:dcff:feed:e6af/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:738 (738.0 B)
eth0      Link encap:Ethernet  HWaddr 00:0c:29:0d:f4:2c
          inet addr:192.168.186.129  Bcast:192.168.186.255
Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe0d:f42c/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:110823 errors:0 dropped:0 overruns:0 frame:0
          TX packets:33148 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:60081009 (60.0 MB)  TX bytes:4176982 (4.1 MB)
lo        Link encap:Local Loopback
```

```
inet addr:127.0.0.1  Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING  MTU:65536  Metric:1
RX packets:4 errors:0 dropped:0 overruns:0 frame:0
TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:336 (336.0 B)  TX bytes:336 (336.0 B)
```



Overlay networks and underlay networks

An overlay is a virtual network that is built on top of underlying network infrastructure (the underlay). The purpose is to implement a network service that is not available in the physical network.

Network overlay dramatically increases the number of virtual subnets that can be created on top of the physical network, which in turn supports multi-tenancy and virtualization.

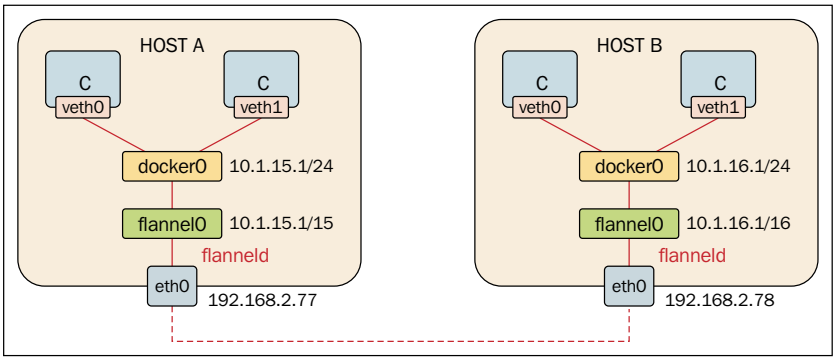
Every container in Docker is assigned an IP address, which is used for communication with other containers. If a container has to communicate with the external network, you set up networking in the host system and expose or map the port from the container to the host machine. With this, applications running inside containers will not be able to advertise their external IP and ports, as the information will not be available to them.

The solution is to somehow assign unique IPs to each Docker container across all hosts and have some networking product that routes traffic between hosts.

There are different projects to deal with Docker networking, as follows:

- Flannel
- Weave
- Open vSwitch

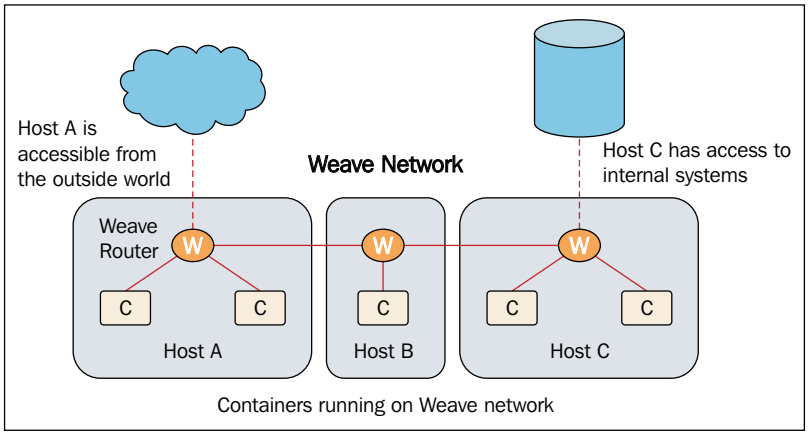
Flannel provides a solution by giving each container an IP that can be used for container-to-container communication. Using packet encapsulation, it creates a virtual overlay network over the host network. By default, Flannel provides a `/24` subnet to hosts, from which the Docker daemon allocates IPs to containers. The following figure shows the communication between containers using Flannel:



Flannel runs an agent, **flanneld**, on each host and is responsible for allocating a subnet lease out of a preconfigured address space. Flannel uses etcd to store the network configuration, allocated subnets, and auxiliary data (such as the host's IP).

Flannel uses the universal TUN/TAP device and creates an overlay network using UDP to encapsulate IP packets. Subnet allocation is done with the help of etcd, which maintains the overlay subnet-to-host mappings.

Weave creates a virtual network that connects Docker containers deployed across hosts/VMs and enables their automatic discovery. The following figure shows a Weave network:



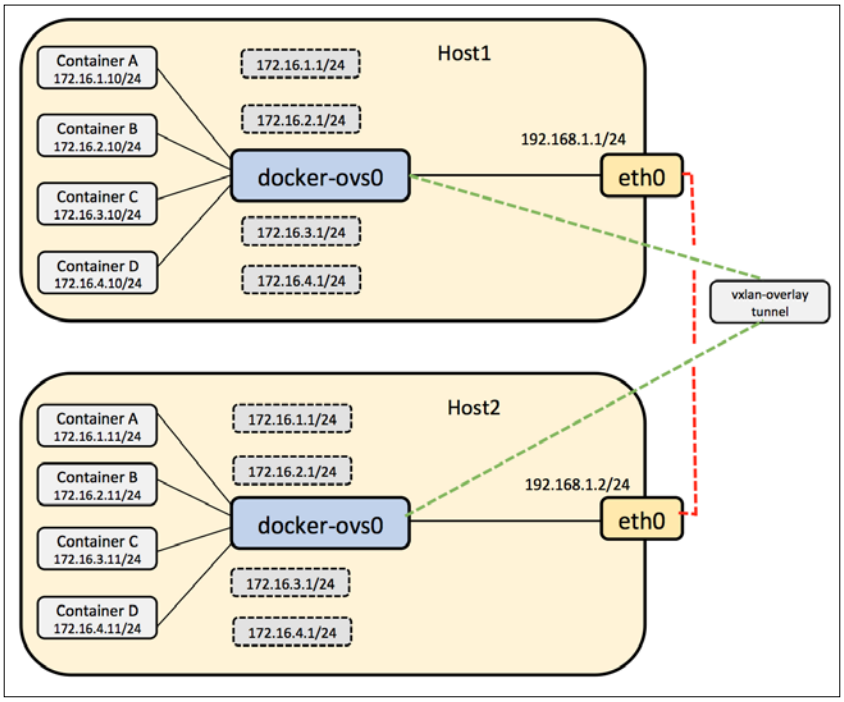
Weave can traverse firewalls and operate in partially connected networks. Traffic can be optionally encrypted, allowing hosts/VMs to be connected across an untrusted network.

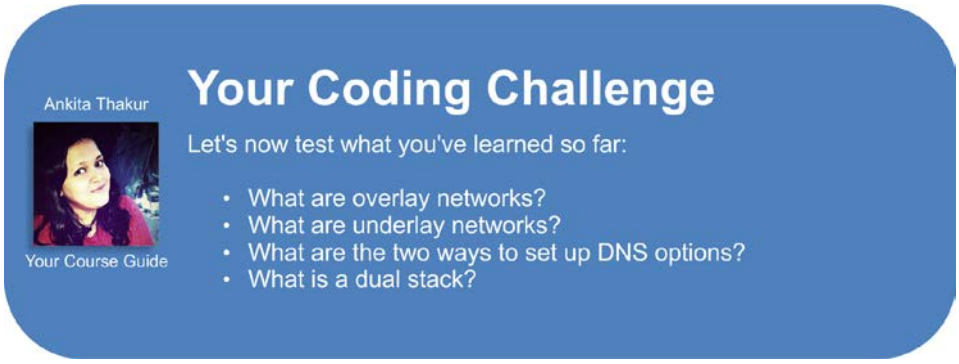
Weave augments Docker's existing (single host) networking capabilities, such as the `docker0` bridge, so these can continue to be used by containers.

Open vSwitch is an open source OpenFlow-capable virtual switch that is typically used with hypervisors to interconnect virtual machines within a host and between different hosts across networks. Overlay networks need to create a virtual datapath using supported tunneling encapsulations, such as VXLAN and GRE.

The overlay datapath is provisioned between tunnel endpoints residing in the Docker host, which gives the appearance of all hosts within a given provider segment being directly connected to one another.

As a new container comes online, the prefix is updated in the routing protocol, announcing its location via a tunnel endpoint. As the other Docker hosts receive the updates, the forwarding rule is installed into the OVS for the tunnel endpoint that the host resides on. When the host is de-provisioned, a similar process occurs and tunnel endpoint Docker hosts remove the forwarding entry for the de-provisioned container. The following figure shows the communication between containers running on multiple hosts through OVS-based VXLAN tunnels:





Summary of Module 2 Chapter 2

Ankita Thakur

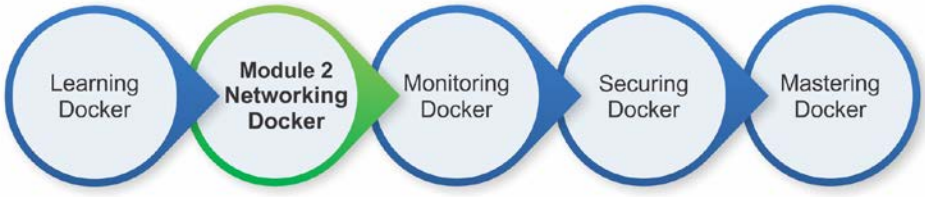
Your Course Guide

In this chapter, we discussed Docker's internal networking architecture. We learned about IPv4, IPv6, and DNS configuration in Docker. Later in the chapter, we covered the Docker bridge and communication between containers within a single host and in multiple hosts.

We also discussed overlay tunneling and different methods that are implemented in Docker networking, such as OVS, Flannel, and Weave.

In the next chapter, we will learn hands-on Docker networking, clubbed with various frameworks.

Your Progress through the Course So Far



3
Building Your First Docker Network

This chapter describes practical examples of Docker networking, spanning multiple containers over multiple hosts. We will cover the following topics:

- Introduction to Pipework
- Multiple containers over multiple hosts
- Towards scaling networks – introducing Open vSwitch
- Networking with overlay networks – Flannel
- Comparison of Docker networking options

Introduction to Pipework

Pipework lets you connect together containers in arbitrarily complex scenarios.

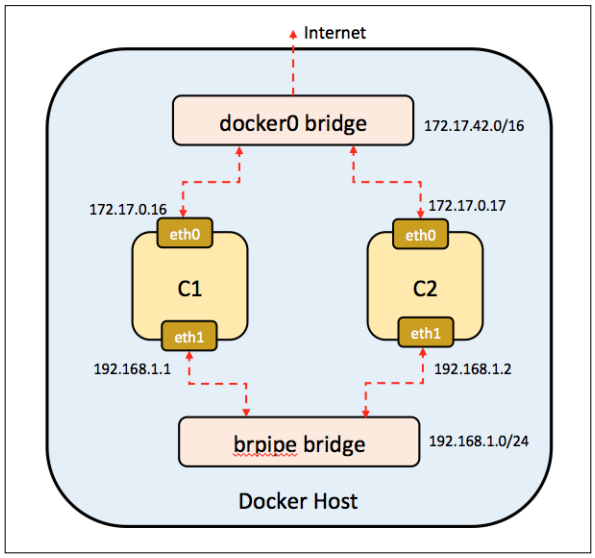
In practical terms, it creates a legacy Linux bridge, adds a new interface to the container, and then attaches the interface to that bridge; containers get a network segment on which to communicate with each other.

Multiple containers over a single host

Pipework is a shell script and installing it is simple:

```
#sudo wget -O /usr/local/bin/pipework
https://raw.githubusercontent.com/jpetazzo/pipework/master/pipewor
k && sudo chmod +x /usr/local/bin/pipework
```

The following figure shows container communication using Pipework:



First, create two containers:

```
#docker run -i -t --name c1 ubuntu:latest /bin/bash
root@5afb44195a69:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:10
          inet addr:172.17.0.16  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:10/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:13 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1038 (1.0 KB)  TX bytes:738 (738.0 B)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
```

```
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

#docker run -i -t --name c2 ubuntu:latest /bin/bash
root@c94d53a76a9b:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:11
          inet addr:172.17.0.17  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:11/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:738 (738.0 B)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Now let's use Pipework to connect them:

```
#sudo pipework brpipe c1 192.168.1.1/24
```

This command creates a bridge, `brpipe`, on the host machine. It adds an `eth1` interface to the container `c1` with the IP address `192.168.1.1` and attaches the interface to the bridge as follows:

```
root@5afb44195a69:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:10
          inet addr:172.17.0.16  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:10/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:13 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1038 (1.0 KB)  TX bytes:738 (738.0 B)
```

```
eth1      Link encap:Ethernet  HWaddr ce:72:c5:12:4a:1a
          inet addr:192.168.1.1  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::cc72:c5ff:fe12:4a1a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:23 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1806 (1.8 KB)  TX bytes:690 (690.0 B)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
#sudo pipework brpipe c2 192.168.1.2/24
```

This command will not create bridge `brpipe` as it already exists. It will add an `eth1` interface to the container `c2` and connect it to the bridge as follows:

```
root@c94d53a76a9b:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:11
          inet addr:172.17.0.17  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:11/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:738 (738.0 B)
eth1      Link encap:Ethernet  HWaddr 36:86:fb:9e:88:ba
          inet addr:192.168.1.2  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::3486:fbff:fe9e:88ba/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
```

```
          collisions:0 txqueuelen:1000
          RX bytes:648 (648.0 B)  TX bytes:690 (690.0 B)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Now the containers are connected and will be able to ping each other as they are on the same subnet, `192.168.1.0/24`. Pipework provides the advantage of adding static IP addresses to the containers.

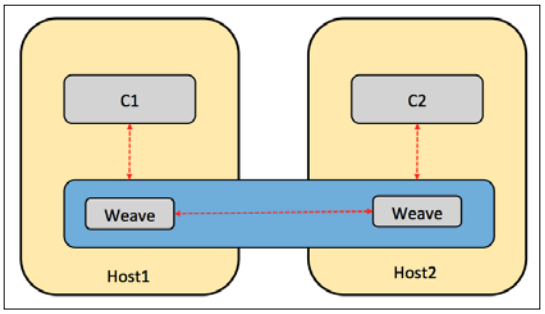
Weave your containers

Weave creates a virtual network that can connect Docker containers across multiple hosts as if they are all connected to a single switch. The Weave router itself runs as a Docker container and can encrypt routed traffic for transmission over the Internet. Services provided by application containers on the Weave network can be made accessible to the outside world, regardless of where those containers are running.

Use the following code to install Weave:

```
#sudo curl -L git.io/weave -o /usr/local/bin/weave
#sudo chmod a+x /usr/local/bin/weave
```

The following figure shows multihost communication using Weave:



On `$HOST1`, we run the following:

```
# weave launch
# eval $(weave proxy-env)
# docker run --name c1 -ti ubuntu
```

Next, we repeat similar steps on `$HOST2`:

```
# weave launch $HOST1
# eval $(weave proxy-env)
# docker run --name c2 -ti ubuntu
```

In the container started on `$HOST1`, the following output is generated:

```
root@c1:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:21
          inet addr:172.17.0.33  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:21/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:38 errors:0 dropped:0 overruns:0 frame:0
          TX packets:34 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:3166 (3.1 KB)  TX bytes:2299 (2.2 KB)
ethwe     Link encap:Ethernet  HWaddr aa:99:8a:d5:4d:d4
          inet addr:10.128.0.3  Bcast:0.0.0.0  Mask:255.192.0.0
          inet6 addr: fe80::a899:8aff:fed5:4dd4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:65535  Metric:1
          RX packets:130 errors:0 dropped:0 overruns:0 frame:0
          TX packets:74 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:11028 (11.0 KB)  TX bytes:6108 (6.1 KB)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

You can see the Weave network interface, `ethwe`, using the `ifconfig` command:

```
root@c2:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:04
          inet addr:172.17.0.4  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:28 errors:0 dropped:0 overruns:0 frame:0
          TX packets:29 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:2412 (2.4 KB)  TX bytes:2016 (2.0 KB)
ethwe     Link encap:Ethernet  HWaddr 8e:7c:17:0d:0e:03
          inet addr:10.160.0.1  Bcast:0.0.0.0  Mask:255.192.0.0
          inet6 addr: fe80::8c7c:17ff:fe0d:e03/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:65535  Metric:1
          RX packets:139 errors:0 dropped:0 overruns:0 frame:0
          TX packets:74 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:11718 (11.7 KB)  TX bytes:6108 (6.1 KB)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

#root@c1:/# ping -c 1 -q c2
PING c2.weave.local (10.160.0.1) 56(84) bytes of data.
--- c2.weave.local ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.317/1.317/1.317/0.000 ms
```

Similarly, in the container started on `$HOST2`, the following output is generated:

```
#root@c2:/# ping -c 1 -q c1
PING c1.weave.local (10.128.0.3) 56(84) bytes of data.
--- c1.weave.local ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.658/1.658/1.658/0.000 ms
```

So there we have it—two containers on separate hosts happily talking to each other.

Open vSwitch

Docker uses the Linux bridge `docker0` by default. However, there are cases where **Open vSwitch** (**OVS**) might be required instead of a Linux bridge. A single Linux bridge can only handle 1024 ports – this limits the scalability of Docker as we can only create 1024 containers, each with a single network interface.

Single host OVS

We will now install OVS on a single host, create two containers, and connect them to an OVS bridge.

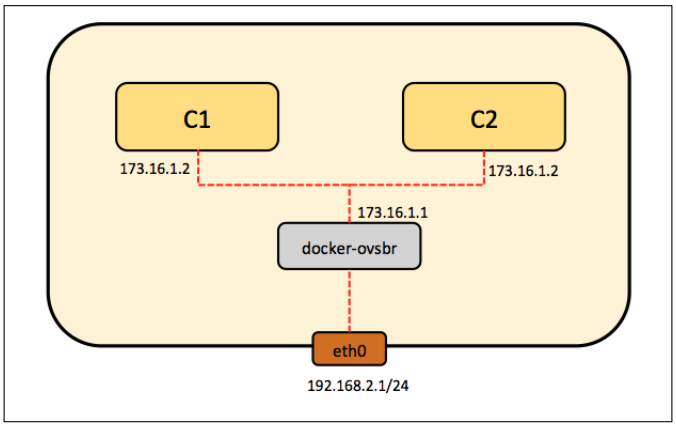
Use this command to install OVS:

```
# sudo apt-get install openvswitch-switch
```

Install the `ovs-docker` utility with the following:

```
# cd /usr/bin
# wget
https://raw.githubusercontent.com/openvswitch/ovs/master/utilities
/ovs-docker
# chmod a+rwx ovs-docker
```

The following diagram shows the single-host OVS:



Creating an OVS bridge

Here, we will be adding a new OVS bridge and configuring it so that we can get the containers connected on a different network, as follows:

```
# ovs-vsctl add-br ovs-br1
# ifconfig ovs-br1 173.16.1.1 netmask 255.255.255.0 up
```

Add a port from the OVS bridge to the Docker container using the following steps:

1. Create two Ubuntu Docker containers:

```
# docker run -I -t --name container1 ubuntu /bin/bash
# docekr run -I -t --name container2 ubuntu /bin/bash
```

2. Connect the container to the OVS bridge:

```
# ovs-docker add-port ovs-br1 eth1 container1 --
ipaddress=173.16.1.2/24
# ovs-docker add-port ovs-br1 eth1 container2 --
ipaddress=173.16.1.3/24
```

3. Test the connection between the two containers connected via an OVS bridge using the `ping` command. First, find out their IP addresses:

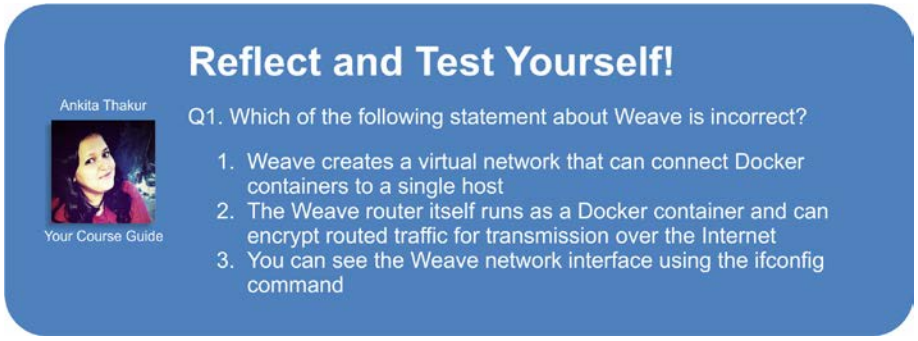
```
# docker exec container1 ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:10:11:02
          inet addr:172.16.17.2  Bcast:0.0.0.0
Mask:255.255.255.0
          inet6 addr: fe80::42:acff:fe10:1102/64 Scope:Link
```

```
          UP BROADCAST RUNNING MULTICAST  MTU:1472
Metric:1
          RX packets:36 errors:0 dropped:0 overruns:0
frame:0
          TX packets:8 errors:0 dropped:0 overruns:0
carrier:0
          collisions:0 txqueuelen:0
          RX bytes:4956 (4.9 KB)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0
frame:0
          TX packets:0 errors:0 dropped:0 overruns:0
carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

# docker exec container2 ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:10:11:03
          inet addr:172.16.17.3  Bcast:0.0.0.0
Mask:255.255.255.0
          inet6 addr: fe80::42:acff:fe10:1103/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1472
Metric:1
          RX packets:27 errors:0 dropped:0 overruns:0
frame:0
          TX packets:8 errors:0 dropped:0 overruns:0
carrier:0
          collisions:0 txqueuelen:0
          RX bytes:4201 (4.2 KB)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0
frame:0
          TX packets:0 errors:0 dropped:0 overruns:0
carrier:0
```

```
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Now that we know the IP addresses of `container1` and `container2`, we can ping them:

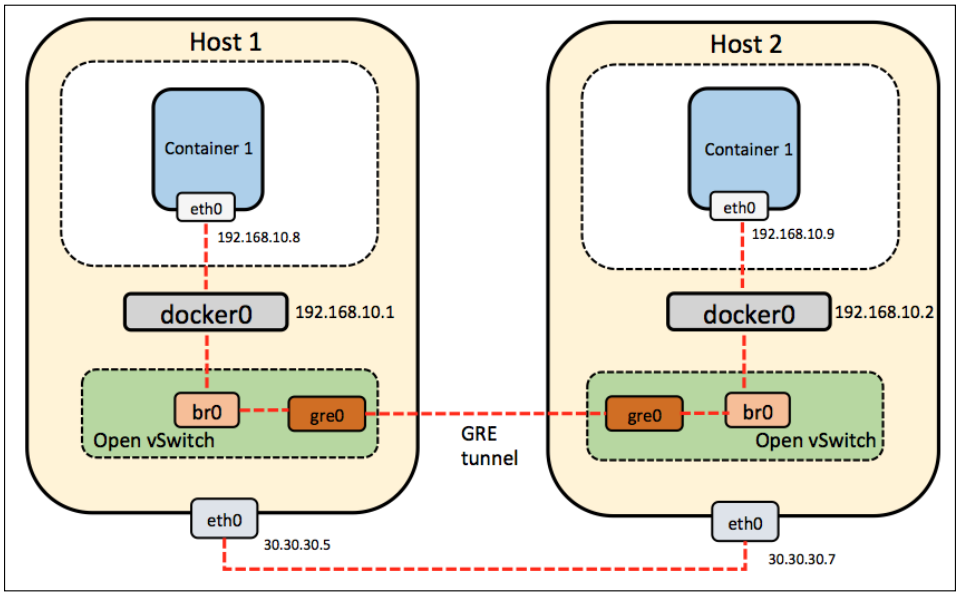
```
# docker exec container2 ping 172.16.17.2
PING 172.16.17.2 (172.16.17.2) 56(84) bytes of data.
64 bytes from 172.16.17.2: icmp_seq=1 ttl=64 time=0.257 ms
64 bytes from 172.16.17.2: icmp_seq=2 ttl=64 time=0.048 ms
64 bytes from 172.16.17.2: icmp_seq=3 ttl=64 time=0.052 ms

# docker exec container1 ping 172.16.17.2
PING 172.16.17.2 (172.16.17.2) 56(84) bytes of data.
64 bytes from 172.16.17.2: icmp_seq=1 ttl=64 time=0.060 ms
64 bytes from 172.16.17.2: icmp_seq=2 ttl=64 time=0.035 ms
64 bytes from 172.16.17.2: icmp_seq=3 ttl=64 time=0.031 ms
```



Multiple host OVS

Let's see how to connect Docker containers on multiple hosts using OVS.

Let's consider our setup as shown in the following diagram, which contains two hosts, **Host 1** and **Host 2**, running Ubuntu 14.04:



Install Docker and Open vSwitch on both the hosts:

```
# wget -qO- https://get.docker.com/ | sh
# sudo apt-get install openvswitch-switch
```

Install the `ovs-docker` utility:

```
# cd /usr/bin
# wget
https://raw.githubusercontent.com/openvswitch/ovs/master/utilities/ov
s-docker
# chmod a+rwx ovs-docker
```

By default, Docker chooses a random network to run its containers in. It creates a bridge, `docker0`, and assigns an IP address (`172.17.42.1`) to it. So, both **Host 1** and **Host 2** `docker0` bridge IP addresses are the same, due to which it is difficult for containers in both the hosts to communicate. To overcome this, let's assign static IP addresses to the network, that is, `192.168.10.0/24`.

Let's see how to change the default Docker subnet.

Execute the following commands on Host 1:

```
# service docker stop
# ip link set dev docker0 down
# ip addr del 172.17.42.1/16 dev docker0
# ip addr add 192.168.10.1/24 dev docker0
# ip link set dev docker0 up
# ip addr show docker0
# service docker start
```

Add the `br0` OVS bridge:

```
# ovs-vsctl add-br br0
```

Create the tunnel to the other host and attach it to the:

```
# add-port br0 gre0 -- set interface gre0 type=gre
options:remote_ip=30.30.30.8
```

Add the `br0` bridge to the `docker0` bridge:

```
# brctl addif docker0 br0
```

Execute the following commands on Host 2:

```
# service docker stop
# iptables -t nat -F POSTROUTING
# ip link set dev docker0 down
# ip addr del 172.17.42.1/16 dev docker0
# ip addr add 192.168.10.2/24 dev docker0
# ip link set dev docker0 up
# ip addr show docker0
# service docker start
```

Add the `br0` OVS bridge:

```
# ip link set br0 up
# ovs-vsctl add-br br0
```

Create the tunnel to the other host and attach it to the:

```
# br0 bridge ovs-vsctl add-port br0 gre0 -- set interface gre0
type=gre options:remote_ip=30.30.30.7
```

Add the `br0` bridge to the `docker0` bridge:

```
# brctl addif docker0 br0
```

The `docker0` bridge is attached to another bridge, `br0`. This time, it's an OVS bridge. This means that all traffic between the containers is routed through `br0` too.

Additionally, we need to connect together the networks from both the hosts in which the containers are running. A GRE tunnel is used for this purpose. This tunnel is attached to the `br0` OVS bridge and, as a result, to `docker0` too.

After executing the preceding commands on both hosts, you should be able to ping the `docker0` bridge addresses from both hosts.

On Host 1, the following output is generated on using the `ping` command:

```
# ping 192.168.10.2
PING 192.168.10.2 (192.168.10.2) 56(84) bytes of data.
64 bytes from 192.168.10.2: icmp_seq=1 ttl=64 time=0.088 ms
64 bytes from 192.168.10.2: icmp_seq=2 ttl=64 time=0.032 ms
^C
--- 192.168.10.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.032/0.060/0.088/0.028 ms
```

On Host 2, the following output is generated on using the `ping` command:

```
# ping 192.168.10.1
PING 192.168.10.1 (192.168.10.1) 56(84) bytes of data.
64 bytes from 192.168.10.1: icmp_seq=1 ttl=64 time=0.088 ms
64 bytes from 192.168.10.1: icmp_seq=2 ttl=64 time=0.032 ms
^C
--- 192.168.10.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.032/0.060/0.088/0.028 ms
```

Let's see how to create containers on both the hosts.

On Host 1, use the following code:

```
# docker run -t -i --name container1 ubuntu:latest /bin/bash
```

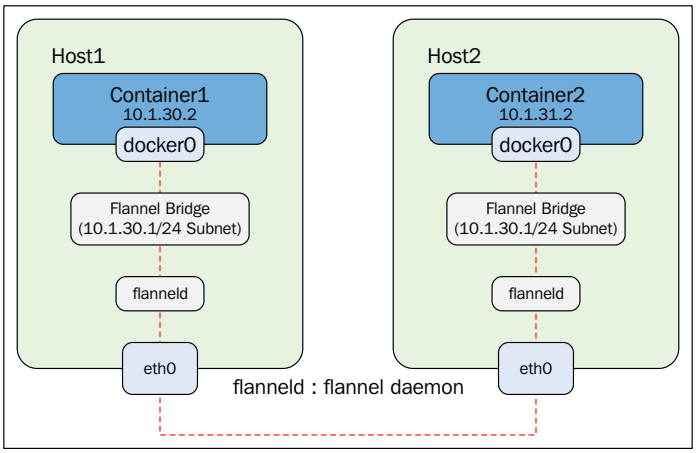
On Host 2, use the following code:

```
# docker run -t -i --name container2 ubuntu:latest /bin/bash
```

Now we can ping `container2` from `container1`. In this way, we connect Docker containers on multiple hosts using Open vSwitch.

Networking with overlay networks – Flannel

Flannel is the virtual network layer that provides the subnet to each host for use with Docker containers. It is packaged with CoreOS but can be configured on other Linux OSes as well. Flannel creates the overlay by actually connecting itself to Docker bridge, to which containers are attached, as shown in the following figure. To setup Flannel, two host machines or VMs are required, which can be CoreOS or, more preferably, Linux OS, as shown in this figure:



The Flannel code can be cloned from GitHub and built locally, if required, on a different flavor of Linux OS, as shown here. It comes preinstalled in CoreOS:

```
# git clone https://github.com/coreos/flannel.git
Cloning into 'flannel'...
remote: Counting objects: 2141, done.
remote: Compressing objects: 100% (19/19), done.
remote: Total 2141 (delta 6), reused 0 (delta 0), pack-reused 2122
Receiving objects: 100% (2141/2141), 4.
Checking connectivity... done.

# sudo docker run -v `pwd`:/opt/flannel -i -t google/golang
/bin/bash -c "cd /opt/flannel && ./build"
Building flanneld...
```

CoreOS machines can be easily configured using Vagrant and VirtualBox, as per the tutorial mentioned in the following link:

`https://coreos.com/os/docs/latest/booting-on-vagrant.html`

After the machines are created and logged in to, we will find a Flannel bridge automatically created using the `etcd` configuration:

```
# ifconfig flannel0
flannel0: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST>  mtu 1472
          inet 10.1.30.0  netmask 255.255.0.0  destination 10.1.30.0
          unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
txqueuelen 500 (UNSPEC)
          RX packets 243  bytes 20692 (20.2 KiB)
          RX errors 0  dropped 0  overruns 0  frame 0
          TX packets 304  bytes 25536 (24.9 KiB)
          TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

The Flannel environment can be checked by viewing `subnet.env`:

```
# cat /run/flannel/subnet.env
FLANNEL_NETWORK=10.1.0.0/16
FLANNEL_SUBNET=10.1.30.1/24
FLANNEL_MTU=1472
FLANNEL_IPMASQ=true
```

The Docker daemon requires to be restarted with the following commands in order to get the networking re-instantiated with the subnet from the Flannel bridge:

```
# source /run/flannel/subnet.env
# sudo rm /var/run/docker.pid
# sudo ifconfig docker0 ${FLANNEL_SUBNET}
# sudo docker -d --bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU}
& INFO[0000] [graphdriver] using prior storage driver "overlay"
INFO[0000] Option DefaultDriver: bridge
INFO[0000] Option DefaultNetwork: bridge
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
INFO[0000] Firewalld running: false
INFO[0000] Loading containers: start.
.............
INFO[0000] Loading containers: done.
```

```
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon
commit=cedd534-dirty execdriver=native-0.2 graphdriver=overlay
version=1.8.3
```

The Flannel environment for the second host can also be checked by viewing `subnet.env`:

```
# cat /run/flannel/subnet.env
FLANNEL_NETWORK=10.1.0.0/16
FLANNEL_SUBNET=10.1.31.1/24
FLANNEL_MTU=1472
FLANNEL_IPMASQ=true
```

A different subnet is allocated to the second host. The Docker service can also be restarted in this host by pointing to the Flannel bridge:

```
# source /run/flannel/subnet.env
# sudo ifconfig docker0 ${FLANNEL_SUBNET}
# sudo docker -d --bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU} &
INFO[0000] [graphdriver] using prior storage driver "overlay"
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
INFO[0000] Option DefaultDriver: bridge
INFO[0000] Option DefaultNetwork: bridge
INFO[0000] Firewalld running: false
INFO[0000] Loading containers: start.
....
INFO[0000] Loading containers: done.
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon
commit=cedd534-dirty execdriver=native-0.2 graphdriver=overlay
version=1.8.3
```

Docker containers can be created in their respective hosts, and they can be tested using the `ping` command in order to check the Flannel overlay network connectivity.

For Host 1, use the following commands:

```
#docker run -it ubuntu /bin/bash
INFO[0013] POST /v1.20/containers/create
INFO[0013] POST
/v1.20/containers/1d1582111801c8788695910e57c02fdba593f443c15e2f
```

```
1db9174ed9078db809/attach?stderr=1&stdin=1&stdout=1&stream=1
INFO[0013] POST
/v1.20/containers/1d1582111801c8788695910e57c02fdba593f443c15e2f
1db9174ed9078db809/start
INFO[0013] POST
/v1.20/containers/1d1582111801c8788695910e57c02fdba593f443c15e2f
1db9174ed9078db809/resize?h=44&w=80

root@1d1582111801:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:0a:01:1e:02
          inet addr:10.1.30.2  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:aff:fe01:1e02/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1472  Metric:1
          RX packets:11 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:969 (969.0 B)  TX bytes:508 (508.0 B)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

For Host 2, use the following commands:

```
# docker run -it ubuntu /bin/bash
root@ed070166624a:/# ifconfig
eth0       Link encap:Ethernet  HWaddr 02:42:0a:01:1f:02
           inet addr:10.1.31.2  Bcast:0.0.0.0  Mask:255.255.255.0
           inet6 addr: fe80::42:aff:fe01:1f02/64 Scope:Link
           UP BROADCAST RUNNING MULTICAST  MTU:1472  Metric:1
           RX packets:18 errors:0 dropped:2 overruns:0 frame:0
           TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:0
           RX bytes:1544 (1.5 KB)  TX bytes:598 (598.0 B)
```

```
lo          Link encap:Local Loopback
            inet addr:127.0.0.1  Mask:255.0.0.0
            inet6 addr: ::1/128 Scope:Host
            UP LOOPBACK RUNNING  MTU:65536  Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:0
            RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
root@ed070166624a:/# ping 10.1.30.2
PING 10.1.30.2 (10.1.30.2) 56(84) bytes of data.
64 bytes from 10.1.30.2: icmp_seq=1 ttl=60 time=3.61 ms
64 bytes from 10.1.30.2: icmp_seq=2 ttl=60 time=1.38 ms
64 bytes from 10.1.30.2: icmp_seq=3 ttl=60 time=0.695 ms
64 bytes from 10.1.30.2: icmp_seq=4 ttl=60 time=1.49 ms
```

Thus, in the preceding example, we can see the complexity that Flannel reduces by running the `flanneld` agent on each host, which is responsible for allocating a subnet lease out of preconfigured address space. Flannel internally uses `etcd` to store the network configuration and other details, such as host IP and allocated subnets. The forwarding of packets is achieved using the backend strategy.

Flannel also aims to resolve the problem of Kubernetes deployment on cloud providers other than GCE, where a Flannel overlay mesh network can ease the issue of assigning a unique IP address to each pod by creating a subnet for each server.

Summary of Module 2 Chapter 3

In this chapter, we learnt how Docker containers communicate across multiple hosts using different networking options such as Weave, OVS, and Flannel. Pipework uses the legacy Linux bridge, Weave creates a virtual network, OVS uses GRE tunneling technology, and Flannel provides a separate subnet to each host in order to connect containers to multiple hosts. Some of the implementations, such as Pipework, are legacy and will become obsolete over a period of time, while others are designed to be used in the context of specific OSes, such as Flannel with CoreOS.

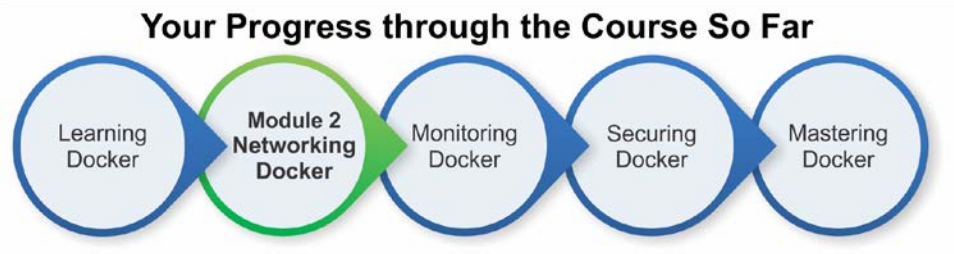
Ankita Thakur

Your Course Guide

The following diagram shows a basic comparison of Docker networking options:

	Network Type
Weave	Virtual Overlay Network
Flannel	Creates separate Subnet
Open vSwitch	GRE Tunneling
Pipework	Legacy Linux Bridge

In the next chapter, we will discuss how Docker containers are networked when using frameworks such as Kubernetes, Docker Swarm, and Mesosphere.



4
Networking in a Docker Cluster

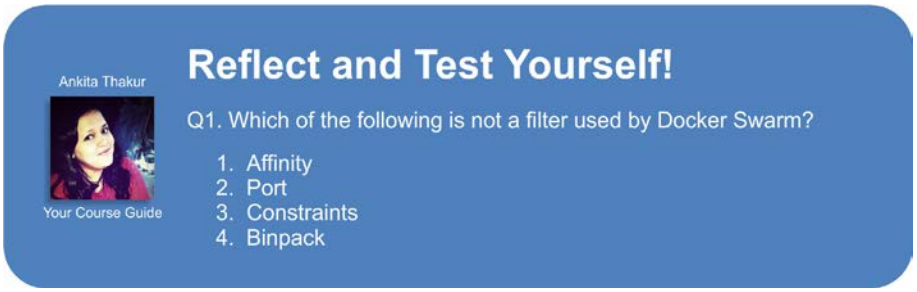
In this chapter, you will learn how Docker containers are networked when using frameworks like Kubernetes, Docker Swarm, and Mesosphere.

We will cover the following topics:

- Docker Swarm
- Kubernetes
    - Networked containers in a Kubernetes cluster
    - How Kubernetes networking differs from Docker networking
    - Kubernetes on AWS
- Mesosphere

Docker Swarm

Docker Swarm is a native clustering system for Docker. Docker Swarm exposes the standard Docker API so that any tool that communicates with the Docker daemon can communicate with Docker Swarm as well. The basic aim is to allow the creation and usage of a pool of Docker hosts together. The cluster manager of Swarm schedules the containers based on the availability resources in a cluster. We can also specify the constrained resources for a container while deploying it. Swarm is designed to pack containers onto a host by saving other host resources for heavier and bigger containers rather than scheduling them randomly to a host in the cluster.



Similar to other Docker projects, Docker Swarm uses a Plug and Play architecture. Docker Swarm provides backend services to maintain a list of IP addresses in your Swarm cluster. There are several services, such as etcd, Consul, and Zookeeper; even a static file can be used. Docker Hub also provides a hosted discovery service, which is used in the normal configuration of Docker Swarm.

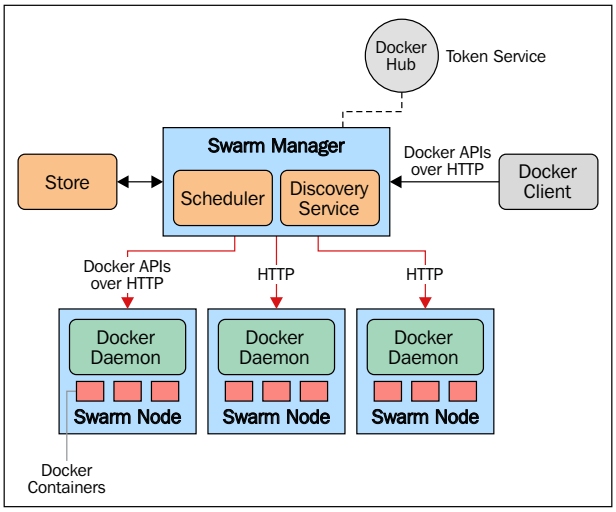
Docker Swarm scheduling uses multiple strategies in order to rank nodes. When a new container is created, Swarm places it on the node on the basis of the highest computed rank, using the following strategies:

1. **Spread**: This optimizes and schedules the containers on the nodes based on the number of containers running on the node at that point of time
2. **Binpack**: The node is selected to schedule the container on the basis of CPU and RAM utilization
3. **Random strategy**: This uses no computation; it selects the node randomly to schedule containers

Docker Swarm also uses filters in order to schedule containers, such as:

- **Constraints**: These use key/value pairs associated with nodes, such as `environment=production`
- **Affinity filter**: This is used to run a container and instruct it to locate and run next to another container based on the label, image, or identifier
- **Port filter**: In this case, the node is selected on the basis of the ports available on it
- **Dependency filter**: This co-schedules dependent containers on the same node
- **Health filter**: This prevents the scheduling of containers on unhealthy nodes

The following figure explains various components of a Docker Swarm cluster:



Docker Swarm setup

Let's set up our Docker Swarm setup, which will have two nodes and one master.

We will be using a Docker client in order to access the Docker Swarm cluster. A Docker client can be set up on a machine or laptop and should have access to all the machines present in the Swarm cluster.

After installing Docker on all three machines, we will restart the Docker service from a command line so that it can be accessed from TCP port 2375 on the localhost (`0.0.0.0:2375`) or from a specific host IP address and can allow connections using a Unix socket on all the Swarm nodes, as follows:

```
$ docker -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock -d &
```

Docker Swarm images are required to be deployed as Docker containers on the master node. In our example, the master node's IP address is `192.168.59.134`. Replace it with your Swarm's master node. From the Docker client machine, we will be installing Docker Swarm on the master node using the following command:

```
$ sudo docker -H tcp://192.168.59.134:2375 run --rm swarm create
Unable to find image 'swarm' locally
Pulling repository swarm
e12f8c5e4c3b: Download complete
cf43a42a05d1: Download complete
```

```
42c4e5c90ee9: Download complete
22cf18566d05: Download complete
048068586dc5: Download complete
2ea96b3590d8: Download complete
12a239a7cb01: Download complete
26b910067c5f: Download complete
4fdfeb28bd618291eeb97a2096b3f841
```

The Swarm token generated after the execution of the command should be noted, as it will be used for the Swarm setup. In our case, it is this:

```
"4fdfeb28bd618291eeb97a2096b3f841"
```

The following are the steps to set up a two-node Docker Swarm cluster:

1. From the Docker client node, the following `docker` command is required to be executed with Node 1's IP address (in our case, `192.168.59.135`) and the Swarm token generated in the preceding code in order to add it to the Swarm cluster:

   ```
   $ docker -H tcp://192.168.59.135:2375 run -d swarm join --
   addr=192.168.59.135:2375 token://
   4fdfeb28bd618291eeb97a2096b3f841
   Unable to find image 'swarm' locally
   Pulling repository swarm
   e12f8c5e4c3b: Download complete
   cf43a42a05d1: Download complete
   42c4e5c90ee9: Download complete
   22cf18566d05: Download complete
   048068586dc5: Download complete
   2ea96b3590d8: Download complete
   12a239a7cb01: Download complete
   26b910067c5f: Download complete
   e4f268b2cc4d896431dacdafdc1bb56c98fed01f58f8154ba13908c7e6fe67
   5b
   ```

2. Repeat the preceding steps for Node 2 by replacing Node 1's IP address with Node 2's.

3. Swarm manager is required to be set up on the master node using the following command on the Docker client node:

```
$ sudo docker -H tcp://192.168.59.134:2375 run -d -p 5001:2375
swarm manage token:// 4fdfeb28bd618291eeb97a2096b3f841

f06ce375758f415614dc5c6f71d5d87cf8edecffc6846cd978fe07fafc3d05
d3
```

   The Swarm cluster is set up and can be managed using the Swarm manager residing on the master node. To list all the nodes, the following command can be executed using a Docker client:

```
$ sudo docker -H tcp://192.168.59.134:2375 run --rm swarm list
\ token:// 4fdfeb28bd618291eeb97a2096b3f841

192.168.59.135:2375

192.168.59.136:2375
```

4. The following command can be used to get information about the cluster:

```
$ sudo docker -H tcp://192.168.59.134:5001 info

Containers: 0

Strategy: spread

Filters: affinity, health, constraint, port, dependency

Nodes: 2

agent-1: 192.168.59.136:2375

    └ Containers: 0

    └ Reserved CPUs: 0 / 8

    └ Reserved Memory: 0 B / 1.023 GiB

 agent-0: 192.168.59.135:2375

    └ Containers: 0

    └ Reserved CPUs: 0 / 8

    └ Reserved Memory: 0 B / 1.023 GiB
```

5. The test `ubuntu` container can be launched onto the cluster by specifying the name as `swarm-ubuntu` and using the following command:

```
$ sudo docker -H tcp://192.168.59.134:5001 run -it --name
swarm-ubuntu ubuntu /bin/sh
```

6. The container can be listed using the Swarm master's IP address:

```
$ sudo docker -H tcp://192.168.59.134:5001 ps
```

This completes the setup of a two-node Docker Swarm cluster.

Docker Swarm networking

Docker Swarm networking has integration with libnetwork and even provides support for overlay networks. libnetwork provides a Go implementation to connect containers; it is a robust container network model that provides network abstraction for applications and the programming interface of containers. Docker Swarm is now fully compatible with the new networking model in Docker 1.9 (note that we will be using Docker 1.9 in the following setup). The key-value store is required for overlay networks, which includes discovery, networks, IP addresses, and more information.

In the following example, we will be using Consul to understand Docker Swarm networking in a better way:

1. We will provision a VirtualBox machine called `sample-keystore` using `docker-machine`:

```
$ docker-machine create -d virtualbox sample-keystore
Running pre-create checks...
Creating machine...
Waiting for machine to be running, this may take a few
minutes...
Machine is running, waiting for SSH to be available...
Detecting operating system of created instance...
Provisioning created instance...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
To see how to connect Docker to this machine, run: docker-
machine.exe env sample-keystore
```

2. We will also deploy the `progrium/consul` container on the `sample-keystore` machine on port `8500` with the following command:

```
$ docker $(docker-machine config sample-keystore) run -d \
    -p "8500:8500" \
    -h "consul" \
    progrium/consul -server –bootstrap
Unable to find image 'progrium/consul:latest' locally
latest: Pulling from progrium/consul
3b4d28ce80e4: Pull complete
e5ab901dcf2d: Pull complete
30ad296c0ea0: Pull complete
```

```
3dba40dec256: Pull complete
f2ef4387b95e: Pull complete
53bc8dcc4791: Pull complete
75ed0b50ba1d: Pull complete
17c3a7ed5521: Pull complete
8aca9e0ecf68: Pull complete
4d1828359d36: Pull complete
46ed7df7f742: Pull complete
b5e8ce623ef8: Pull complete
049dca6ef253: Pull complete
bdb608bc4555: Pull complete
8b3d489cfb73: Pull complete
c74500bbce24: Pull complete
9f3e605442f6: Pull complete
d9125e9e799b: Pull complete
Digest:
sha256:8cc8023462905929df9a79ff67ee435a36848ce7a10f18d6d0faba9
306b97274
Status: Downloaded newer image for progrium/consul:latest
1a1be5d207454a54137586f1211c02227215644fa0e36151b000cfcde3b0df
7c
```

3. Set the local environment to the `sample-keystore` machine:

```
$ eval "$(docker-machine env sample-keystore)"
```

4. We can list the consul container as follows:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
   STATUS          PORTS
                                                  NAMES
1a1be5d20745   progrium/consul   /bin/start -server   5 minutes ago
 Up 5 minutes    53/tcp, 53/udp, 8300-8302/tcp, 8400/tcp, 8301-
8302/udp, 0.0.0.0:8500->8500/tcp    cocky_bhaskara
```

5. Create a Swarm cluster using `docker-machine`. The two machines can be created in VirtualBox; one can act as the Swarm master. As we create each Swarm node, we will be passing the options required for Docker Engine to have an overlay network driver:

```
$ docker-machine create -d virtualbox --swarm --swarm-
image="swarm" --swarm-master --swarm-
discovery="consul://$(docker-machine ip sample-keystore):8500"
```

```
--engine-opt="cluster-store=consul://$(docker-machine ip
sample-keystore):8500" --engine-opt="cluster-
advertise=eth1:2376" swarm-master

Running pre-create checks...

Creating machine...

Waiting for machine to be running, this may take a few
minutes...

Machine is running, waiting for SSH to be available...

Detecting operating system of created instance...

Provisioning created instance...

Copying certs to the local machine directory...

Copying certs to the remote machine...

Setting Docker configuration on the remote daemon...

Configuring swarm...

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

The use of the parameters used in the preceding command is as follows:

- `--swarm`: This is used to configure a machine with Swarm.
- `--engine-opt`: This option is used to define arbitrary daemon options required to be supplied. In our case, we will supply the engine daemon with the `--cluster-store` option during creation time, which tells the engine the location of the key-value store for the overlay network usability. The `--cluster-advertise` option will put the machine on the network at the specific port.
- `--swarm-discovery`: It is used to discover services to use with Swarm, in our case, `consul` will be that service.
- `--swarm-master`: This is used to configure a machine as the Swarm master.

6. Another host can also be created and added to Swarm cluster, like this:

```
$ docker-machine create -d virtualbox --swarm --swarm-
image="swarm:1.0.0-rc2" --swarm-discovery="consul://$(docker-
machine ip sample-keystore):8500" --engine-opt="cluster-
store=consul://$(docker-machine ip sample-keystore):8500" --
engine-opt="cluster-advertise=eth1:2376" swarm-node-1

Running pre-create checks...

Creating machine...

Waiting for machine to be running, this may take a few
minutes...

Machine is running, waiting for SSH to be available...
```

```
Detecting operating system of created instance...
Provisioning created instance...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Configuring swarm...
To see how to connect Docker to this machine, run: docker-
machine env swarm-node-1
```

7. The machines can be listed as follows:

```
$ docker-machine ls
NAME             ACTIVE   DRIVER       STATE     URL
  SWARM
sample-keystore   -     virtualbox   Running
tcp://192.168.99.100:2376
swarm-master      -     virtualbox   Running
tcp://192.168.99.101:2376  swarm-master (master)
swarm-node-1      -     virtualbox   Running
tcp://192.168.99.102:2376   swarm-master
```

8. Now, we will set the Docker environment to `swarm-master`:

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

9. The following command can be executed on the master in order to create the overlay network and have multihost networking:

```
$ docker network create -driver overlay sample-net
```

10. The network bridge can be checked on the master using the following command:

```
$ docker network ls
NETWORK ID          NAME            DRIVER
9f904ee27bf5      sample-net      overlay
7fca4eb8c647       bridge         bridge
b4234109be9b       none            null
cf03ee007fb4       host            host
```

11. When switching to a Swarm node, we can easily list the newly created overlay network, like this:

```
$ eval $(docker-machine env swarm-node-1)
$ docker network ls
NETWORK ID        NAME            DRIVER
```

```
7fca4eb8c647        bridge              bridge
b4234109be9b        none                null
cf03ee007fb4        host                host
9f904ee27bf5      sample-net            overlay
```

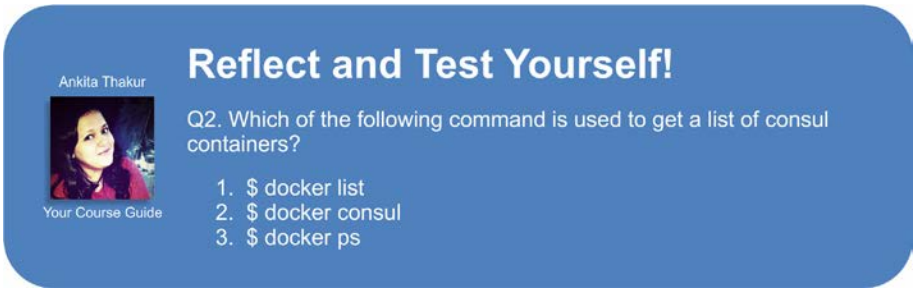
12. Once the network is created, we can start the container on any of the hosts, and it will be part of the network:

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

13. Start the sample `ubuntu` container with the constraint environment set to the first node:

```
$ docker run -itd --name=os --net=sample-net --
env="constraint:node==swarm-master" ubuntu
```

14. We can check using the `ifconfig` command that the container has two network interfaces, and it will be accessible from the container deployed using Swarm manager on any other host.



Kubernetes

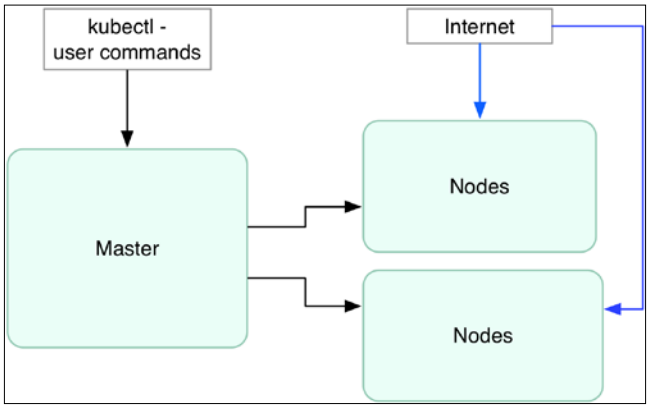
Kubernetes is a container cluster management tool. Currently, it supports Docker and Rocket. It is an open source project supported by Google, and the project was launched in June 2014 at Google I/O. It supports deployment on various cloud providers such as GCE, Azure, AWS, and vSphere as well as on bare metal. The Kubernetes manager is lean, portable, extensible, and self-healing.

Kubernetes has various important components, as explained in the following list:

- **Node**: This is a physical or virtual-machine part of a Kubernetes cluster, running the Kubernetes and Docker services, onto which pods can be scheduled.

- **Master**: This maintains the runtime state of the Kubernetes server runtime. It is the point of entry for all the client calls to configure and manage Kubernetes components.
- **Kubectl**: This is the command-line tool used to interact with the Kubernetes cluster to provide master access to Kubernetes APIs. Through it, the user can deploy, delete, and list pods.
- **Pod**: This is the smallest scheduling unit in Kubernetes. It is a collection of Docker containers that share volumes and don't have port conflicts. It can be created by defining a simple JSON file.
- **Replication controller**: It manages the lifecycle of a pod and ensures that a specified number of pods are running at a given time by creating or killing pods as required.
- **Label**: Labels are used to identify and organize pods and services based on key-value pairs.

The following diagram shows the Kubernetes Master/Minion flow:



Deploying Kubernetes on AWS

Let's get started with Kubernetes cluster deployment on AWS, which can be done by using the config file that already exists in the Kubernetes codebase:

1. Log in to AWS Console at `http://aws.amazon.com/console/`.
2. Open the IAM console at `https://console.aws.amazon.com/iam/home?#home`.
3. Choose the IAM username, select the **Security Credentials** tab, and click on the **Create Access Key** option.

4. After the keys have been created, download and keep them in a secure place. The downloaded `.csv` file will contain an `Access Key ID` and `Secret Access Key`, which will be used to configure the AWS CLI.
5. Install and configure the AWS CLI. In this example, we have installed AWS CLI on Linux using the following command:

```
$ sudo pip install awscli
```

6. In order to configure the AWS CLI, use the following command:

```
$ aws configure
AWS Access Key ID [None]: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
AWS Secret Access Key [None]: YYYYYYYYYYYYYYYYYYYYYYYYYYYY
Default region name [None]: us-east-1
Default output format [None]: text
```

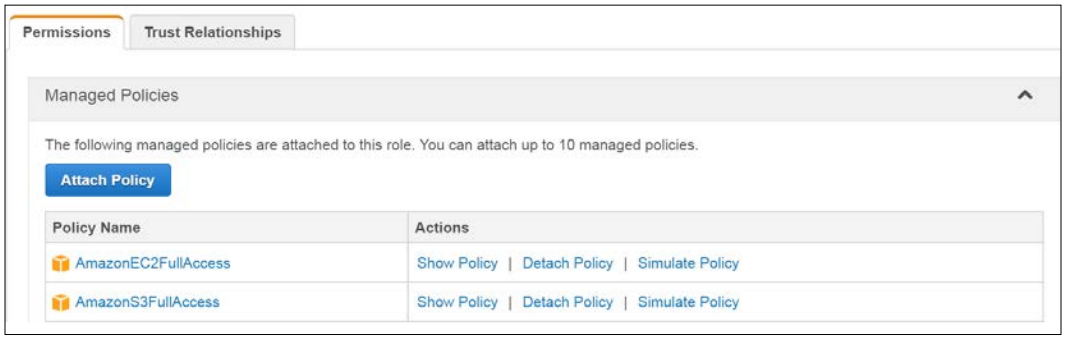
7. After configuring the AWS CLI, we will create a profile and attach a role to it with full access to S3 and EC2:

```
$ aws iam create-instance-profile --instance-profile-name Kube
```

8. The role can be created separately using the console or AWS CLI with a JSON file that defines the permissions the role can have:

```
$ aws iam create-role --role-name Test-Role --assume-role-
policy-document /root/kubernetes/Test-Role-Trust-Policy.json
```

   A role can be attached to the preceding profile, which will have complete access to EC2 and S3, as shown in the following screenshot:



9. After the creation of the role, it can be attached to a policy using the following command:

```
$ aws iam add-role-to-instance-profile --role-name Test-Role -
-instance-profile-name Kube
```

10. By default, the script uses the default profile. We can change it as follows:

```
$ export AWS_DEFAULT_PROFILE=Kube
```

11. The Kubernetes cluster can be easily deployed using one command, as follows:

```
$ export KUBERNETES_PROVIDER=aws; wget -q -O -
https://get.k8s.io | bash

Downloading kubernetes release v1.1.1 to
/home/vkohli/kubernetes.tar.gz

--2015-11-22 10:39:18--
https://storage.googleapis.com/kubernetes-
release/release/v1.1.1/kubernetes.tar.gz

Resolving storage.googleapis.com (storage.googleapis.com)...
216.58.220.48, 2404:6800:4007:805::2010

Connecting to storage.googleapis.com
(storage.googleapis.com)|216.58.220.48|:443... connected.

HTTP request sent, awaiting response... 200 OK

Length: 191385739 (183M) [application/x-tar]

Saving to: 'kubernetes.tar.gz'

100%[======================================>] 191,385,739
1002KB/s   in 3m 7s

2015-11-22 10:42:25 (1002 KB/s) - 'kubernetes.tar.gz' saved
[191385739/191385739]

Unpacking kubernetes release v1.1.1

Creating a kubernetes on aws...

... Starting cluster using provider: aws

... calling verify-prereqs

... calling kube-up

Starting cluster using os distro: vivid

Uploading to Amazon S3

Creating kubernetes-staging-e458a611546dc9dc0f2a2ff2322e724a

make_bucket: s3://kubernetes-staging-
e458a611546dc9dc0f2a2ff2322e724a/

+++ Staging server tars to S3 Storage: kubernetes-staging-
e458a611546dc9dc0f2a2ff2322e724a/devel

upload: ../../../tmp/kubernetes.6B8Fmm/s3/kubernetes-
salt.tar.gz to s3://kubernetes-staging-
e458a611546dc9dc0f2a2ff2322e724a/devel/kubernetes-salt.tar.gz

Completed 1 of 19 part(s) with 1 file(s) remaining
```

12. The preceding command will call `kube-up.sh` and, in turn, `utils.sh` using the `config-default.sh` script, which contains the basic configuration of a K8S cluster with four nodes, as follows:

```
ZONE=${KUBE_AWS_ZONE:-us-west-2a}
MASTER_SIZE=${MASTER_SIZE:-t2.micro}
MINION_SIZE=${MINION_SIZE:-t2.micro}
NUM_MINIONS=${NUM_MINIONS:-4}
AWS_S3_REGION=${AWS_S3_REGION:-us-east-1}
```

13. The instances are `t2.micro` running Ubuntu OS. The process takes 5 to 10 minutes, after which the IP addresses of the master and minions get listed and can be used to access the Kubernetes cluster.

Kubernetes networking and its differences to Docker networking

Kubernetes strays from the default Docker system's networking model. The objective is for each pod to have an IP at a level imparted by the system's administration namespace, which has full correspondence with other physical machines and containers over the system. Allocating IPs per pod unit makes for a clean, retrogressive, and good model where units can be dealt with much like VMs or physical hosts from the point of view of port allotment, system administration, naming, administration disclosure, burden adjustment, application design, and movement of pods from one host to another. All containers in all pods can converse with all other containers in all other pods using their addresses. This also helps move traditional applications to a container-oriented approach.

As every pod gets a real IP address, they can communicate with each other without any need for translation. By making the same configuration of IP addresses and ports both inside as well as outside of the pod, we can create a NAT-less flat address space. This is different from the standard Docker model since there, all containers have a private IP address, which will allow them to be able to access the containers on the same host. But in the case of Kubernetes, all the containers inside a pod behave as if they are on the same host and can reach each other's ports on the localhost. This reduces the isolation between containers and provides simplicity, security, and performance. Port conflict can be one of the disadvantages of this; thus, two different containers inside one pod cannot use the same port.

In GCE, using IP forwarding and advanced routing rules, each VM in a Kubernetes cluster gets an extra 256 IP addresses in order to route traffic across pods easily.

Routes in GCE allow you to implement more advanced networking functions in the VMs, such as setting up many-to-one NAT. This is leveraged by Kubernetes.

This is in addition to the main Ethernet bridge which the VM has; this bridge is termed as the container bridge `cbr0` in order to differentiate it from the Docker bridge, `docker0`. In order to transfer packets out of the GCE environment from a pod, it should undergo an SNAT to the VM's IP address, which GCE recognizes and allows.

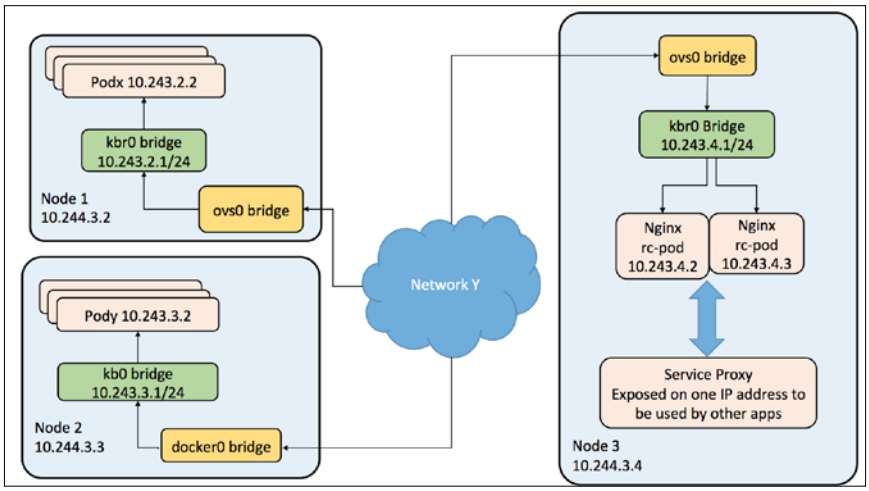
Other implementations with the primary aim of providing an IP-per-pod model are Open vSwitch, Flannel, and Weave.

In the case of a GCE-like setup of an Open vSwitch bridge for Kubernetes, the model where the Docker bridge gets replaced by `kbr0` to provide an extra 256 subnet addresses is followed. Also, an OVS bridge (`ovs0`) is added, which adds a port to the Kubernetes bridge in order to provide GRE tunnels to transfer packets across different minions and connect pods residing on these hosts. The IP-per-pod model is also elaborated more in the upcoming diagram, where the service abstraction concept of Kubernetes is also explained.

A service is another type of abstraction that is widely used and suggested for use in Kubernetes clusters as it allows a group of pods (applications) to be accessed via virtual IP addresses and gets proxied to all internal pods in a service. An application deployed in Kubernetes could be using three replicas of the same pod, which have different IP addresses. However, the client can still access the application on the one IP address which is exposed outside, irrespective of which backend pod takes the request. A service acts as a load balancer between different replica pods and a single point of communication for clients utilizing this application. Kubeproxy, one of the services of Kubernetes, provides load balancing and uses rules to access the service IPs and redirects them to the correct backend pod.

Deploying the Kubernetes pod

Now, in the following example, we will be deploying two nginx replication pods (`rc-pod`) and exposing them via a service in order to understand Kubernetes networking. Deciding where the application can be exposed via a virtual IP address and which replica of the pod (load balancer) the request is to be proxied to is taken care of by **Service Proxy**. Please refer to the following diagram for more details:



The following are the steps to deploy the Kubernetes pod:

1. In the Kubernetes master, create a new folder:

   ```
   $ mkdir nginx_kube_example
   $ cd nginx_kube_example
   ```

2. In the editor of your choice, create the `.yaml` file that will be used to deploy the nginx pods:

   ```
   $ vi nginx_pod.yaml
   ```

   Copy the following into the file:

   ```
   apiVersion: v1
   kind: ReplicationController
   metadata:
     name: nginx
   spec:
     replicas: 2
   ```

```
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
  spec:
    containers:
    - name: nginx
      image: nginx
      ports:
      - containerPort: 80
```

3. Create the nginx pod using `kubectl`:

```
$ kubectl create -f nginx_pod.yaml
```

4. In the preceding pod creation process, we created two replicas of the nginx pod, and its details can be listed using the following command:

```
$ kubectl get pods
```

   The following is the output generated:

```
NAME          READY     REASON    RESTARTS   AGE
nginx-karne   1/1       Running   0          14s
nginx-mo5ug   1/1       Running   0          14s
```

   To list replication controllers on a cluster, use the `kubectl get` command:

```
$ kubectl get rc
```

   The following is the output generated:

```
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR    REPLICAS
nginx        nginx          nginx      app=nginx   2
```

5. The container on the deployed minion can be listed using the following command:

```
$ docker ps
```

   The following is the output generated:

```
CONTAINER ID        IMAGE                                  COMMAND
CREATED             STATUS              PORTS              NAMES
```

```
1d3f9cedff1d        nginx:latest                              "nginx
-g 'daemon of   41 seconds ago      Up 40 seconds
                    k8s_nginx.6171169d_nginx-karne_default
_5d5bc813-3166-11e5-8256-ecf4bb2bbd90_886ddf56
0b2b03b05a8d        nginx:latest
"nginx -g 'daemon of   41 seconds ago      Up 40 seconds
```

6. Deploy the nginx service using the following `.yaml` file in order to expose the nginx pod on host port `82`:

```
$ vi nginx_service.yaml
```

Copy the following into the file:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: nginxservice
  name: nginxservice
spec:
  ports:
    # The port that this service should serve on.
    - port: 82
  # Label keys and values that must match in order to receive
traffic for this service.
  selector:
    app: nginx
  type: LoadBalancer
```

7. Create the nginx service using the `kubectl create` command:

```
$kubectl create -f nginx_service.yaml
services/nginxservice
```

8. The nginx service can be listed using the following command:

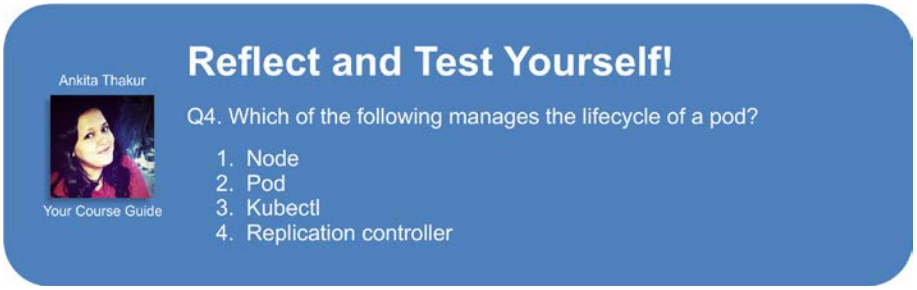
```
$ kubectl get services
```

The following is the output generated:

```
NAME           LABELS
SELECTOR    IP(S)          PORT(S)
kubernetes     component=apiserver,provider=kubernetes
<none>      192.168.3.1    443/TCP
nginxservice   name=nginxservice
app=nginx   192.168.3.43   82/TCP
```

9. Now, the nginx server's test page can be accessed on the following URL via the service:

```
http://192.168.3.43:82
```



Mesosphere

Mesosphere is a software solution that provides ways of managing server infrastructures and basically expands upon the cluster-management capabilities of Apache Mesos. Mesosphere has also launched the **DCOS** (**data center operating system**), used to manage data centers by spanning all the machines and treating them as a single computer, providing a highly scalable and elastic way of deploying apps on top of it. DCOS can be installed on any public cloud or your own private data center, ranging from AWS, GCE, and Microsoft Azure to VMware. Marathon is the framework for Mesos and is designed to launch and run applications; it serves as a replacement for the init system. Marathon provides various features such as high availability, application health check, and service discovery, which help you run applications in Mesos clustered environments.

This session describes how to bring up a single-node Mesos cluster.

Docker containers

Mesos can run and manage Docker containers using the Marathon framework.

In this exercise, we will use CentOS 7 to deploy a Mesos cluster:

1. Install Mesosphere and Marathon using the following command:

```
# sudo rpm -Uvh
http://repos.mesosphere.com/el/7/noarch/RPMS/mesosphere-el-
repo-7-1.noarch.rpm
```

```
# sudo yum -y install mesos marathon
```

Apache Mesos uses Zookeeper to operate. Zookeeper acts as the master election service in the Mesosphere architecture and stores states for the Mesos nodes.

2. Install Zookeeper and the Zookeeper server package by pointing to the RPM repository for Zookeeper, as follows:

```
# sudo rpm -Uvh http://archive.cloudera.com/cdh4/one-click-install/redhat/6/x86_64/cloudera-cdh-4-0.x86_64.rpm
# sudo yum -y install zookeeper zookeeper-server
```

3. Validate Zookeeper by stopping and restarting it:

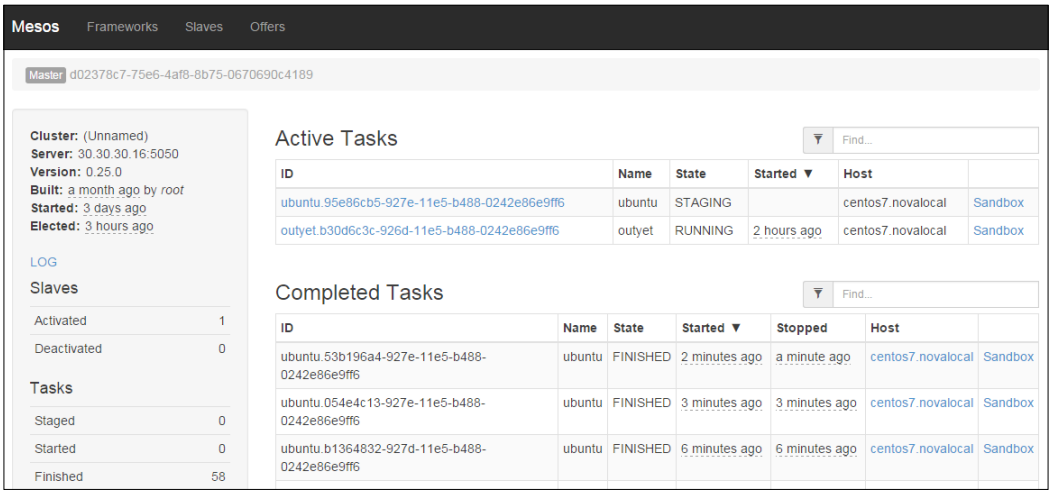
```
# sudo service zookeeper-server stop
# sudo service zookeeper-server start
```

Mesos uses a simple architecture to give you intelligent task distribution across a cluster of machines without worrying about where they are scheduled.

4. Configure Apache Mesos by starting the `mesos-master` and `mesos-slave` processes as follows:

```
# sudo service mesos-master start
# sudo service mesos-slave start
```

5. Mesos will be running on port `5050`. As shown in the following screenshot, you can access the Mesos interface with your machine's IP address, here, `http://192.168.10.10:5050`:



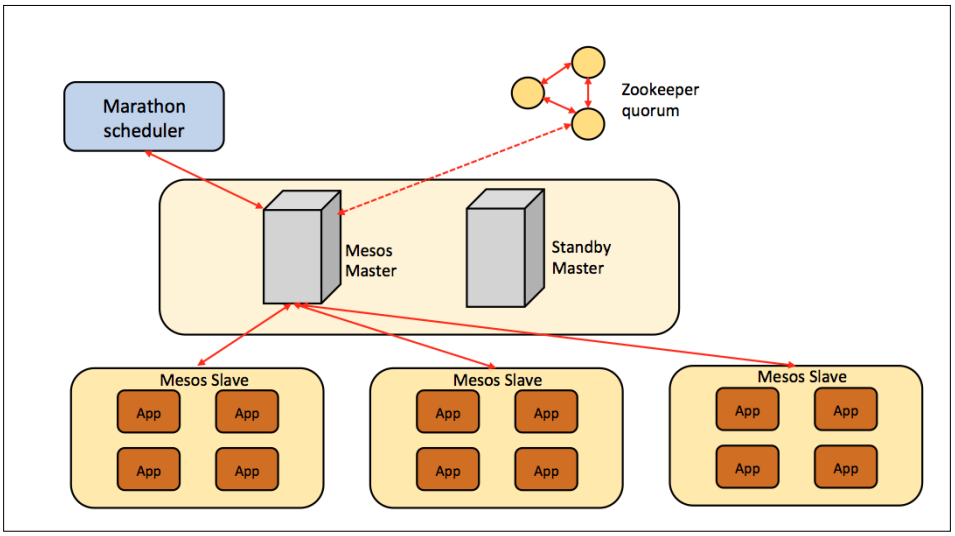
6. Test Mesos using the `mesos-execute` command:

```
# export MASTER=$(mesos-resolve `cat /etc/mesos/zk` 
2>/dev/null)

# mesos help

# mesos-execute --master=$MASTER --name="cluster-test" --
command="sleep 40"
```

7. With the `mesos-execute` command running, enter *Ctrl* + *Z* to suspend the command. You can see how it appears in the web UI and command line:

```
# hit ctrl-z

# mesos ps --master=$MASTER
```

   The Mesosphere stack uses Marathon to manage processes and services. It serves as a replacement for the traditional init system. It simplifies the running of applications in a clustered environment. The following figure shows the Mesosphere Master slave topology with Marathon:



   Marathon can be used to start other Mesos frameworks; as it is designed for long-running applications, it will ensure that the applications it has launched will continue running even if the slave nodes they are running on fail.

8. Start the Marathon service using the following command:

```
# sudo service marathon start
```

   You can view the Marathon GUI at `http://192.168.10.10:8080`.

Deploying a web app using Docker

In this exercise, we will install a simple Outyet web application:

1. Install Docker using the following commands:

```
# sudo yum install -y golang git device-mapper-event-libs
docker
# sudo chkconfig docker on
# sudo service docker start
# export GOPATH=~/go
# go get github.com/golang/example/outyet
# cd $GOPATH/src/github.com/golang/example/outyet
# sudo docker build -t outyet.
```

2. The following command tests the Docker file before adding it to Marathon:

```
# sudo docker run --publish 6060:8080 --name test --rm outyet
```

3. Go to `http://192.168.10.10:6060/` on your browser in order to confirm it works. Once it does, you can hit *CTRL* + *C* to exit the Outyet Docker.
4. Create a Marathon application using Marathon Docker support, as follows:

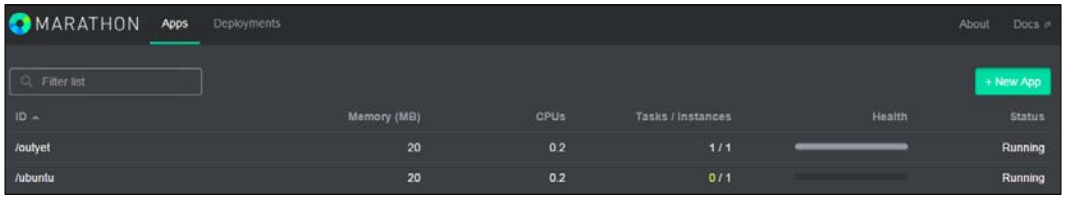
```
# vi /home/user/outyet.json
{
   "id": "outyet",
   "cpus": 0.2,
   "mem": 20.0,
   "instances": 1,
   "constraints": [["hostname", "UNIQUE", ""]],
   "container": {
      "type": "DOCKER",
      "docker": {
         "image": "outyet",
         "network": "BRIDGE",
         "portMappings": [ { "containerPort": 8080,
           "hostPort": 0, "servicePort": 0, "protocol": "tcp" }
        ]
     }
   }
}
```

```
# echo 'docker,mesos' | sudo tee /etc/mesos-slave/containerizers
# sudo service mesos-slave restart
```

5. Containers are configured and managed better with Marathon Docker, as follows:

```
# curl -X POST http://192.168.10.10:8080/v2/apps -d
/home/user/outyet.json -H "Content-type: application/json"
```

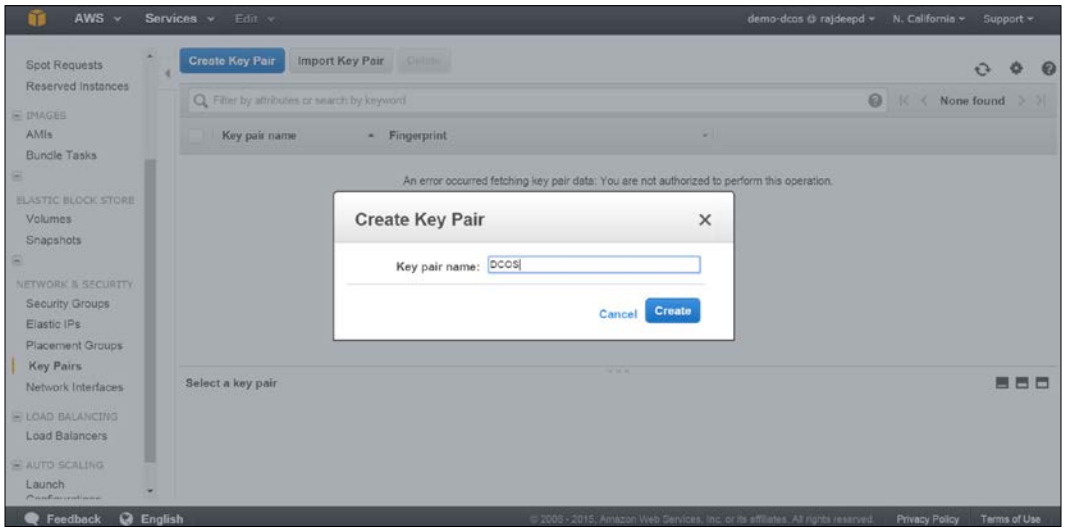
6. You can check all your applications on the Marathon GUI at `http://192.168.10.10:8080`, as shown in the following screenshot:



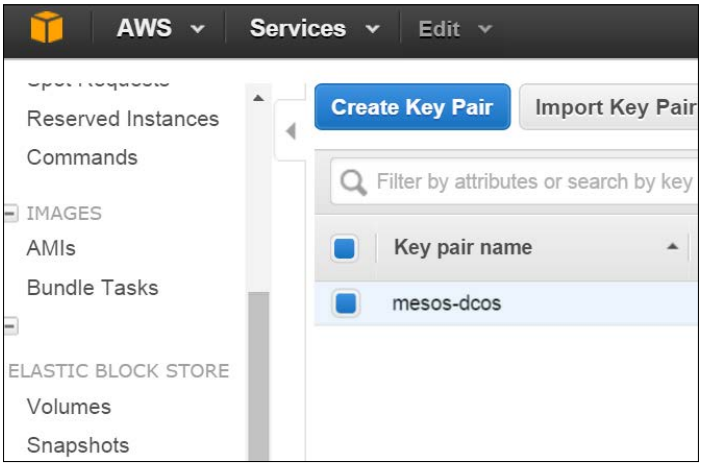
Deploying Mesos on AWS using DCOS

In this final section, we will be deploying the latest launch of DCOS by Mesosphere on AWS in order to manage and deploy Docker services in our data center:

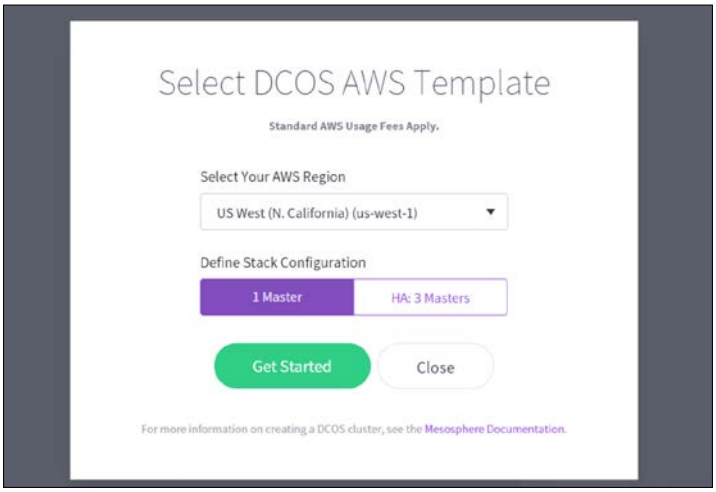
1. Create an AWS key pair in the region where the cluster is required to be deployed by going to the navigation pane and choosing **Key Pairs** under **NETWORK & SECURITY**:



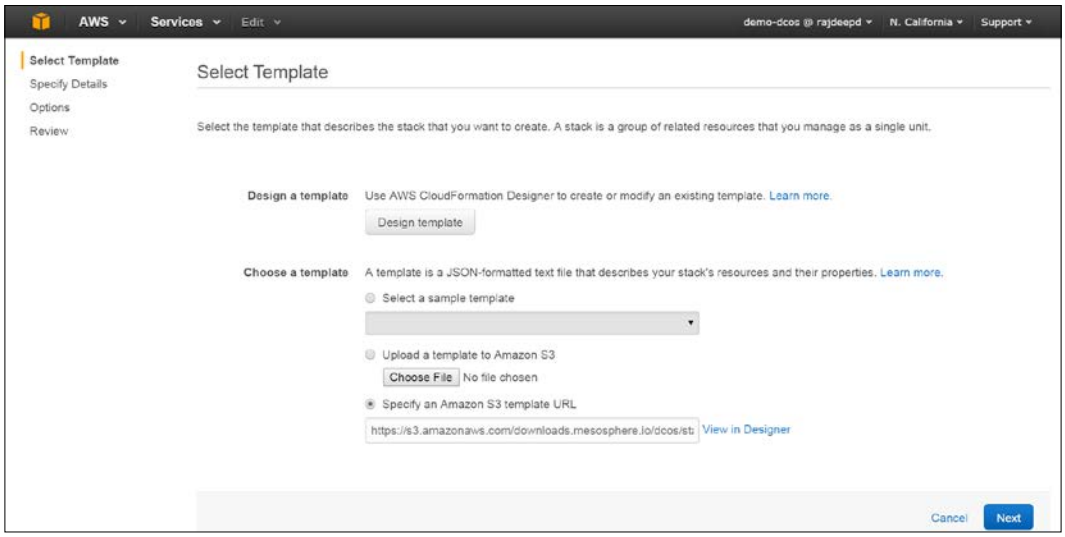
2. After being created, the key can be viewed as follows and the generated key pair (`.pem`) file should be stored in a secure location for future use:



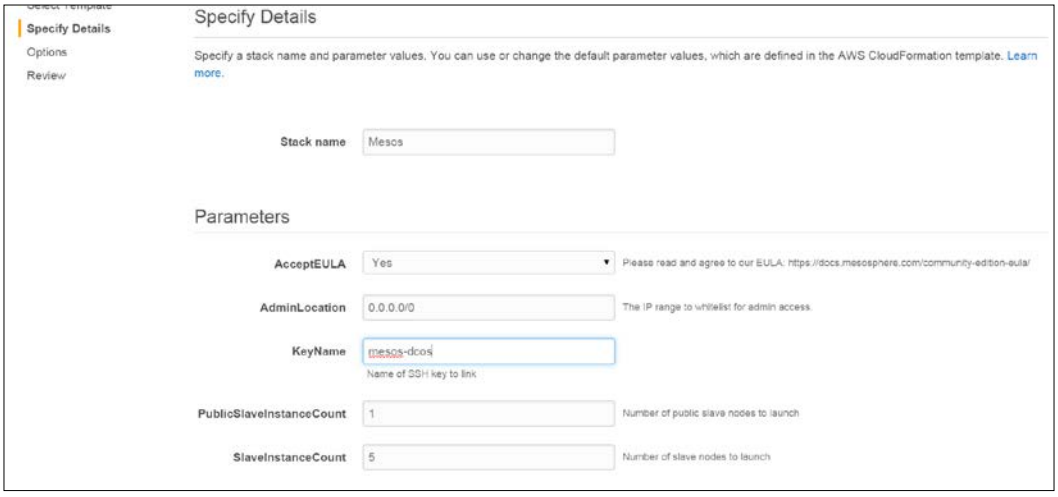
3. The DCOS cluster can be created by selecting the **1 Master** template on the official Mesosphere site:



It can also be done by providing the link for the Amazon S3 template URL in the stack deployment:

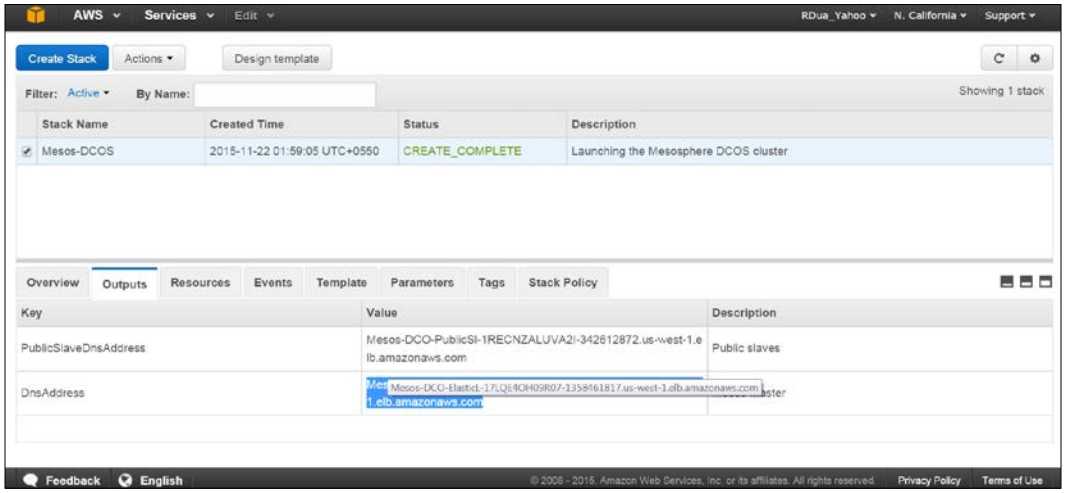


4. Click on the **Next** button. Fill in the details such as **Stack name** and **KeyName**, generated in the previous step:



5. Review the details before clicking on the **Create** button:

6. After 5 to 10 minutes, the Mesos stack will be deployed and the Mesos UI can be accessed at the URL shown in the following screenshot:



7. Now, we will be installing the DCOS CLI on a Linux machine with Python (2.7 or 3.4) and pip preinstalled, using the following commands:

```
$ sudo pip install virtualenv
$ mkdir dcos
$ cd dcos
$ curl -O https://downloads.mesosphere.io/dcos-cli/install.sh
% Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                 Dload  Upload   Total   Spent
Left  Speed
100  3654  100  3654    0     0   3631      0  0:00:01  0:00:01
--:--:--  3635
$ ls
install.sh
$ bash install.sh . http://mesos-dco-elasticl-17lqe4oh09r07-
1358461817.us-west-1.elb.amazonaws.com
Installing DCOS CLI from PyPI...
New python executable in /home/vkohli/dcos/bin/python
Installing setuptools, pip, wheel...done.
[core.reporting]: set to 'True'
```

```
[core.dcos_url]: set to 'http://mesos-dco-elasticl-
17lqe4oh09r07-1358461817.us-west-1.elb.amazonaws.com'

[core.ssl_verify]: set to 'false'

[core.timeout]: set to '5'

[package.cache]: set to '/home/vkohli/.dcos/cache'

[package.sources]: set to '[u'https://github.com/mesosphere/
universe/archive/version-1.x.zip']'

Go to the following link in your browser:

https://accounts.mesosphere.com/oauth/authorize?scope=&redirect_ur
i=urn%3Aietf%3Awg%3Aoauth%3A2.0%3Aoob&response_type=code&client_
id=6a552732-ab9b-410d-9b7d-d8c6523b09a1&access_type=offline

Enter verification code: Skipping authentication.

Enter email address: Skipping email input.

Updating source [https://github.com/mesosphere/universe/archive/
version-
1.x.zip]

Modify your bash profile to add DCOS to your PATH? [yes/no]
yes

Finished installing and configuring DCOS CLI.

Run this command to set up your environment and to get
started:

source ~/.bashrc && dcos help
```

The DCOS help file can be listed as follows:

```
$ source ~/.bashrc && dcos help

Command line utility for the Mesosphere Datacenter Operating
System (DCOS). The Mesosphere DCOS is a distributed operating
system built around Apache Mesos. This utility provides tools
for easy management of a DCOS installation.

Available DCOS commands:

        config       Get and set DCOS CLI configuration properties
        help         Display command line usage information
        marathon     Deploy and manage applications on the DCOS
        node         Manage DCOS nodes
        package      Install and manage DCOS packages
        service      Manage DCOS services
        task         Manage DCOS tasks
```

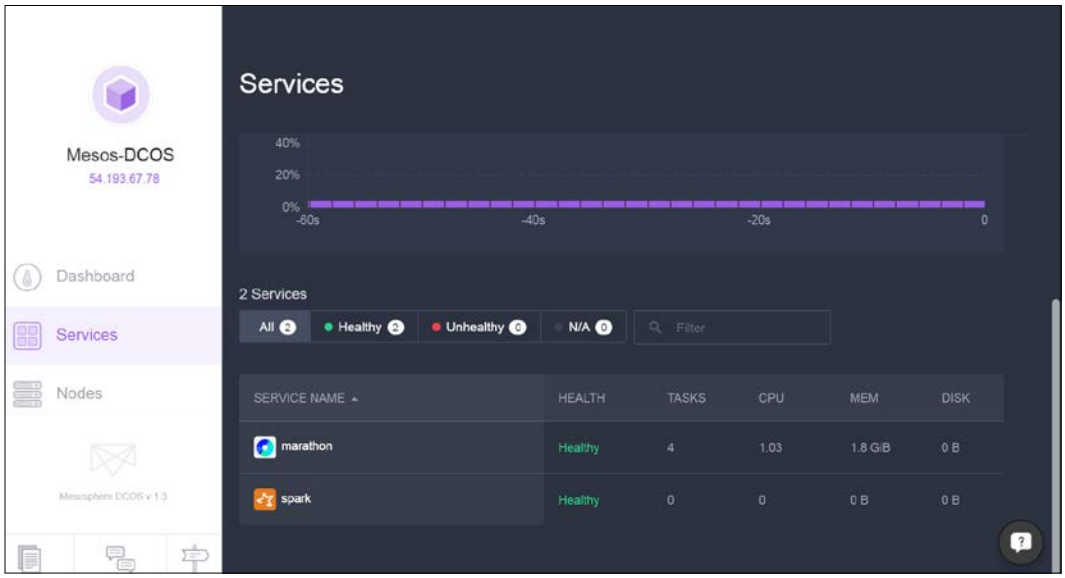
8. Now, we will deploy a Spark application on top of the Mesos cluster using the DCOS package after updating it. Get a detailed command description with `dcos <command> --help`:

```
$ dcos config show package.sources
[
  "https://github.com/mesosphere/universe/archive/version-
1.x.zip"
]
$ dcos package update
Updating source
[https://github.com/mesosphere/universe/archive/version-
1.x.zip]


$ dcos package search
NAME       VERSION            FRAMEWORK     SOURCE
                                                    DESCRIPTION
arangodb   0.2.1                True
https://github.com/mesosphere/universe/archive/version-1.x.zip
A distributed free and open-source database with a flexible
data model for documents, graphs, and key-values. Build high
performance applications using a convenient SQL-like query
language or JavaScript extensions.
cassandra  0.2.0-1              True
https://github.com/mesosphere/universe/archive/version-1.x.zip
Apache Cassandra running on Apache Mesos.
chronos    2.4.0                True
https://github.com/mesosphere/universe/archive/version-1.x.zip
A fault tolerant job scheduler for Mesos which handles
dependencies and ISO8601 based schedules.
hdfs       0.1.7                True
https://github.com/mesosphere/universe/archive/version-1.x.zip
Hadoop Distributed File System (HDFS), Highly Available.
kafka      0.9.2.0              True
https://github.com/mesosphere/universe/archive/version-1.x.zip
Apache Kafka running on top of Apache Mesos.
marathon   0.11.1               True
https://github.com/mesosphere/universe/archive/version-1.x.zip
A cluster-wide init and control system for services in cgroups
or Docker containers.
spark      1.5.0-multi-roles-v2  True
https://github.com/mesosphere/universe/archive/version-1.x.zip
Spark is a fast and general cluster computing system for Big
Data.
```

9. The Spark package can be installed as follows:

```
$ dcos package install spark

Note that the Apache Spark DCOS Service is beta and there may
be bugs, incomplete features, incorrect documentation or other
discrepancies.

We recommend a minimum of two nodes with at least 2 CPU and
2GB of RAM available for the Spark Service and running a Spark
job.

Note: The Spark CLI may take up to 5min to download depending
on your connection.

Continue installing? [yes/no] yes

Installing Marathon app for package [spark] version [1.5.0-
multi-roles-v2]

Installing CLI subcommand for package [spark] version [1.5.0-
multi-roles-v2]
```

10. After deployment, it can be seen in the DCOS UI under the **Services** tab, as shown in the following screenshot:



11. In order to deploy a dummy Docker application on the preceding Marathon cluster, we can use the JSON file to define the container image, command to execute, and ports to be exposed after deployment:

```
$ nano definition.json
{
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "superguenter/demo-app"
    }
  },
  "cmd":  "python -m SimpleHTTPServer $PORT",
  "id": "demo",
  "cpus": 0.01,
  "mem": 256,
  "ports": [3000]
}
```

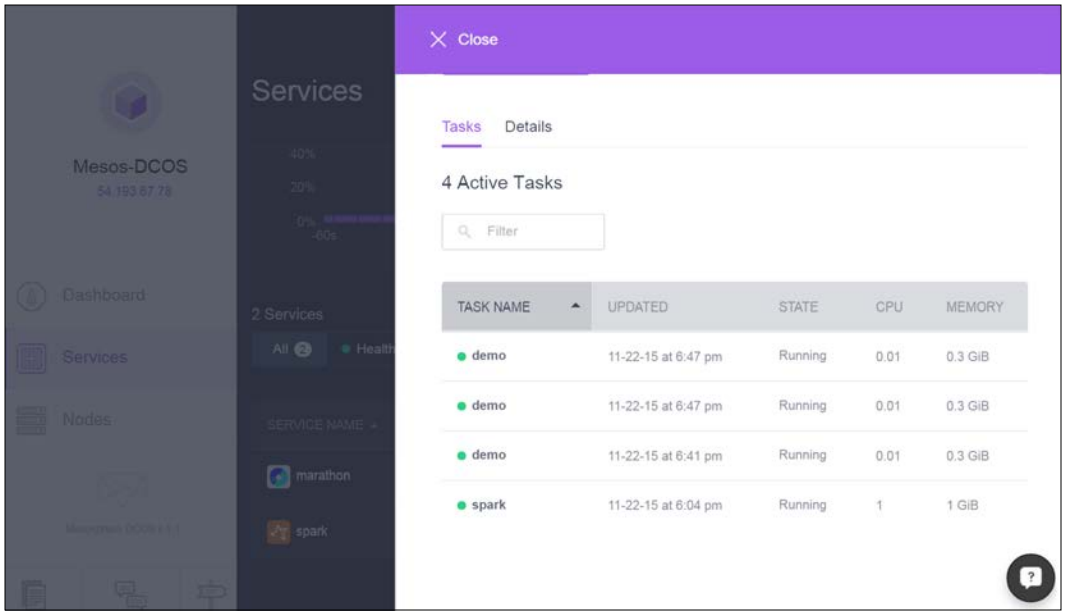
12. The app can be added to Marathon and listed as follows:

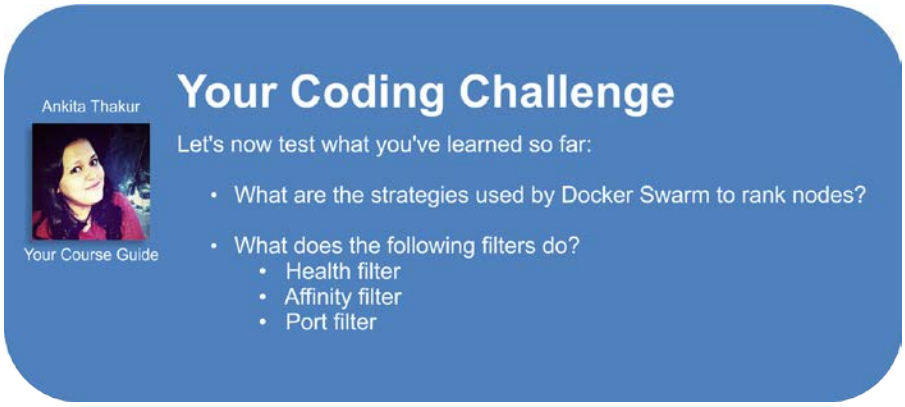
```
$ dcos marathon app add definition.json
$ dcos marathon app list
ID       MEM    CPUS  TASKS  HEALTH  DEPLOYMENT  CONTAINER  CMD
/demo   256.0   0.01   1/1    ---       ---        DOCKER   python
-m SimpleHTTPServer $PORT
/spark  1024.0  1.0    1/1    1/1       ---        DOCKER   mv
/mnt/mesos/sandbox/log4j.properties conf/log4j.properties &&
./bin/spark-class
org.apache.spark.deploy.mesos.MesosClusterDispatcher --port $PORT0
--webui-port $PORT1 --master mesos://zk://master.mesos:2181/mesos
--zk master.mesos:2181 --host $HOST --name spark
```

13. Three instances of the preceding Docker app can be started as follows:

```
$ dcos marathon app update --force demo instances=3
Created deployment 28171707-83c2-43f7-afa1-5b66336e36d7
$ dcos marathon deployment list
APP    ACTION  PROGRESS  ID
/demo  scale     0/1     28171707-83c2-43f7-afa1-5b66336e36d7
```

14. The deployed application can be seen in the DCOS UI by clicking on the **Tasks** tab under **Services**:





Summary of Module 2 Chapter 4

In this chapter, we learned about Docker networking using various frameworks, such as the native Docker Swarm. Using libnetwork or out-of-the-box overlay networks, Swarm provides multihost networking features.

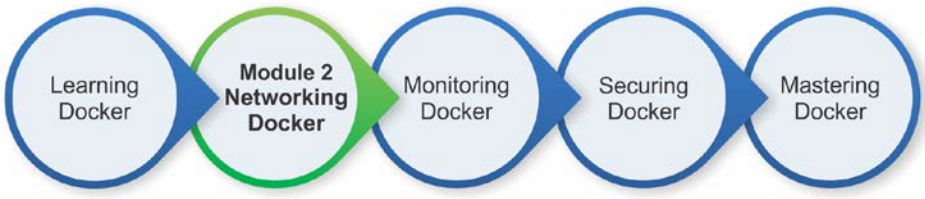
Ankita Thakur

Your Course Guide

Kubernetes, on the other hand, has a different perspective from Docker, in which each pod gets its unique IP address and communication between pods can occur with the help of services. Using Open vSwitch or IP forwarding and advanced routing rules, Kubernetes networking can be enhanced to provide connectivity between pods on different subnets across hosts and the ability to expose the pods to the external world. In the case of Mesosphere, we can see that Marathon is used as the backend for the networking of the deployed containers. In the case of DCOS by Mesosphere, the entire deployed stack of machines is treated as one machine in order to provide a rich networking experience between deployed container services.

In the next chapter, we will learn about tuning and troubleshooting in the Docker network using various tools.

Your Progress through the Course So Far



5
Next Generation Networking Stack for Docker – libnetwork

In this chapter, we will learn about a new networking stack for Docker – libnetwork, which provides a pluggable architecture with a default implementation for single and multi-host virtual networking:

- Introduction
    - Goal
    - Design
- CNM objects
    - CNM attributes
    - CNM lifecycle
- Drivers
    - Bridge driver
    - Overlay network driver
- Using overlay network with Vagrant
- Overlay network with Docker Machine and Docker Swarm
- Creating an overlay network manually and using it for containers
- Container network interface
- Calico's libnetwork driver

Goal

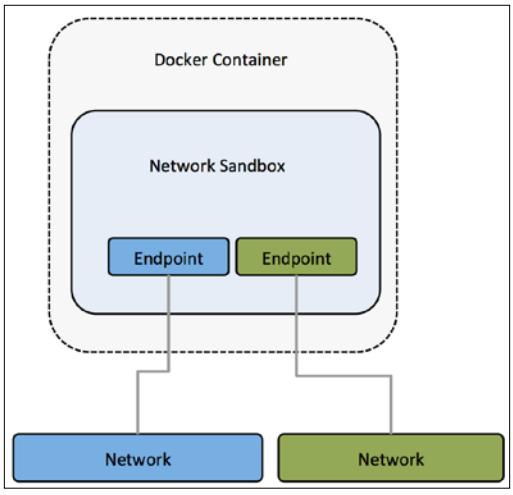
libnetwork which is written in go language is a new way for connecting Docker containers. The aim is to provide a container network model that helps programmers and provides the abstraction of network libraries. The long-term goal of libnetwork is to follow the Docker and Linux philosophy to deliver modules that work independently. libnetwork has the aim to provide a composable need for networking in containers. It also aims to modularize the networking logic in Docker Engine and libcontainer into a single, reusable library by:

- Replacing the networking module of Docker Engine with libnetwork
- Being a model that allows local and remote drivers to provide networking to containers
- Providing a tool dnet for managing and testing libnetwork—still a work in progress (reference from `https://github.com/docker/libnetwork/issues/45`).

Design

libnetwork implements a **container network model** (**CNM**). It formalizes the steps required to provide networking for containers, while providing an abstraction that can be used to support multiple network drivers. Its endpoint APIs are primarily used for managing the corresponding object and book-keeps them in order to provide a level of abstraction as required by the CNM model.

The CNM is built on three main components. The following figure shows the network sandbox model of libnetwork:



CNM objects

Let's discuss the CNM objects in detail.

Sandbox

This contains the configuration of a container's network stack, which includes management of routing tables, the container's interface, and DNS settings. An implementation of a sandbox can be a Linux network namespace, a FreeBSD jail, or other similar concept. A sandbox may contain many endpoints from multiple networks. It also represents a container's network configuration such as IP-address, MAC address, and DNS entries. libnetwork makes use of the OS-specific parameters to populate the network configuration represented by sandbox. libnetwork provides a framework to implement sandbox in multiple operating systems. Netlink is used to manage the routing table in namespace, and currently two implementations of sandbox exist, `namespace_linux.go` and `configure_linux.go`, to uniquely identify the path on the host filesystem.

A sandbox is associated with a single Docker container. The following data structure shows the runtime elements of a sandbox:

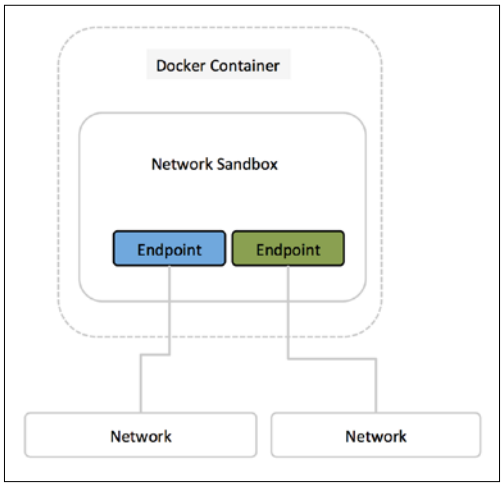
```
type sandbox struct {
  id            string
  containerID   string
  config        containerConfig
  osSbox        osl.Sandbox
  controller    *controller
  refCnt        int
  endpoints     epHeap
  epPriority    map[string]int
  joinLeaveDone chan struct{}
  dbIndex       uint64
  dbExists      bool
  isStub        bool
  inDelete      bool
  sync.Mutex
}
```

A new sandbox is instantiated from a network controller (which is explained in more detail later):

```
func (c *controller) NewSandbox(containerID string, options
...SandboxOption) (Sandbox, error) {
  …..
}
```

Endpoint

An endpoint joins a sandbox to the network and provides connectivity for services exposed by a container to the other containers deployed in the same network. It can be an internal port of Open vSwitch or a similar veth pair. An endpoint can belong to only one network but may only belong to one sandbox. An endpoint represents a service and provides various APIs to create and manage the endpoint. It has a global scope but gets attached to only one network, as shown in the following figure:



An endpoint is specified by the following data structure:

```
type endpoint struct {
  name          string
  id            string
  network       *network
  iface         *endpointInterface
  joinInfo      *endpointJoinInfo
  sandboxID     string
  exposedPorts  []types.TransportPort
  anonymous     bool
  generic       map[string]interface{}
  joinLeaveDone chan struct{}
  prefAddress   net.IP
  prefAddressV6 net.IP
  ipamOptions   map[string]string
  dbIndex       uint64
  dbExists      bool
  sync.Mutex
}
```

An endpoint is associated with a unique ID and name. It is attached to a network and a sandbox ID. It is also associated with an IPv4 and IPv6 address space. Each endpoint is associated with an `endpointInterface` struct.

Network

A network is a group of endpoints that are able to communicate with each other directly. It provides the required connectivity within the same host or multiple hosts, and whenever a network is created or updated, the corresponding driver is notified. An example is a VLAN or Linux bridge, which has a global scope within a cluster.

Networks are controlled from a network controller, which we will discuss in the next section. Every network has a name, address space, ID, and network type:

```
type network struct {
  ctrlr        *controller
  name         string
  networkType  string
  id           string
  ipamType     string
  addrSpace    string
  ipamV4Config []*IpamConf
  ipamV6Config []*IpamConf
  ipamV4Info   []*IpamInfo
  ipamV6Info   []*IpamInfo
  enableIPv6   bool
  postIPv6     bool
  epCnt        *endpointCnt
  generic      options.Generic
  dbIndex      uint64
  svcRecords   svcMap
  dbExists     bool
  persist      bool
  stopWatchCh  chan struct{}
  drvOnce      *sync.Once
  internal     bool
  sync.Mutex
}
```

Network controller

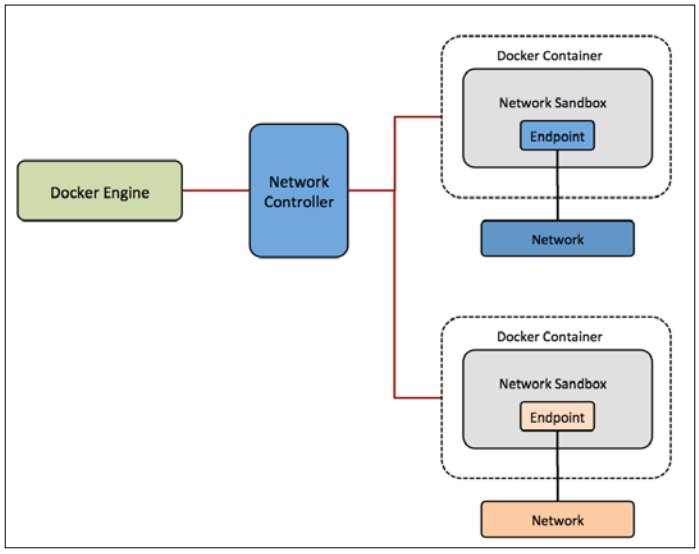
A network controller object provides APIs to create and manage a network object. It is an entry point in the libnetwork by binding a particular driver to a given network, and it supports multiple active drivers, both in-built and remote. Network controller allows users to bind a particular driver to a given network:

```
type controller struct {
  id              string
  drivers         driverTable
  ipamDrivers     ipamTable
  sandboxes       sandboxTable
  cfg             *config.Config
  stores          []datastore.DataStore
  discovery       hostdiscovery.HostDiscovery
  extKeyListener net.Listener
  watchCh         chan *endpoint
  unWatchCh       chan *endpoint
  svcDb           map[string]svcMap
  nmap            map[string]*netWatch
  defOsSbox       osl.Sandbox
  sboxOnce        sync.Once
  sync.Mutex
}
```

Each network controller has reference to the following:

- One or more drivers in the data structure driverTable
- One or more sandboxes in the data structure
- DataStore
- ipamTable

The following figure shows how **Network Controller** sits between the **Docker Engine** and the containers and networks they are attached to:



CNM attributes

There are two types of attributes, as follows:

- **Options**: They are not end-user visible but are the key-value pairs of data to provide a flexible mechanism to pass driver-specific configuration from user to driver directly. libnetwork operates on the options only if the key matches a well-known label as a result value is picked up, which is represented by a generic object.
- **Labels**: They are a subset of options that are end-user variables represented in the UI using the `-labels` option. Their main function is to perform driver-specific operations and they are passed from the UI.

CNM lifecycle

Consumers of the container network model interact through the CNM objects and its APIs to network the containers that they manage.

Drivers register with network controller. Built-in drivers register inside of libnetwork, while remote drivers register with libnetwork via a plugin mechanism (WIP). Each driver handles a particular network type.

A network controller object is created using the `libnetwork.New()` API to manage the allocation of networks and optionally configure a driver with driver-specific options.

The network is created using the controller's `NewNetwork()` API by providing a name and `networkType`. The `networkType` parameter helps to choose a corresponding driver and binds the created network to that driver. From this point, any operation on the network will be handled by that driver.

The `controller.NewNetwork()` API also takes in optional options parameters that carry driver-specific options and labels, which the drivers can make use for its purpose.

`network.CreateEndpoint()` can be called to create a new endpoint in a given network. This API also accepts optional options parameters that vary with the driver.

Drivers will be called with `driver.CreateEndpoint` and it can choose to reserve IPv4/IPv6 addresses when an endpoint is created in a network. The driver will assign these addresses using the `InterfaceInfo` interface defined in the `driver` API. The IPv4/IPv6 addresses are needed to complete the endpoint as a service definition along with the ports the endpoint exposes. A service endpoint is a network address and the port number that the application container is listening on.

`endpoint.Join()` can be used to attach a container to an endpoint. The `Join` operation will create a sandbox if it doesn't exist for that container. The drivers make use of the sandbox key to identify multiple endpoints attached to the same container.

There is a separate API to create an endpoint and another to join the endpoint.

An endpoint represents a service that is independent of the container. When an endpoint is created, it has resources reserved for the container to get attached to the endpoint later. It gives a consistent networking behavior.

`endpoint.Leave()` is invoked when a container is stopped. The driver can clean up the states that it allocated during the `Join()` call. libnetwork will delete the sandbox when the last referencing endpoint leaves the network.

libnetwork keeps holding on to IP addresses as long as the endpoint is still present. These will be reused when the container (or any container) joins again. It ensures that the container's resources are re-used when they are stopped and started again.

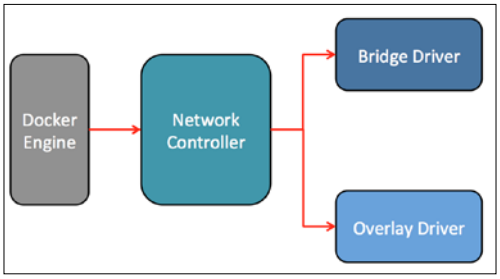
`endpoint.Delete()` is used to delete an endpoint from a network. This results in deleting the endpoint and cleaning up the cached `sandbox.Info`.

`network.Delete()` is used to delete a network. Delete is allowed if there are no endpoints attached to the network.

Driver

A driver owns a network and is responsible for making the network work and manages it. Network controller provides an API to configure the driver with specific labels/options that are not directly visible to the user but are transparent to libnetwork and can be handled by drivers directly. Drivers can be both in-built (such as bridge, host, or overlay) and remote (from plugin providers) to be deployed in various use cases and deployment scenarios.

The driver owns the network implementation and is responsible for managing it, including **IP Address Management (IPAM)**. The following figure explains the process:



The following are the in-built drivers:

- **Null**: In order to provide backward compatibility with old `docker --net=none`, this option exists primarily in the case when no networking is required.
- **Bridge**: It provides a Linux-specific bridging implementation driver.
- **Overlay**: The overlay driver implements networking that can span multiple hosts network encapsulation such as VXLAN. We will be doing a deep-dive on two of its implementations: basic setup with Consul and Vagrant setup to deploy the overlay driver.
- **Remote**: It provides a means of supporting drivers over a remote transport and a specific driver can be written as per choice.

Bridge driver

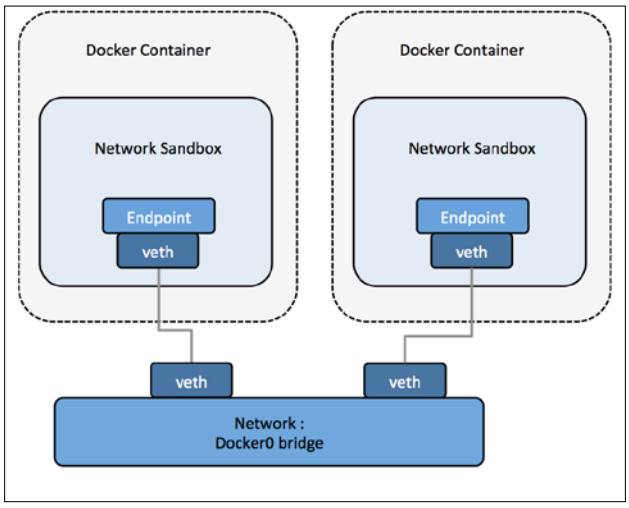
A bridge driver represents a wrapper on a Linux bridge acting as a network for libcontainer. It creates a veth pair for each network created. One end is connected to the container and the other end is connected to the bridge. The following data structure represents a bridge network:

```
type driver struct {
  config      *configuration
  etwork      *bridgeNetwork
  natChain    *iptables.ChainInfo
  filterChain *iptables.ChainInfo
  networks    map[string]*bridgeNetwork
  store       datastore.DataStore
  sync.Mutex
}
```

Some of the actions performed in a bridge driver:

- Configuring IPTables
- Managing IP forwarding
- Managing Port Mapping
- Enabling Bridge Net Filtering
- Setting up IPv4 and IPv6 on the bridge

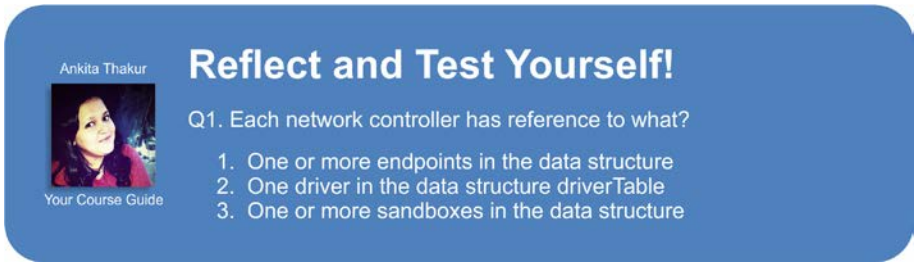
The following diagram shows how the network is represented using `docker0` and `veth` pairs to connect endpoints with the `docker0` bridge:



Overlay network driver

Overlay network in libnetwork uses VXLan along with a Linux bridge to create an overlaid address space. It supports multi-host networking:

```
const (
  networkType  = "overlay"
  vethPrefix   = "veth"
  vethLen      = 7
  vxlanIDStart = 256
  vxlanIDEnd   = 1000
  vxlanPort    = 4789
  vxlanVethMTU = 1450
)
type driver struct {
  eventCh      chan serf.Event
  notifyCh     chan ovNotify
  exitCh       chan chan struct{}
  bindAddress  string
  neighIP      string
  config       map[string]interface{}
  peerDb       peerNetworkMap
  serfInstance *serf.Serf
  networks     networkTable
  store        datastore.DataStore
  ipAllocator  *idm.Idm
  vxlanIdm     *idm.Idm
  once         sync.Once
  joinOnce     sync.Once
  sync.Mutex
}
```



Using overlay network with Vagrant

Overlay network is created between two containers, and VXLan tunnel connects the containers through a bridge.

Overlay network deployment Vagrant setup

This setup has been deployed using the Docker experimental version, which keeps on updating regularly and might not support some of the features:

1. Clone the official libnetwork repository and switch to the `docs` folder:

```
$ git clone
$ cd libnetwork/docs
```

2. The Vagrant script pre-exists in the repository; we will deploy the three-node setup for our Docker overlay network driver testing by using the following command:

```
$ vagrant up
Bringing machine 'consul-server' up with 'virtualbox'
provider...
Bringing machine 'net-1' up with 'virtualbox' provider...
Bringing machine 'net-2' up with 'virtualbox' provider...
==> consul-server: Box 'ubuntu/trusty64' could not be found.
Attempting to find and install...
    consul-server: Box Provider: virtualbox
    consul-server: Box Version: >= 0
==> consul-server: Loading metadata for box 'ubuntu/trusty64'
    consul-server: URL: https://atlas.hashicorp.com/ubuntu/
trusty64
==> consul-server: Adding box 'ubuntu/trusty64'
(v20151217.0.0) for
provider: virtualbox
    consul-server: Downloading:
https://atlas.hashicorp.com/ubuntu/boxes/trusty64/versions/201
51217.0.0/providers/virtualbox.box
==> consul-server: Successfully added box 'ubuntu/trusty64'
(v20151217.0.0) for 'virtualbox'!
==> consul-server: Importing base box 'ubuntu/trusty64'...
==> consul-server: Matching MAC address for NAT networking...
```

```
==> consul-server: Checking if box 'ubuntu/trusty64' is up to
date...
==> consul-server: Setting the name of the VM:
libnetwork_consul-server_1451244524836_56275
==> consul-server: Clearing any previously set forwarded
ports...
==> consul-server: Clearing any previously set network
interfaces...
==> consul-server: Preparing network interfaces based on
configuration...
    consul-server: Adapter 1: nat
    consul-server: Adapter 2: hostonly
==> consul-server: Forwarding ports...
    consul-server: 22 => 2222 (adapter 1)
==> consul-server: Running 'pre-boot' VM customizations...
==> consul-server: Booting VM...
==> consul-server: Waiting for machine to boot. This may take
a few minutes...
consul-server:
101aac79c475b84f6aff48352ead467d6b2b63ba6b64cc1b93c630489f7e3f
4c
==> net-1: Box 'ubuntu/vivid64' could not be found. Attempting
to find and install...
    net-1: Box Provider: virtualbox
    net-1: Box Version: >= 0
==> net-1: Loading metadata for box 'ubuntu/vivid64'
    net-1: URL: https://atlas.hashicorp.com/ubuntu/vivid64
\==> net-1: Adding box 'ubuntu/vivid64' (v20151219.0.0) for
provider: virtualbox
    net-1: Downloading:
https://atlas.hashicorp.com/ubuntu/boxes/vivid64/versions/2015
1219.0.0/providers/virtualbox.box
contd...
```

3. We can list the deployed machine by Vagrant as follows:

```
$ vagrant status
Current machine states:
consul-server           running (virtualbox)
net-1                   running (virtualbox)
```

```
net-2                     running (virtualbox)

This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information
about a specific VM, run `vagrant status NAME`.
```

4. The setup is complete thanks to the Vagrant script; now, we can SSH to the Docker hosts and start the testing containers:

```
$ vagrant ssh net-1

Welcome to Ubuntu 15.04 (GNU/Linux 3.19.0-42-generic x86_64)

* Documentation:https://help.ubuntu.com/

System information as of Sun Dec 27 20:04:06 UTC 2015

System load:  0.0               Users logged in:       0

Usage of /:   4.5% of 38.80GB   IP address for eth0:
10.0.2.15

Memory usage: 24%               IP address for eth1:
192.168.33.11

Swap usage:   0%                IP address for docker0:
172.17.0.1

Processes:    78

Graph this data and manage this system at:
https://landscape.canonical.com/

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

5. We can create a new Docker container, and inside the container we can list the contents of the `/etc/hosts` file in order to verify that it has the overlay bridge specification, which was previously deployed, and it automatically connects to it on the launch:

```
$ docker run -it --rm ubuntu:14.04 bash

Unable to find image 'ubuntu:14.04' locally

14.04: Pulling from library/ubuntu

6edcc89ed412: Pull complete

bdf37643ee24: Pull complete

ea0211d47051: Pull complete

a3ed95caeb02: Pull complete

Digest: sha256:d3b59c1d15c3cfb58d9f2eaab8a232f21fc670c67c11f582bc4
8fb32df17f3b3

Status: Downloaded newer image for ubuntu:14.04


root@65db9144c65b:/# cat /etc/hosts
```

```
172.21.0.4  2ac726b4ce60
127.0.0.1   localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.21.0.3  distracted_bohr
172.21.0.3  distracted_bohr.multihost
172.21.0.4  modest_curie
172.21.0.4  modest_curie.multihost
```

6. Similarly, we can create the Docker container in the other host `net-2` as well and can verify the working of the overlay network driver as both the containers will be able to ping each other in spite of being deployed on different hosts.

In the previous example, we started the Docker container with the default options and they got automatically added to a multi-host network of type overlay.

We can also creat a separate overlay bridge and add containers to it manually using the `--publish-service` option, which is part of Docker experimental:

```
vagrant@net-1:~$ docker network create -d overlay tester
447e75fd19b236e72361c270b0af4402c80e1f170938fb22183758c444966427
vagrant@net-1:~$ docker network ls
NETWORK ID          NAME                DRIVE
447e75fd19b2        tester              overlay
b77a7d741b45        bridge              bridge
40fe7cfeee20        none                null
62072090b6ac        host                host
```

The second host will also see this network and we can create containers added to the overlay network in both of these hosts by using the following option in the Docker command:

```
$ docker run -it --rm --publish-service=bar.tester.overlay
ubuntu:14.04 bash
```

We will be able to verify the working of the overlay driver as both the containers will be able to ping each other. Also, tools such as tcpdump, wireshark, smartsniff, and so on can be used to capture the vXLAN package.

Overlay network with Docker Machine and Docker Swarm

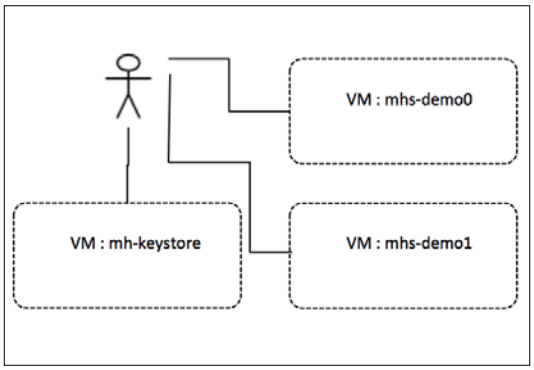
This section explains the basics of creating a multi-host network. The Docker Engine supports multi-host networking through the overlay network driver. Overlay drivers need the following pre-requisites to work:

- 3.16 Linux kernel or higher
- Access to a key-value store
- Docker supports the following key-value stores: Consul, etcd, and ZooKeeper
- A cluster of hosts connected to the key-value store
- Docker Engine daemon on each host in the cluster

This example uses Docker Machine and Docker Swarm to create the multi-network host.

Docker Machine is used to create the key-value store server and the cluster. The cluster created is a Docker Swarm cluster.

The following diagram explains how three VMs are set up using Docker Machine:



Prerequisites

- Vagrant
- Docker Engine
- Docker Machine
- Docker Swarm

Key-value store installation

An overlay network requires a key-value store. The key-value store stores information about the network state such as discovery, networks, endpoints, IP addresses, and so on. Docker supports various key-value stores such as Consul, etcd, and Zoo Keeper. This section has been implemented using Consul.

The following are the steps to install key-value store:

1. Provision a VirtualBox virtual machine called `mh-keystore`.

   When a new VM is provisioned, the process adds the Docker Engine to the host. Consul instance will be using the consul image from the Docker Hub account (`https://hub.docker.com/r/progrium/consul/`):

```
$ docker-machine create -d virtualbox mh-keystore
Running pre-create checks...
Creating machine...
(mh-keystore) Creating VirtualBox VM...
(mh-keystore) Creating SSH key...
(mh-keystore) Starting VM...
Waiting for machine to be running, this may take a few minutes...
Machine is running, waiting for SSH to be available...
Detecting operating system of created instance...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect Docker to this machine, run: docker-
machine env mh-keystore
```

2. Start the `progrium/consul` container created previously running on the `mh-keystore` virtual machine:

```
$ docker $(docker-machine config mh-keystore) run -d \
>     -p "8500:8500" \
>     -h "consul" \
```

```
>     progrium/consul -server -bootstrap

Unable to find image 'progrium/consul:latest' locally
latest: Pulling from progrium/consul
3b4d28ce80e4: Pull complete
…
d9125e9e799b: Pull complete
Digest:
sha256:8cc8023462905929df9a79ff67ee435a36848ce7a10f18d6d0faba9
306b97274
Status: Downloaded newer image for progrium/consul:latest
032884c7834ce22707ed08068c24c503d599499f1a0a58098c31be9cc84d8e
6c
```

A bash expansion `$(docker-machine config mh-keystore)` is used to pass the connection configuration to the Docker `run` command. The client starts a program from the `progrium/consul` image running in the `mh-keystore` machine. The container is called `consul` (flag `-h`) and is listening on port `8500` (you can choose any other port as well).

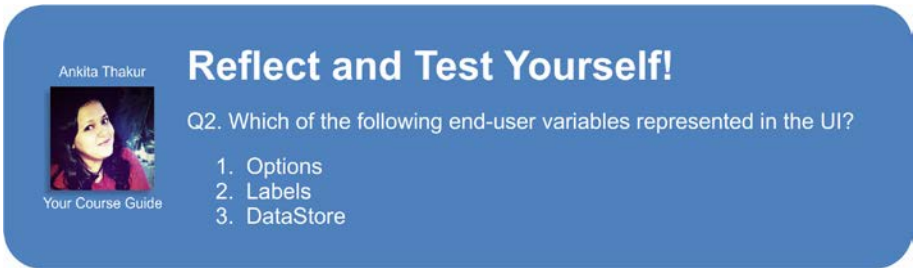
3. Set the local environment to the `mh-keystore` virtual machine:

```
$ eval "$(docker-machine env mh-keystore)"
```

4. Execute the `docker ps` command to make sure the Consul container is up:

```
$ docker ps
CONTAINER ID      IMAGE             COMMAND
CREATED
032884c7834c   progrium/consul   "/bin/start -server -"   47
seconds ago
   STATUS           PORTS
Up 46 seconds  53/tcp, 53/udp, 8300-8302/tcp, 8301-8302/udp,
8400/tcp, 0.0.0.0:8500->8500/tcp
NAMES
sleepy_austin
```

Output has been formatted to fit in the page.



Create a Swarm cluster with two nodes

In this step, we will use Docker Machine to provision two hosts for your network. We will create two virtual machines in VirtualBox. One of the machines will be Swarm master, which will be created first.

As each host is created, options for the overlay network driver will be passed to the Docker Engine using Swarm using the following steps:

1. Create a Swarm master virtual machine `mhs-demo0`:

```
$ docker-machine create \
-d virtualbox \
--swarm --swarm-master \
--swarm-discovery="consul://$(docker-machine ip mh-
keystore):8500" \
--engine-opt="cluster-store=consul://$(docker-machine ip mh-
keystore):8500" \
--engine-opt="cluster-advertise=eth1:2376" \
mhs-demo0
```

   At creation time, you supply the engine daemon with the `--cluster-store` option. This option tells the engine the location of the key-value store for the overlay network. The bash expansion `$(docker-machine ip mh-keystore)` resolves to the IP address of the Consul server you created in step 1 of the preceding section. The `--cluster-advertise` option advertises the machine on the network.

2. Create another virtual machine `mhs-demo1` and add it to the Docker Swarm cluster:

```
$ docker-machine create -d virtualbox \
    --swarm \
    --swarm-discovery="consul://$(docker-machine ip mh-
keystore):8500" \
```

```
    --engine-opt="cluster-store=consul://$(docker-machine ip
mh-keystore):8500" \
    --engine-opt="cluster-advertise=eth1:2376" \
mhs-demo1

Running pre-create checks...
Creating machine...
(mhs-demo1) Creating VirtualBox VM...
(mhs-demo1) Creating SSH key...
(mhs-demo1) Starting VM...
Waiting for machine to be running, this may take a few
minutes...
Machine is running, waiting for SSH to be available...
Detecting operating system of created instance...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Configuring swarm...
Checking connection to Docker...
Docker is up and running!
To see how to connect Docker to this machine, run: docker-
machine env mhs-demo1
```

3. List virtual machines using Docker Machine to confirm that they are all up and running:

```
$ docker-machine ls

NAME          ACTIVE   DRIVER       STATE     URL
SWARM                 DOCKER   ERRORS
mh-keystore   *        virtualbox   Running
tcp://192.168.99.100:2376                         v1.9.1
mhs-demo0     -        virtualbox   Running
tcp://192.168.99.101:2376   mhs-demo0 (master)    v1.9.1
mhs-demo1     -        virtualbox   Running
tcp://192.168.99.102:2376   mhs-demo0             v1.9.1
```

At this point, virtual machines are running. We are ready to create a multi-host network for containers using these virtual machines.

Creating an overlay network

The following command is used to create an overlay network:

```
$ docker network create --driver overlay my-net
```

We will only need to create the network on a single host in the Swarm cluster. We used the Swarm master but this command can run on any host in the Swarm cluster:

1. Check that the overlay network is running using the following command:

```
$ docker network ls

bd85c87911491d7112739e6cf08d732eb2a2841c6ca1efcc04d0b20bbb832a33
rdua1-ltm:overlay-tutorial rdua$ docker network ls
NETWORK ID          NAME                DRIVER
bd85c8791149        my-net              overlay
fff23086faa8        mhs-demo0/bridge    bridge
03dd288a8adb        mhs-demo0/none      null
2a706780454f        mhs-demo0/host      host
f6152664c40a        mhs-demo1/bridge    bridge
ac546be9c37c        mhs-demo1/none      null
c6a2de6ba6c9       mhs-demo1/host     host
```

   Since we are using the Swarm master environment, we are able to see all the networks on all the Swarm agents: the default networks on each engine and the single overlay network. In this case, there are two engines running on `mhs-demo0` and `mhs-demo1`.

   Each `NETWORK ID` is unique.

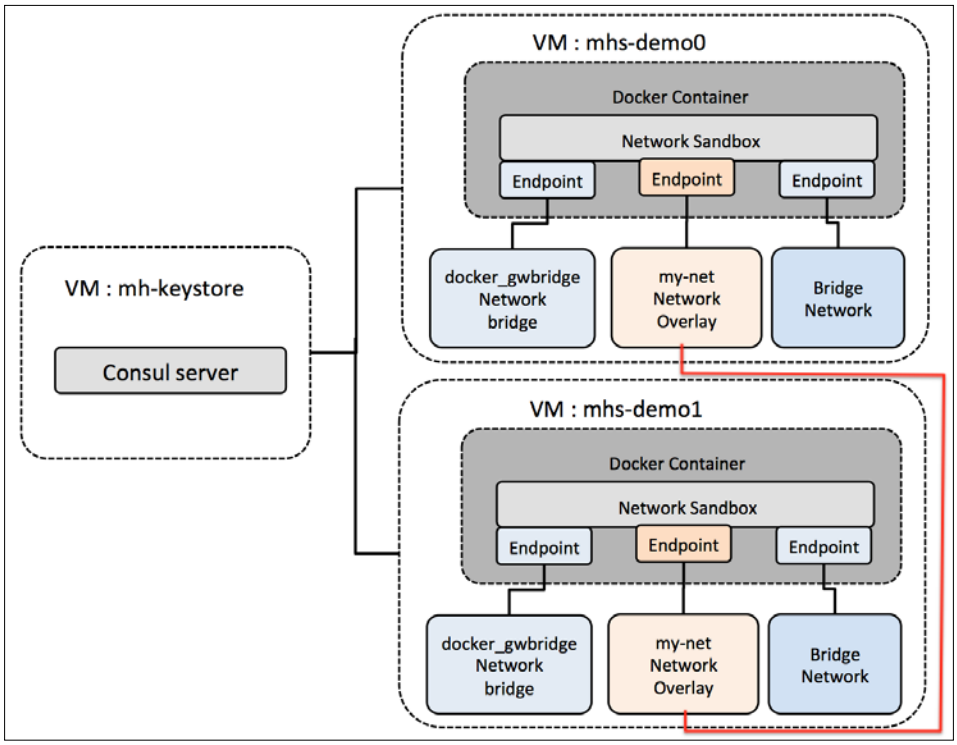
2. Switch to each Swarm agent in turn and list the networks:

```
$ eval $(docker-machine env mhs-demo0)

$ docker network ls
NETWORK ID          NAME                DRIVER
bd85c8791149        my-net              overlay
03dd288a8adb        none                  null
2a706780454f        host                  host
fff23086faa8        bridge              bridge

$ eval $(docker-machine env mhs-demo1)
```

```
$ docker network ls

NETWORK ID          NAME                DRIVER
bd85c8791149        my-net              overlay
358c45b96beb        docker_gwbridge     bridge
f6152664c40a        bridge              bridge
ac546be9c37c        none                null
c6a2de6ba6c9        host                host
```

Both agents report they have the `my-net` network with the overlay driver. We have a multi-host overlay network running.

The following figure shows how two containers will have containers created and tied together using the overlay `my-net`:



Creating containers using an overlay network

The following are the steps for creating containers using an overlay network:

1. Create a container `c0` on `mhs-demo0` and connect to the `my-net` network:

```
$ eval $(docker-machine env mhs-demo0)
root@843b16be1ae1:/#

$ sudo docker run -i -t --name=c0 --net=my-net  debian
/bin/bash
```

   Execute `ifconfig` to find the IPaddress of `c0`. In this case, it is `10.0.0.4`:

```
root@843b16be1ae1:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:0a:00:00:04
          inet addr:10.0.0.4  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:aff:fe00:4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:17 errors:0 dropped:0 overruns:0 frame:0
          TX packets:17 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1474 (1.4 KB)  TX bytes:1474 (1.4 KB)

eth1      Link encap:Ethernet  HWaddr 02:42:ac:12:00:03
          inet addr:172.18.0.3  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe12:3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
```

```
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

2. Create a container, `c1` on `mhs-demo1`, and connect to the `my-net` network:

```
$ eval $(docker-machine env mhs-demo1)

$ sudo docker run -i -t --name=c1 --net=my-net  debian /bin/bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
0bf056161913: Pull complete
1796d1c62d0c: Pull complete
e24428725dd6: Pull complete
89d5d8e8bafb: Pull complete
Digest:
sha256:a2b67b6107aa640044c25a03b9e06e2a2d48c95be6ac17fb1a387e7
5eebafd7c
Status: Downloaded newer image for ubuntu:latest
        root@2ce83e872408:/#
```

3. Execute `ifconfig` to find the IP address of `c1`. In this case, it is `10.0.0.3`:

```
root@2ce83e872408:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:0a:00:00:03
          inet addr:10.0.0.3  Bcast:0.0.0.0
Mask:255.255.255.0
          inet6 addr: fe80::42:aff:fe00:3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:13 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
```

```
          collisions:0 txqueuelen:0
          RX bytes:1066 (1.0 KB)  TX bytes:578 (578.0 B)

eth1      Link encap:Ethernet  HWaddr 02:42:ac:12:00:02
          inet addr:172.18.0.2  Bcast:0.0.0.0
Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe12:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:578 (578.0 B)  TX bytes:578 (578.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

4. Ping `c1` (`10.0.0.3`) from `c0` (`10.0.0.4`) and vice versa:

```
root@2ce83e872408:/# ping 10.0.04
PING 10.0.04 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=0.370 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=0.443 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.441 ms
```

Container network interface

Container network interface (**CNI**) is a specification that defines how executable plugins can be used to configure network interfaces for Linux application containers. The official GitHub repository of CNI explains how a go library explains the implementing specification.

The container runtime first creates a new network namespace for the container in which it determines which network this container should belong to and which plugins to be executed. The network configuration is in the JSON format and defines on the container startup which plugin should be executed for the network. CNI is actually an evolving open source technology that is derived from the rkt networking protocol. Each CNI plugin is implemented as an executable and is invoked by a container management system, docker, or rkt.

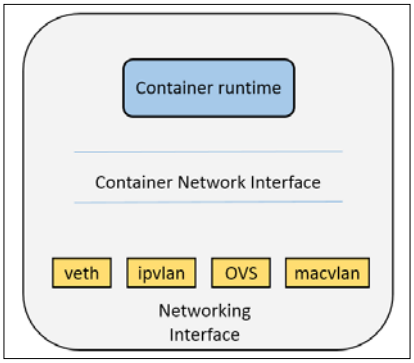
After inserting the container in the network namespace, namely by attaching one end of a veth pair to a container and attaching the other end to a bridge, it then assigns an IP to the interface and sets up routes consistent with IP address management by invoking an appropriate IPAM plugin.

The CNI model is currently used for the networking of kubelets in the Kubernetes model. Kubelets are the most important components of Kubernetes nodes, which takes the load of running containers on top of them.

The package CNI for kubelet is defined in the following Kubernetes package:

```
Constants
const (
    CNIPluginName        = "cni"
    DefaultNetDir        = "/etc/cni/net.d"
    DefaultCNIDir        = "/opt/cni/bin"
    DefaultInterfaceName = "eth0"
    VendorCNIDirTemplate = "%s/opt/%s/bin"
)
func ProbeNetworkPlugins
func ProbeNetworkPlugins(pluginDir string) []network.NetworkPlugin
```

The following figure shows the CNI placement:



CNI plugin

As per the official GitHub repository (`https://github.com/appc/cni`), the parameters that the CNI plugin need in order to add a container to the network are:

- **Version**: The version of CNI spec that the caller is using (container call invoking the plugin).
- **Container ID**: This is optional, but recommended, and defines that there should be a unique ID across an administrative domain while the container is live. For example, the IPAM system may require that each container is allocated a unique ID so that it can be correlated properly to a container running in the background.
- **Network namespace path**: This represents the path to the network namespace to be added, for example, `/proc/[pid]/ns/net` or a `bind-mount/link` to it.
- **Network configuration**: It is the JSON document that describes a network to which a container can be joined and is explained in the following section.
- **Extra arguments**: It allows granular configuration of CNI plugins on a per-container basis.
- **Name of the interface inside the container**: It is the name that gets assigned to the container and complies with Linux restriction, which exists for interface names.

The results achieved are as follows:

- **IPs assigned to the interface**: This is either an IPv4 address or an IPv6 address assigned to the network as per requirements.
- **List of DNS nameservers**: This is a priority-ordered address list of DNS name servers.

Network configuration

The network configuration is in the JSON format that can be stored on disk or generated from other sources by container runtime. The following fields in the JSON have importance, as explained in the following:

- **cniVersion (string)**: It is Semantic Version 2.0 of the CNI specification to which this configuration meets.
- **name (string)**: It is the network name. It is unique across all containers on the host (or other administrative domain).
- **type (string)**: Refers to the filename of the CNI plugin executable.

- **ipMasq (boolean)**: Optional, sets up an IP masquerade on the host as it is necessary for the host to act as a gateway to subnets that are not able to route to the IP assigned to the container.
- **ipam**: Dictionary with IPAM-specific values.
- **type (string)**: Refers to the filename of the IPAM plugin executable.
- **routes (list)**: List of subnets (in CIDR notation) that the CNI plugin should make sure are reachable by routing through the network. Each entry is a dictionary containing:
    - **dst (string)**: A subnet in CIDR notation
    - **gw (string)**: It is the IP address of the gateway to use. If not specified, the default gateway for the subnet is assumed (as determined by the IPAM plugin).

An example configuration for plugin-specific OVS is as follows:

```
{
  "cniVersion": "0.1.0",
  "name": "pci",
  "type": "ovs",
  // type (plugin) specific
  "bridge": "ovs0",
  "vxlanID": 42,
  "ipam": {
    "type": "dhcp",
    "routes": [ { "dst": "10.3.0.0/16" }, { "dst": "10.4.0.0/16" } ]
  }
}
```

IP allocation

The CNI plugin assigns an IP address to the interface and installs necessary routes for the interface, thus it provides great flexibility for the CNI plugin and many CNI plugins internally have the same code to support several IP management schemes.

To lessen the burden on the CNI plugin, a second type of plugin, **IP address management plugin** (**IPAM**), is defined, which determines the interface IP/subnet, gateway, and routes and returns this information to the main plugin to apply. The IPAM plugin obtains information via a protocol, `ipam` section defined in the network configuration file, or data stored on the local filesystem.

IP address management interface

The IPAM plugin is invoked by running an executable, which is searched in a predefined path and is indicated by a CNI plugin via `CNI_PATH`. The IPAM plugin receives all the system environment variables from this executable, which are passed to the CNI plugin.

IPAM receives a network configuration file via stdin. Success is indicated by a zero return code and the following JSON, which gets printed to stdout (in the case of the `ADD` command):

```
{
  "cniVersion": "0.1.0",
  "ip4": {
    "ip": <ipv4-and-subnet-in-CIDR>,
    "gateway": <ipv4-of-the-gateway>,  (optional)
    "routes": <list-of-ipv4-routes>    (optional)
  },
  "ip6": {
    "ip": <ipv6-and-subnet-in-CIDR>,
    "gateway": <ipv6-of-the-gateway>,  (optional)
    "routes": <list-of-ipv6-routes>    (optional)
  },
  "dns": <list-of-DNS-nameservers>     (optional)
}
```

The following is an example of running Docker networking with CNI:

1. First, install Go Lang 1.4+ and jq (command line JSON processor) to build the CNI plugins:

```
$ wget https://storage.googleapis.com/golang/go1.5.2.linux-amd64.
tar.gz
$ tar -C /usr/local -xzf go1.5.2.linux-amd64.tar.gz
$ export PATH=$PATH:/usr/local/go/bin
$ go version
go version go1.5.2 linux/amd64
$ sudo apt-get install jq
```

2. Clone the official CNI GitHub repository:

```
$ git clone https://github.com/appc/cni.git
Cloning into 'cni'...
remote: Counting objects: 881, done.
```

```
remote: Total 881 (delta 0), reused 0 (delta 0), pack-reused
881
Receiving objects: 100% (881/881), 543.54 KiB | 313.00 KiB/s,
done.
Resolving deltas: 100% (373/373), done.
Checking connectivity... done.
```

3. We will now create a `netconf` file in order to describe the network:

```
mkdir -p /etc/cni/net.d
root@rajdeepd-virtual-machine:~# cat >/etc/cni/net.d/10-
mynet.conf <<EOF
>{
>  "name": "mynet",
>  "type": "bridge",
>  "bridge": "cni0",
>  "isGateway": true,
>  "ipMasq": true,
>  "ipam": {
>    "type": "host-local",
>    "subnet": "10.22.0.0/16",
>    "routes": [
>      { "dst": "0.0.0.0/0" }
>    ]
>  }
>}
> EOF
```

4. Build the CNI plugins:

```
~/cni$ ./build
Building API
Building reference CLI
Building plugins
  flannel
  bridge
  ipvlan
  macvlan
  ptp
  dhcp
  host-local
```

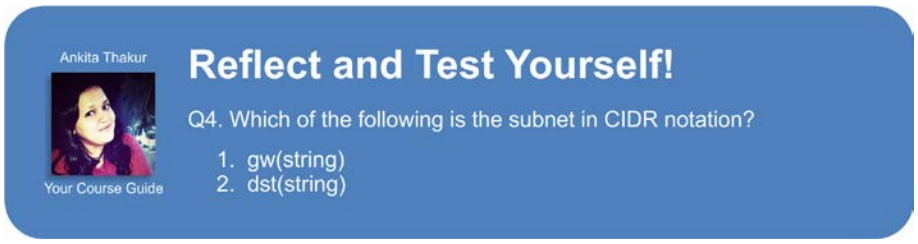
5. Now we will execute the `priv-net-run.sh` script in order to create the private network with the CNI plugin:

```
~/cni/scripts$ sudo CNI_PATH=$CNI_PATH ./priv-net-run.sh
ifconfig
eth0      Link encap:Ethernet  HWaddr 8a:72:75:7d:6d:6c
          inet addr:10.22.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::8872:75ff:fe7d:6d6c/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:90 (90.0 B)  TX bytes:90 (90.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

6. Run a Docker container with the network namespace, which was set up previously using the CNI plugin:

```
~/cni/scripts$ sudo CNI_PATH=$CNI_PATH ./docker-run.sh --rm
busybox:latest /bin/ifconfig
eth0      Link encap:Ethernet  HWaddr 92:B2:D3:E5:BA:9B
          inet addr:10.22.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::90b2:d3ff:fee5:ba9b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:180 (180.0 B)  TX bytes:168 (168.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
```

```
UP LOOPBACK RUNNING  MTU:65536  Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```



Project Calico's libnetwork driver

Calico provides a scalable networking solution for connecting containers, VMs, or bare metal. Calico provides connectivity using the scalable IP networking principle as a layer 3 approach. Calico can be deployed without overlays or encapsulation. The Calico service should be deployed as a container on each node and provides each container with its own IP address. It also handles all the necessary IP routing, security policy rules, and distribution of routes across a cluster of nodes.

The Calico architecture contains four important components in order to provide a better networking solution:

- Felix, the Calico worker process, is the heart of Calico networking, which primarily routes and provides desired connectivity to and from the workloads on host. It also provides the interface to kernels for outgoing endpoint traffic.
- BIRD, the route distribution open source BGP, exchanges routing information between hosts. The kernel endpoints, which are picked up by BIRD, are distributed to BGP peers in order to provide inter-host routing. Two BIRD processes run in the calico-node container, IPv4 (bird) and one for IPv6 (bird6).
- Confd, a templating process to auto-generate configuration for BIRD, monitors the etcd store for any changes to BGP configuration such as log levels and IPAM information. Confd also dynamically generates BIRD configuration files based on data from etcd and triggers automatically as updates are applied to data. Confd triggers BIRD to load new files whenever a configuration file is changed.

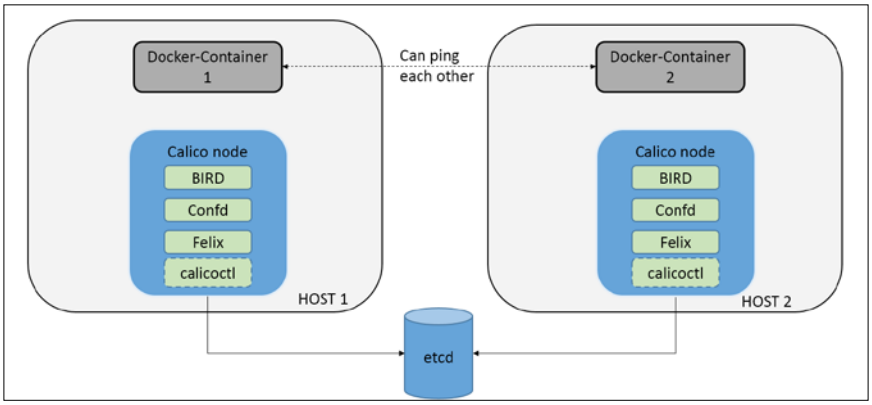
- calicoctl, the command line used to configure and start the Calico service, even allows the datastore (etcd) to define and apply security policy. The tool also provides the simple interface for general management of Calico configuration irrespective of whether Calico is running on VMs, containers, or bare metal. The following commands are supported at calicoctl:

```
$ calicoctl
Override the host:port of the ETCD server by setting the
environment variable ETCD_AUTHORITY [default: 127.0.0.1:2379]
Usage: calicoctl <command> [<args>...]

status            Print current status information

node              Configure the main calico/node container and
establish Calico networking

container         Configure containers and their addresses

profile           Configure endpoint profiles

endpoint          Configure the endpoints assigned to existing
containers

pool              Configure ip-pools

bgp               Configure global bgp

ipam              Configure IP address management

checksystem       Check for incompatibilities on the host
system

diags             Save diagnostic information

version           Display the version of calicoctl

config            Configure low-level component configuration

See 'calicoctl <command> --help' to read about a specific
subcommand.
```

As per the official GitHub page of the Calico repository (`https://github.com/projectcalico/calico-containers`), the following integration of Calico exists:

- Calico as a Docker network plugin
- Calico without Docker networking
- Calico with Kubernetes
- Calico with Mesos
- Calico with Docker Swarm

The following figure shows the Calico architecture:



In the following tutorial we will run the manual set up of Calico on a single node machine with Docker 1.9, which finally brings libnetwork out of its experimental version to main release, and Calico can be configured directly without the need of other Docker experimental versions:

1. Get the etcd latest release and configure it on the default port 2379:

```
$ curl -L
https://github.com/coreos/etcd/releases/download/v2.2.1/etcd-
v2.2.1-linux-amd64.tar.gz -o etcd-v2.2.1-linux-amd64.tar.gz
  % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                 Dload  Upload   Total   Spent
Left  Speed
100   606    0   606    0     0    445      0 --:--:--
0:00:01 --:--:--   446
100 7181k  100 7181k    0     0   441k      0  0:00:16
0:00:16 --:--:-- 1387k
$ tar xzvf etcd-v2.2.1-linux-amd64.tar.gz
etcd-v2.2.1-linux-amd64/
etcd-v2.2.1-linux-amd64/Documentation/
etcd-v2.2.1-linux-amd64/Documentation/04_to_2_snapshot_migration.
md
etcd-v2.2.1-linux-amd64/Documentation/admin_guide.md
etcd-v2.2.1-linux-amd64/Documentation/api.md
contd..
etcd-v2.2.1-linux-amd64/etcd
etcd-v2.2.1-linux-amd64/etcdctl
```

```
etcd-v2.2.1-linux-amd64/README-etcdctl.md
etcd-v2.2.1-linux-amd64/README.md


$ cd etcd-v2.2.1-linux-amd64
$ ./etcd
2016-01-06 15:50:00.065733 I | etcdmain: etcd Version: 2.2.1
2016-01-06 15:50:00.065914 I | etcdmain: Git SHA: 75f8282
2016-01-06 15:50:00.065961 I | etcdmain: Go Version: go1.5.1
2016-01-06 15:50:00.066001 I | etcdmain: Go OS/Arch: linux/amd64
Contd..
2016-01-06 15:50:00.107972 I | etcdserver: starting server...
[version: 2.2.1, cluster version: 2.2]
2016-01-06 15:50:00.508131 I | raft: ce2a822cea30bfca is
starting a new election at term 5
2016-01-06 15:50:00.508237 I | raft: ce2a822cea30bfca became
candidate at term 6
2016-01-06 15:50:00.508253 I | raft: ce2a822cea30bfca received
vote from ce2a822cea30bfca at term 6
2016-01-06 15:50:00.508278 I | raft: ce2a822cea30bfca became
leader at term 6
2016-01-06 15:50:00.508313 I | raft: raft.node:
ce2a822cea30bfca elected leader ce2a822cea30bfca at term 6
2016-01-06 15:50:00.509810 I | etcdserver: published
{Name:default ClientURLs:[http://localhost:2379
http://localhost:4001]} to cluster 7e27652122e8b2ae
```

2. Open the new terminal and configure the Docker daemon with the etcd key-value store by running the following commands:

```
$ service docker stop
$ docker daemon --cluster-store=etcd://0.0.0.0:2379
INFO[0000] [graphdriver] using prior storage driver "aufs"
INFO[0000] API listen on /var/run/docker.sock
INFO[0000] Firewalld running: false
INFO[0015] Default bridge (docker0) is assigned with an IP
address 172.16.59.1/24. Daemon option --bip can be used to set
a preferred IP address
WARN[0015] Your kernel does not support swap memory limit.
INFO[0015] Loading containers: start.
.....INFO[0034] Skipping update of resolv.conf file with
ipv6Enabled: false because file was touched by user
```

```
INFO[0043] Loading containers: done.

INFO[0043] Daemon has completed initialization

INFO[0043] Docker daemon       commit=a34a1d5
execdriver=native-0.2 graphdriver=aufs version=1.9.1

INFO[0043] GET /v1.21/version

INFO[0043] GET /v1.21/version

INFO[0043] GET /events

INFO[0043] GET /v1.21/version
```

3. Now, in the new terminal, start the Calico container in the following way:

```
$ ./calicoctl node --libnetwork

No IP provided. Using detected IP: 10.22.0.1

Pulling Docker image calico/node:v0.10.0

Calico node is running with id:
79e75fa6d875777d31b8aead10c2712f54485c031df50667edb4d7d7cb6bb2
6c

Pulling Docker image calico/node-libnetwork:v0.5.2

Calico libnetwork driver is running with id:
bc7d65f6ab854b20b9b855abab4776056879f6edbcde9d744f218e556439997f

$ docker ps

CONTAINER ID        IMAGE                              COMMAND
       CREATED              STATUS              PORTS
NAMES

7bb7a956af37        calico/node-libnetwork:v0.5.2
"./start.sh"           3 minutes ago       Up 3 minutes
               calico-libnetwork

13a0314754d6        calico/node:v0.10.0                "/sbin/start_
runit"    3 minutes ago       Up 3 minutes
               calico-node

1f13020cc3a0        weaveworks/plugin:1.4.1            "/home/weave/
plugin"   3 days ago          Up 3 minutes
               weaveplugin
```

4. Create the Calico bridge using the `docker network` command recently introduced in the Docker CLI:

```
$docker network create -d calico net1

$ docker network ls

NETWORK ID          NAME                DRIVER

9b5f06307cf2        docker_gwbridge     bridge

1638f754fbaf        host                host
```

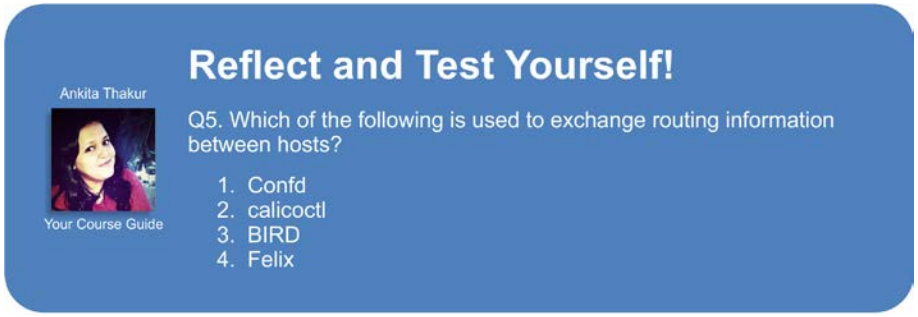
```
02b10aaa25d7        weave               weavemesh
65dc3cbcd2c0        bridge              bridge
f034d78cc423        net1                calico
```

5. Start the `busybox` container connected to the Calico `net1` bridge:

```
$docker run --net=net1 -itd --name=container1 busybox
1731629b6897145822f73726194b1f7441b6086ee568e973d8a88b554e838366
$ docker ps
CONTAINER ID        IMAGE                              COMMAND
CREATED             STATUS              PORTS                NAMES
1731629b6897        busybox                            "sh"
6 seconds ago       Up 5 seconds
container1
7bb7a956af37        calico/node-libnetwork:v0.5.2      "./
start.sh"           6 minutes ago       Up 6 minutes
calico-libnetwork
13a0314754d6        calico/node:v0.10.0                "/
sbin/start_runit"    6 minutes ago       Up 6 minutes
calico-node
1f13020cc3a0        weaveworks/plugin:1.4.1            "/
home/weave/plugin"   3 days ago          Up 6 minutes
weaveplugin
$ docker attach 1731
/ #
/ # ifconfig
cali0     Link encap:Ethernet  HWaddr EE:EE:EE:EE:EE:EE
          inet addr:10.0.0.2  Bcast:0.0.0.0
Mask:255.255.255.0
          inet6 addr: fe80::ecee:eeff:feee:eeee/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:29 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:5774 (5.6 KiB)  TX bytes:648 (648.0 B)


eth1      Link encap:Ethernet  HWaddr 02:42:AC:11:00:02
          inet addr:172.17.0.2  Bcast:0.0.0.0
Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:2/64 Scope:Link
```

```
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:21 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:4086 (3.9 KiB)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Inside the container we can see that the container is now connected to the Calico bridge and can connect to the other containers deployed on the same bridge.



Ankita Thakur

Your Course Guide

Your Coding Challenge

You've seen Project Calico in detail but 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. Try experimenting with the other plugins—to name a few, Weave, Nuage Networks, Midokura, VMware, Cisco, and Microsoft.

Summary of Module 2 Chapter 6

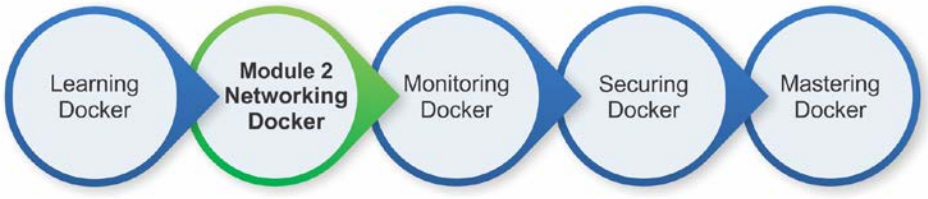
Ankita Thakur

Your Course Guide

In this chapter, we looked into some of the deeper and more conceptual aspects of Docker networking, one of them being libnetworking. While explaining libnetworking, we also studied the CNM model and its various objects and components with its implementation code snippets. Next, we looked into drivers of CNM, the prime one being the overlay driver, in detail, with deployment as part of the Vagrant setup. We also looked at the stand-alone integration of containers with the overlay network and as well with Docker Swarm and Docker Machine. In the next section, we explained about the CNI interface, its executable plugins, and a tutorial of configuring Docker networking with the CNI plugin.

In the last section, project Calico is explained in detail, which provides a scalable networking solution based out of libnetwork and provides integration with Docker, Kubernetes, Mesos, bare-metal, and VMs primarily.

Your Progress through the Course So Far



Course Module 3
Monitoring Docker

Course Module 1: Learning Docker
Chapter 1: Getting Started with Docker
Chapter 2: Up and Running
Chapter 3: Container Image Storage
Chapter 4: Working with Docker Containers and Images
Chapter 5: Publishing Images
Chapter 6: Running Your Private Docker Infrastructure
Chapter 7: Running Services in a Container
Chapter 8: Sharing Data with Containers
Chapter 9: Docker Machine
Chapter 10: Orchestrating Docker
Chapter 11: Docker Swarm
Chapter 12: Testing with Docker
Chapter 13: Debugging Containers

Course Module 2: Networking Docker
Chapter 1: Docker Networking Primer
Chapter 2: Docker Networking Internals
Chapter 3: Building Your First Docker Network
Chapter 4: Networking in a Docker Cluster
Chapter 5: Next Generation Networking Stack for Docker – libnetwork

Course Module 3: Monitoring Docker
Chapter 1: Introduction to Docker Monitoring
Chapter 2: Using the Built-in Tools
Chapter 3: Advanced Container Resource Analysis
Chapter 4: A Traditional Approach to Monitoring Containers
Chapter 5: Querying with Sysdig
Chapter 6: Exploring Third Party Options
Chapter 7: Collecting Application Logs from within the Container
Chapter 8: What Are the Next Steps?

Ankita Thakur

Your Course Guide

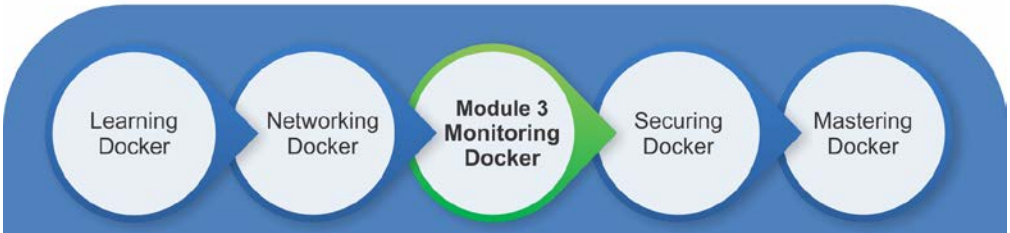
Monitor Docker containers with Course Module 3, Monitoring Docker

Course Module 3
Monitoring Docker

Course Module 4: Securing Docker
Chapter 1: Securing Docker Hosts
Chapter 2: Securing Docker Components
Chapter 3: Securing and Hardening Linux Kernels
Chapter 4: Docker Bench for Security
Chapter 5: Monitoring and Reporting Docker Security Incidents
Chapter 6: Using Docker's Built-in Security Features
Chapter 7: Securing Docker with Third-Party Tools
Chapter 8: Keeping up Security

Course Module 5: Mastering Docker
Chapter 1: Docker in Production
Chapter 2: Shipyard
Chapter 3: Panamax
Chapter 4: Tutum
Chapter 5: Advanced Docker
A Final Run-Through
Reflect and Test Yourself! Answers

Course Module 3



With the increase in the adoption of Docker containers, the need to monitor which containers are running, what resources they are consuming, and how it affects the overall performance of the system, has become a time-related need. This module will cover monitoring containers using Docker's native monitoring functions, various plugins, and also third-party tools that help in monitoring. It will first cover how to obtain detailed stats for the active containers, resources consumed, and container behavior. This module will also show you how to use these stats to improve the overall performance of the system. Next, you will learn how to use Sysdig to both view your containers' performance metrics in real time and record sessions to query later. By the end of this module, you will have a complete knowledge of how to implement monitoring for your containerized applications and make the most of the metrics you are collecting.

So, let's get the most out of your application and resources with the right implementation of your monitoring method.

Ankita Thakur

Your Course Guide

1
Introduction to Docker Monitoring

Docker has been a recent but very important addition to a SysAdmins toolbox.

Docker describes itself as an open platform for building, shipping, and running distributed applications. This means that developers can bundle their code and pass it to their operations team. From here, they can deploy safe in the knowledge that it will be done so in a way that introduces consistency with the environment in which the code is running.

When this process is followed, it should make the age-old developers versus operations argument of "it worked on my local development server"—a thing of the past. Since before its "production ready" 1.0 release back in June 2014, there had been over 10,000 Dockerized applications available. By the end of 2014, that number had risen to over 71,000. You can see how Docker grew in 2014 by looking at the infographic that was published by Docker in early 2015, which can be found at `https://blog.docker.com/2015/01/docker-project-2014-a-whirlwind-year-in-review/`.

While the debate is still raging about how production ready the technology is, Docker has gained an impressive list of technology partners, including RedHat, Canonical, HP, and even Microsoft.

Companies such as Google, Spotify, Soundcloud, and CenturyLink, have all open sourced tools that support Docker in some way, shape, or form and there has also been numerous independent developers who have released apps that provide additional functionality to the core Docker product set. Also, all the companies have sprung up around the Docker ecosystem.

This module assumes that you have had some level of experience building, running, and managing Docker containers, and that you would now like to start to metrics from your running applications to further tune them, or that you would like to know when a problem occurs with a container so that you can debug any ongoing issues.

If you have never used Docker before, you may want to try one of the excellent modules that serve and introduce you to all the things that Docker provides, modules such as *Learning Docker*, *Packt Publishing*, or Docker's own introduction to containers, which can be found at their documentation pages, as follows:

- Learning Docker: `https://www.packtpub.com/virtualization-and-cloud/learning-docker`
- Official Docker docs: `https://docs.docker.com/`

Now, we have a brought ourselves up to speed with what Docker is; the rest of this chapter will cover the following topics:

- How different is it to monitor containers versus more traditional servers such as virtual machines, bare metal machine, and cloud instances (Pets, Cattle, Chickens, and Snowflakes).
- What are the minimum versions of Docker you should be running?
- How to follow instructions on bringing up an environment locally using Vagrant in order to follow the practical exercises in this module

Pets, Cattle, Chickens, and Snowflakes

Before we start discussing the various ways in which you can monitor your containers, we should get an understanding of what a SysAdmins world looks like these days and also where containers fit into it.

A typical SysAdmin will probably be looking after an estate of servers that are hosted in either an on-site or third-party data center, some may even manage instances hosted in a public cloud such as Amazon Web Services or Microsoft Azure, and some SysAdmins may juggle all their server estates across multiple hosting environments.

Each of these different environments has its own way of doing things, as well as performing best practices. Back in February 2012, Randy Bias gave a talk at Cloudscaling that discussed architectures for open and scalable clouds. Towards the end of the slide deck, Randy introduced the concept of Pets versus Cattle (which he attributes to Bill Baker, who was then an engineer at Microsoft).

You can view the original slide deck at `http://www.slideshare.net/randybias/architectures-for-open-and-scalable-clouds`.

Pets versus Cattle is now widely accepted as a good analogy to describe modern hosting practices.

Pets

Pets are akin to traditional physical servers or virtual machines, as follows:

- Each pet has a name; for example, `myserver.domain.com`.
- When they're not well, you take them to the vet to help them get better. You employ SysAdmins to look after them.
- You pay close attention to them, sometimes for years. You take backups, patch them, and ensure that they are fully documented.

Cattle

Cattle, on the other hand, represent more modern cloud computing instances, as follows:

- You've got too many to name, so you give them numbers; for example, the URL could look something like `ip123123123123.eu.public-cloud.com`.
- When they get sick, you shoot them and if your herd requires it, you replace anything you've killed: A server crashes or shows signs that it is having problems, you terminate it and your configuration automatically replaces it with an exact replica.
- You put them in a field and watch them from far and you don't expect them to live long. Rather than monitoring the individual instances, you monitor the cluster. When more resources are needed, you add more instances and once the resource is no longer required, you terminate the instances to get you back to your base configuration.

Chickens

Next up is a term that is a good way of describing how containers fit into the Pets versus Cattle world; in a blog post title "Cloud Computing: Pets, Cattle and ... Chickens?" on ActiveState, Bernard Golden describes containers as Chickens:

- They're more efficient than cattle when it comes to resource use. A container can boot in seconds where a instance or server can take minutes; it also uses less CPU power than a typical virtual machine or cloud instance.

- There are many more chickens than cattle. You can quite densely pack containers onto your instances or servers.
- Chickens tend to have a shorter lifespan than cattle and pets. Containers lend themselves to running micros-services; these containers may only be active for a few minutes.

The original blog post can be found at `http://www.activestate.com/blog/2015/02/cloud-computing-pets-cattle-and-chickens`.

Snowflakes

The final term is not animal-related and it describes a type of server that you defiantly don't want to have in your server estate, a Snowflake. This term was penned by Martin Fowler in a blog post titled "SnowflakeServer". Snowflakes is a term applied to "legacy" or "inherited" servers:

- Snowflakes are delicate and are treated with kid gloves. Typically, the server has been in the data center since you started. No one knows who originally configured it and there is no documentation of it; all you know is that it is important.
- Each one is unique and is impossible to exactly reproduce. Even the most hardened SysAdmin fears to reboot the machine incase it doesn't boot afterwards, as it is running end-of-life software that can not easily be reinstalled.

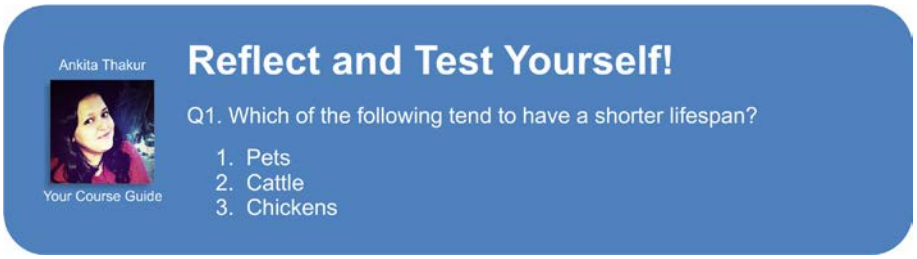
Martin's post can be found at `http://martinfowler.com/bliki/SnowflakeServer.html`.

So what does this all mean?

Depending on your requirements and the application you want to deploy, your containers can be launched onto either pet or cattle style servers. You can also create a clutch of chickens and have your containers run micro-services.

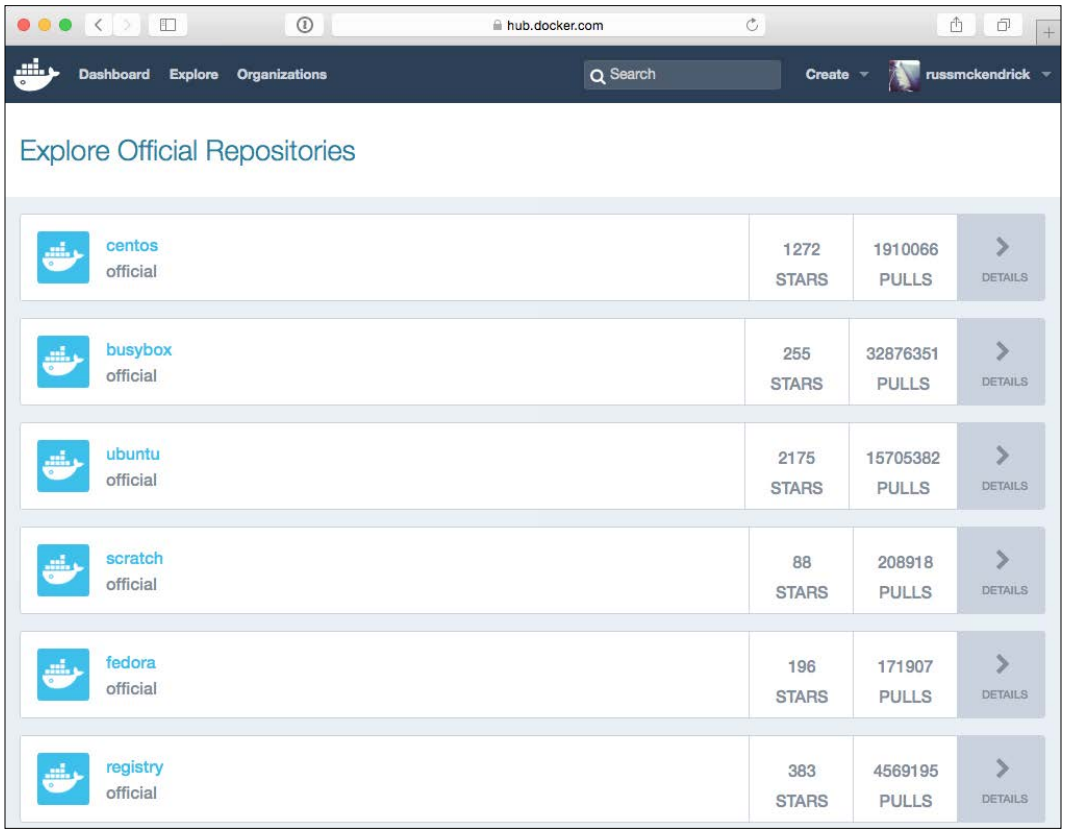
Also, in theory, you can replace your feared snowflake servers with a container-based application that meets all the end-of-life software requirements while remaining deployable on a modern supportable platform.

Each of the different styles of server has different monitoring requirements, in the final chapter we will look at Pets, Cattle, Chickens, and Snowflakes again and discuss the tools we have covered in the coming chapters. We will also cover best practices you should take into consideration when planning your monitoring.



We assume that you have Docker Compose 1.3.3 or later installed; the `docker-compose.yml` files mentioned in this module have been written with this version in mind.

Finally, the majority of the images we will be deploying during this module will be sourced from the Docker Hub (`https://hub.docker.com/`), which not only houses a public registry containing over 40,000 public images but also 100 official images. The following screenshot shows the official repositories listing on the Docker Hub website:



You can also sign up and use the Docker Hub to host your own public and private images.

Launching a local environment

Wherever possible, I will try to ensure that the practical exercises in this module will be able to be run on a local machine such as your desktop or laptop. For the purposes of this module, I will assume that your local machine is running either a recent version OS X or an up-to-date Linux distribution and has a high enough specification to run the software mentioned in this chapter.

The two tools we will be using to launch our Docker instances will also run on Windows; therefore, it should be possible to follow the instructions within this, although you may have to refer the usage guides for any changes to the syntax.

Due to the way in which Docker is architected, a lot of the content of this module will have you running commands and interacting with the command line on the virtual server that is acting as the host machine, rather than the containers themselves. Because of this, we will not be using either Docker Machine or Kitematic.

As we will be installing additional packages on the host machines, to ensure that there are no problems further, we be running a full operating system.

I will be providing practical examples for both operating systems (CentOS and Ubuntu).

To ensure the experience is as consistent as possible, we will be installing Vagrant and VirtualBox to run the virtual machine that will act as a host to run our containers.

Vagrant, written by Mitchell Hashimoto, is a command line tool for creating and configuring reproducible and portable virtual machine environments. There have been numerous blog posts and articles that actually pitch Docker against Vagrant; however, in our case, the two technologies work quite well together in providing a repeatable and consistent environment.

Vagrant is available for Linux, OS X, and Windows. For details on how to install, go to the Vagrant website at `https://www.vagrantup.com/`.

VirtualBox is a great all round open source virtualization platform originally developed by Sun and now maintained by Oracle. It allows you to run both 32-bit and 64-bit guest operating systems on your local machine. Details on how to download and install VirtualBox can be found at `https://www.virtualbox.org/`; again, VirtualBox can be installed on Linux, OS X, and Windows.

Cloning the environment

The source for the environment along with the practical examples can be found on GitHub in the Monitoring Docker repository at `https://github.com/russmckendrick/monitoring-docker`.

To clone the repository on a terminal on your local machine, run the following commands (replacing the file path as needed):

```
mkdir ~/Documents/Projects
cd ~/Documents/Projects/
git clone https://github.com/russmckendrick/monitoring-docker.git
```

Once cloned, you should see a directory called `monitoring-docker` and then enter that directory, as follows:

```
cd ~/Documents/Projects/monitoring-docker
```

Running a virtual server

In the repository, you will find two folders containing the necessary `Vagrant` file to launch either a CentOS 7 or a Ubuntu 14.04 virtual server.

If you would like to use the CentOS 7 vagrant box, change the directory to `vagrant-centos`:

```
cd vagrant-centos
```

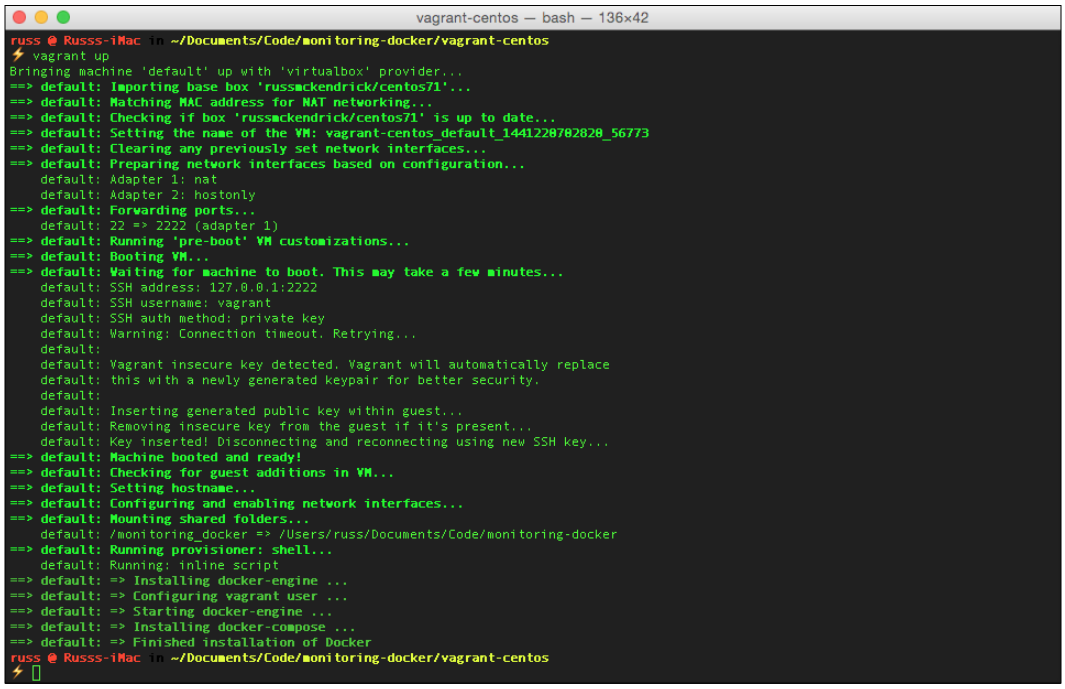
Once you are in the vagrant-centos directory, you will see that there is a `Vagrant` file; this file is all you need to launch a CentOS 7 virtual server. After the virtual server has been booted, the latest version of `docker` and `docker-compose` will be installed and the `monitoring-docker` directory will also be mounted inside the virtual machine using the mount point `/monitoring-docker`.

To launch the virtual server, simply type the following command:

```
vagrant up
```

This will download the latest version of the vagrant box from `https://atlas.hashicorp.com/russmckendrick/boxes/centos71` and then boot the virtual server; it's a 450 MB download so it may take several minutes to download; it only has to do this once.

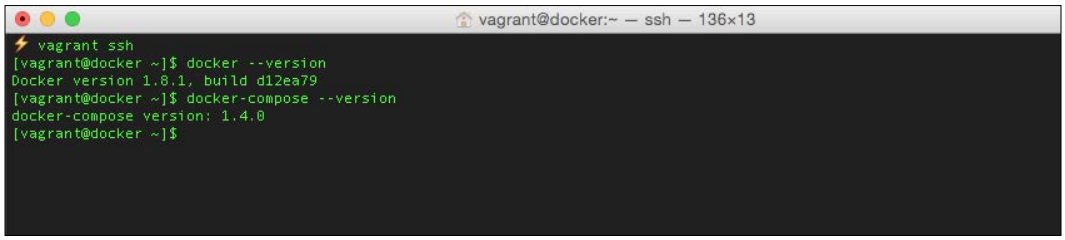
If all goes well, you should see something similar to the following output:



Now that you have booted the virtual server, you can connect to it using the following command:

```
vagrant ssh
```

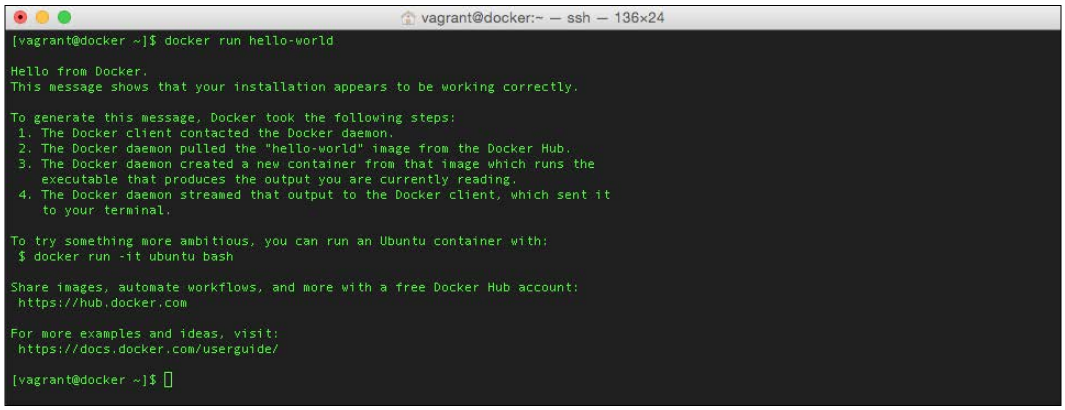
Once logged in, you should verify that `docker` and `docker-compose` are both available:



Finally, you can try running the `hello-world` container using the following command:

```
docker run hello-world
```

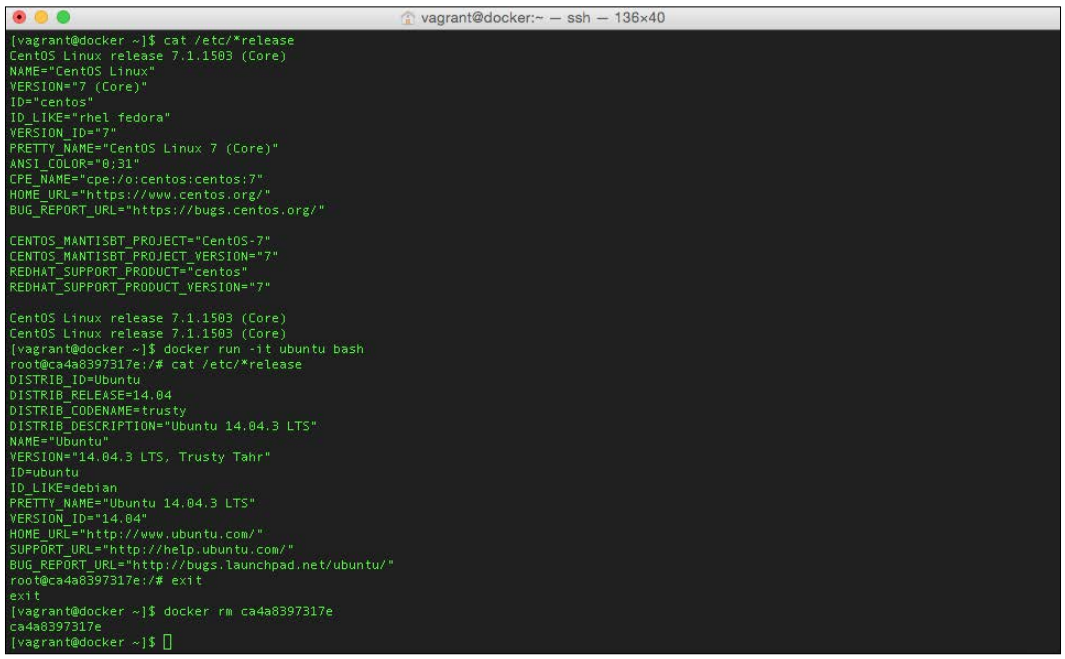
If everything goes as expected, you should see the following output:



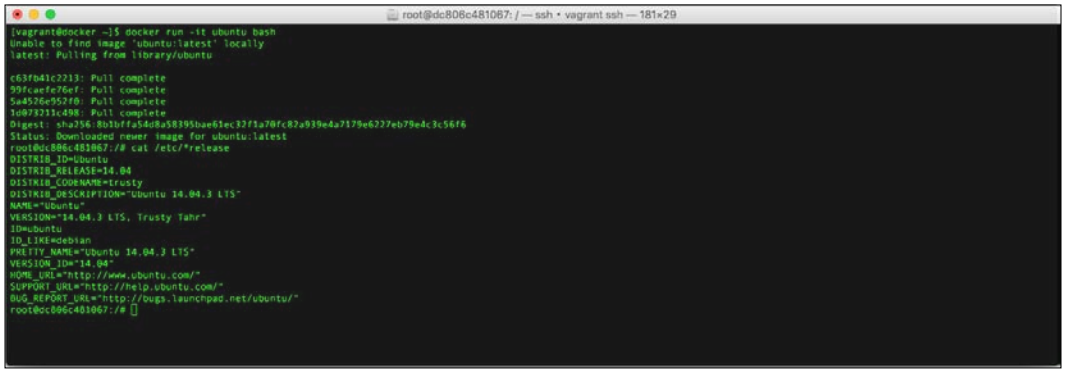
To try something more ambitious, you can run an Ubuntu container with the following command:

```
docker run -it ubuntu bash
```

Before we launch and enter the Ubuntu container, lets confirm that we are running the CentOS host machine by checking the release file that can be found in `/etc`:



Now, we can launch the Ubuntu container. Using the same command, we can confirm that we are inside the Ubuntu container by viewing its release file:



To exit the container just type in `exit`. This will stop the container from running, as it has terminated the only running process within the container, which was bash, and returned you to the host CentOS machine.

As you can see here from our CentOS 7 host, we have launched and removed an Ubuntu container.

Both the CentOS 7 and Ubuntu Vagrant files will configure a static IP address on your virtual machine. It is `192.168.33.10`; also, there is a DNS record for this IP address available at `docker.media-glass.es`. These will allow you to access any containers that expose themselves to a browser at either `http://192.168.33.10/` or `http://docker.media-glass.es/`.

The URL `http://docker.media-glass.es/` will only work while the vagrant box is up, and you have a container running which serves Web pages.

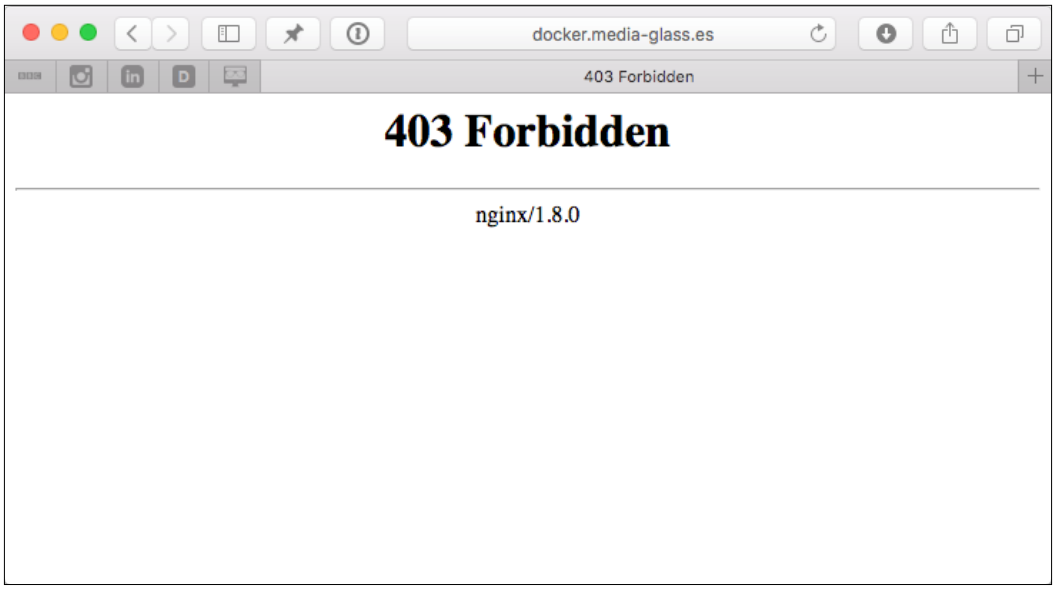
You can see this in action by running the following command:

```
docker run -d -p 80:80russmckendrick/nginx-php
```

Downloading the example code

The code files for this course are available at `https://github.com/EdwinMoses/Docker-Code`.

This will download and launch a container running NGINX. You can then go to `http://192.168.33.10/` or `http://docker.media-glass.es/` in your browser; you should see a forbidden page. This is because we have not yet given NGINX any content to serve (more on this will be covered later in the module):



For more examples and ideas, go to the website at `http://docs.docker.com/userguide/`.

Halting the virtual server

To log out of the virtual server and return to your local machine, you type `exit`.

You should now see your local machine's terminal prompt; however, the virtual server you booted will still be running in the background happily, using resources, until you either power it down using the following command:

```
vagrant halt
```

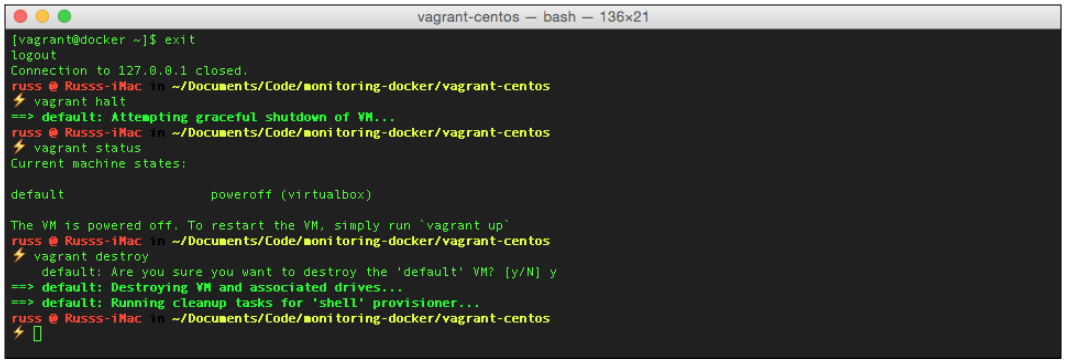
Terminate the virtual server altogether using `vagrant destroy`:

```
vagrant destroy
```

To check the current status of the virtual server, you can run the following command:

```
vagrant status
```

The result of the preceding command is given in the following output:



Either powering the virtual server back on or creating it from scratch again, can be achieved by issuing the `vagrant up` command again.

The preceding details show how to use the CentOS 7 vagrant box. If you would prefer to launch an Ubuntu 14.04 virtual server, you can download and install the vagrant box by going into the `vagrant-ubuntu` directory using the following command:

```
cd ~/Documents/Projects/monitoring-docker/vagrant-ubuntu
vagrant up
```

From here, you will be able run vagrant up and follow the same instructions used to boot and interact with the CentOS 7 virtual server.

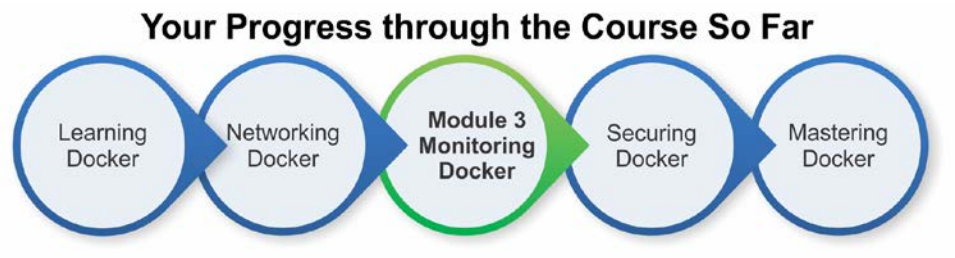
Summary of Module 3 Chapter 1

Ankita Thakur

Your Course Guide

In this chapter, we talked about different types of server and also discussed how your containerized applications can fit into each of the categories. We have also installed VirtualBox and used Vagrant to launch either a CentOS 7 or Ubuntu 14.04 virtual server, with docker and docker-compose installed.

Our new virtual server environment will be used throughout the upcoming chapters to test the various different types of monitoring. In the next chapter, we will start our journey by using Docker's in-built functionality to explore metrics about our running containers.



2
Using the Built-in Tools

In the later chapters of this module, we will explore the monitoring parts of the large eco-system that has started to flourish around Docker over the last 24 months. However, before we press ahead with that, we should take a look at what is possible with a vanilla installation of Docker. In this chapter, we will cover the following topics:

- Using the tools built into Docker to get real-time metrics on container performance
- Using standard operating system commands to get metrics on what Docker is doing
- Generating a test load so you can view the metrics changing

Docker stats

Since version 1.5, there has been a basic statistic command built into Docker:

```
docker stats --help

Usage: docker stats [OPTIONS] CONTAINER [CONTAINER...]

Display a live stream of one or more containers' resource usage
statistics

  --help=false         Print usage
  --no-stream=false    Disable streaming stats and only pull the first
result
```

This command will stream details of the resource utilization of your containers in real time. The best way to find out about the command is to see it in action.

Running Docker stats

Let's launch a container using the vagrant environment, which we covered in the last chapter:

```
[russ@mac ~]$ cd ~/Documents/Projects/monitoring-docker/vagrant-centos/
[russ@mac ~]$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'russmckendrick/centos71'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'russmckendrick/centos71' is up to date...

.....

==> default: => Installing docker-engine ...
==> default: => Configuring vagrant user ...
==> default: => Starting docker-engine ...
==> default: => Installing docker-compose ...
==> default: => Finished installation of Docker
[russ@mac ~]$ vagrant ssh
```

Now that you are connected to the vagrant server, launch the container using the Docker compose file in `/monitoring_docker/Chapter01/01-basic/`:

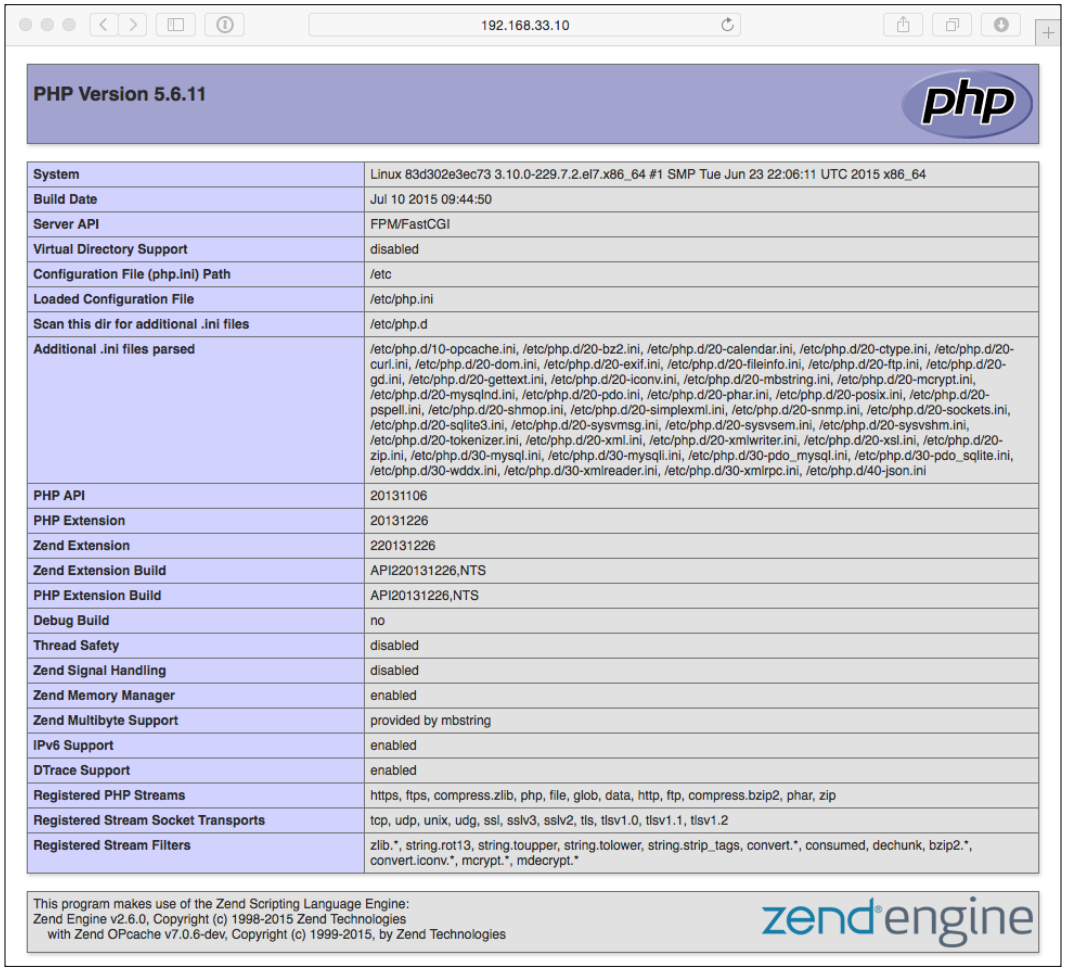
```
$ cd /monitoring_docker/Chapter01/01-basic/
$ docker-compose up -d
Creating 01basic_web_1...
```

You have now pulled down and launched a container in the background. The container is called `01basic_web_1` and it runs NGINX and PHP serving a single PHP information page (`http://php.net/manual/en/function.phpinfo.php`).

To check whether everything has been launched as expected, run `docker-compose ps`. You should see your single container with `State` of `Up`:

```
$ docker-compose ps
    Name              Command          State          Ports
-----------------------------------------------------------------
01basic_web_1   /usr/local/bin/run   Up      0.0.0.0:80->80/tcp
```

Finally, you should be able to see the page containing the output of the PHP information at `http://192.168.33.10/` (this IP address is hardcoded into the vagrant configuration), if you put it in your local browser:



Now, you have a container up and running; let's look at some of the basic stats. We know from the output of `docker-compose` that our container is called `01basic_web_1`, so enter the following command to start streaming statistics in your terminal:

```
docker stats 01basic_web_1
```

It will take a second to initiate; after this is done, you should see your container listed along with the statistics for the following:

- `CPU %`: This shows you how much of the available CPU resource the container is currently using.
- `MEM USEAGE/LIMIT`: This tells you how much RAM the container is utilizing; it also displays how much allowance the container has. If you haven't explicitly set a limit, it will show the total amount of RAM on the host machine.
- `MEM %`: This shows you what percentage of the RAM allowance the container is using.
- `NET I/O`: This gives a running total of how much bandwidth has been transferred in and out of the container.

If you go back to your browser window and start to refresh `http://192.168.33.10/`, you will see that the values in each of the columns start to change. To stop streaming the statistics, press *Ctrl* + *c*.

Rather than keeping on hitting refresh over and over again, let's generate a lot of traffic to `01basic_web_1`, which should put the container under a heavy load.

Here, we will launch a container that will send 10,000 requests to `01basic_web_1` using ApacheBench (`https://httpd.apache.org/docs/2.2/programs/ab.html`). Although it will take a minute or two to execute, we should run `docker stats` as soon as possible:

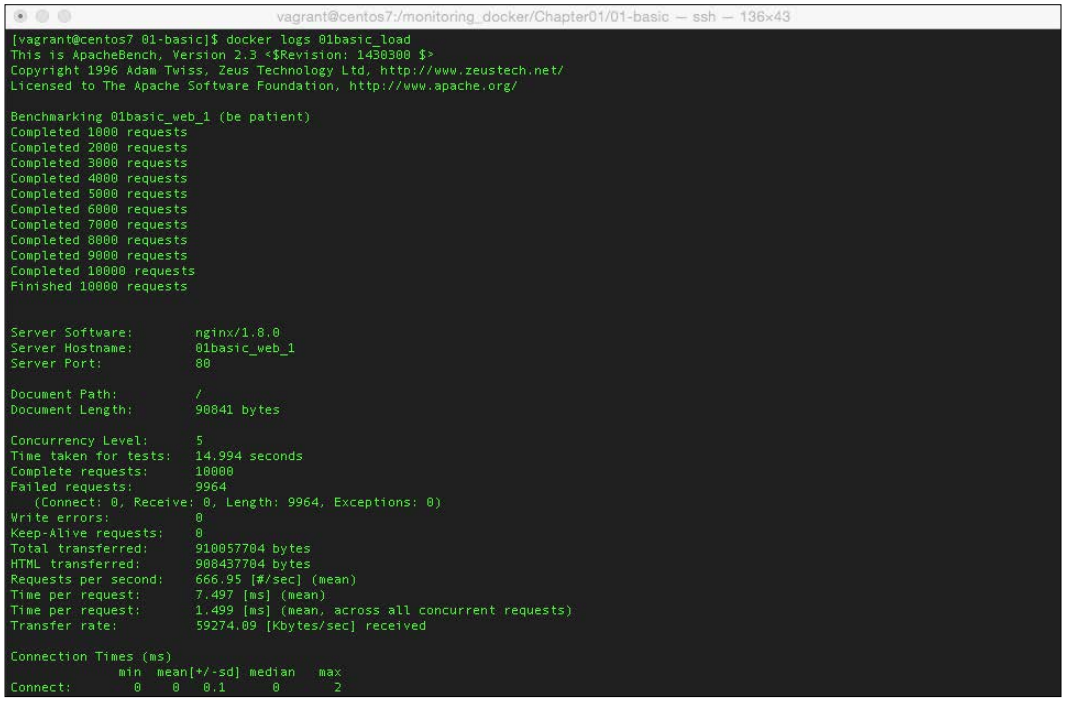
```
docker run -d --name=01basic_load --link=01basic_web_1 russmckendrick/ab
ab -k -n 10000 -c 5 http://01basic_web_1/ && docker stats 01basic_web_1
01basic_load
```

After the ApacheBench image has been downloaded and the container that will be called `01basic_load` starts, you should see the statistics for both `01basic_web_1` and `01basic_load` begin to stream in your terminal:

```
CONTAINER      CPU %     MEM USAGE/LIMIT     MEM %    NET I/O
01basic_load   18.11%    12.71 MB/1.905 GB   0.67%    335.2 MB/5.27 MB
01basic_web_1 139.62%    96.49 MB/1.905 GB   5.07%    5.27 MB/335.2 MB
```

After a while, you will notice that most of the statistics for `01basic_load` will drop off to zero; this means that the test has completed and that the container running the test has exited. The `docker stats` command can only stream statistics for the running containers; ones that have exited are no longer running and, therefore, do not produce output when running `docker stats`.

Exit from `docker stats` using *Ctrl + c*; to see the results of the ApacheBench command, you can type `docker logs 01basic_load`; you should see something like the following screenshot:



You shouldn't worry if you see any failures like in the preceding output. This exercise was purely to demonstrate how to view the statistics of the running containers and not to tune a web server to handle the amount of traffic we sent to it using ApacheBench.

To remove the containers that we launched, run the following commands:

```
$ docker-compose stop
Stopping 01basic_web_1...
$ docker-compose rm
Going to remove 01basic_web_1
Are you sure? [yN] y
Removing 01basic_web_1...
$ docker rm 01basic_load
01basic_load
```

To check whether everything has been removed successfully, run `docker ps -a` and you should not be able to see any running or exited containers that have `01basic_` in their names.

What just happened?

While running the ApacheBench test, you may have noticed that the CPU utilization on the container running NGINX and PHP was high; in the example in the previous section, it was using 139.62 percent of the available CPU resource.

As we did not attach any resource limits to the containers we launched, it was easy for our test to use all of the available resources on the host **Virtual Machine** (**VM**). If this VM was being used by several users, all running their own containers, they may have started to notice that their applications had started to slow down or, even worse, the applications had started showing errors.

If you ever find yourself in this situation, you can use `docker stats` to help track down the culprit.

Running `docker stats $(docker ps -q)` will stream the statistics for all the currently running containers:

```
CONTAINER       CPU %     MEM USAGE/LIMIT     MEM %    NET I/O
361040b7b33e    0.07%     86.98 MB/1.905 GB   4.57%    2.514 kB/738 B
56b459ae9092    120.06%   87.05 MB/1.905 GB   4.57%    2.772 kB/738 B
a3de616f84ba    0.04%     87.03 MB/1.905 GB   4.57%    2.244 kB/828 B
abdbee7b5207    0.08%     86.61 MB/1.905 GB   4.55%    3.69 kB/738 B
b85c49cf740c    0.07%     86.15 MB/1.905 GB   4.52%    2.952 kB/738 B
```

As you may have noticed, this displays the container ID rather than the name; this information should, however, be enough to spot the resource hog so that you can quickly stop it:

```
$ docker stop 56b459ae9092
56b459ae9092
```

Once stopped, you can then get the name of the rogue container by running the following command:

```
$ docker ps -a | grep 56b459ae9092
56b459ae9092        russmckendrick/nginx-php   "/usr/local/bin/run" 9
minutes ago       Exited (0) 26 seconds ago      my_bad_container
```

Alternatively, for more detailed information, you can run `docker inspect 56b459ae9092`, which will give you all the information you need on the container.

What about processes?

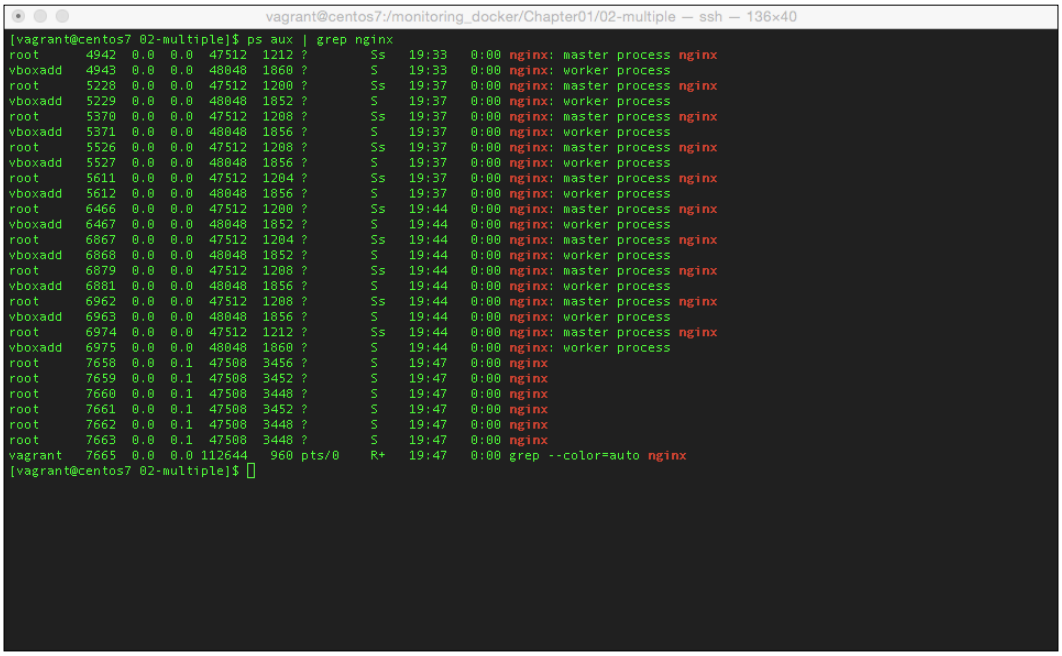
One of the great things about Docker is that it isn't really virtualization; as mentioned in the previous chapter, it is a great way of isolating processes rather than running an entire operating system.

This can get confusing when running tools such as `top` or `ps`. To get an idea just how confusing this can get, lets launch several containers using `docker-compose` and see for ourselves:

```
$ cd /monitoring_docker/Chapter01/02-multiple
$ docker-compose up -d
Creating 02multiple_web_1...
$ docker-compose scale web=5
Creating 02multiple_web_2...
Creating 02multiple_web_3...
Creating 02multiple_web_4...
Creating 02multiple_web_5...
Starting 02multiple_web_2...
Starting 02multiple_web_3...
Starting 02multiple_web_4...
Starting 02multiple_web_5...
```

Now, we have five web servers that have all been launched from the same image using the same configuration. One of the first things I do when logging into a server to troubleshoot a problem is run `ps -aux`; this will show all the running processes. As you can see, when running the command, there are a lot processes listed.

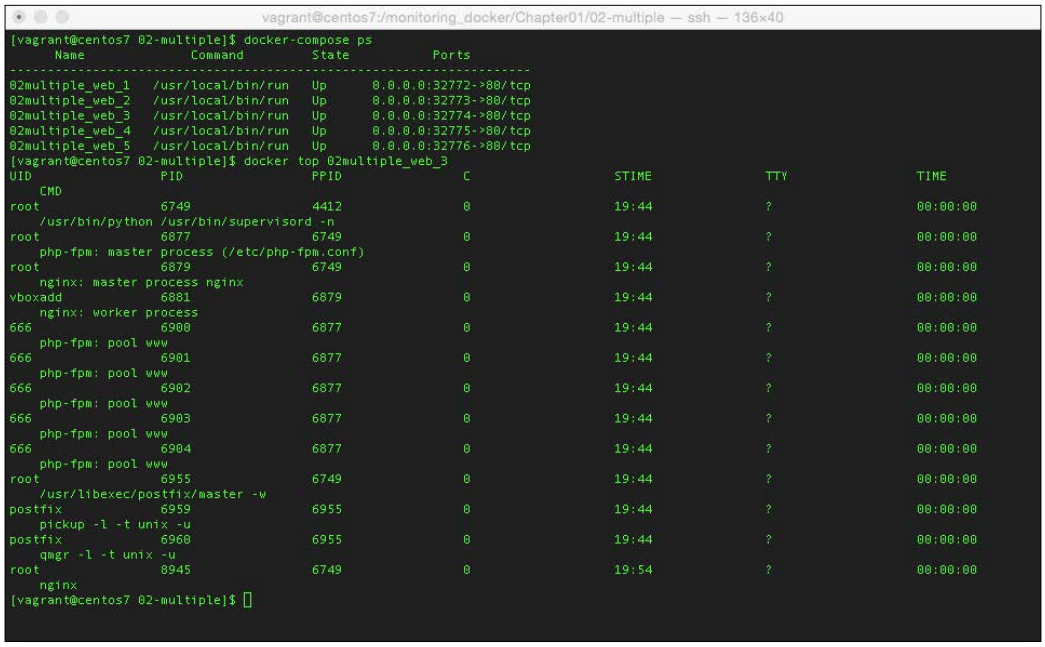
Even just trying to look at the processes for NGINX is confusing, as there is nothing to differentiate the processes from one container to another, as shown in the following output:



So, how can you know which container owns which processes?

Docker top

This command lists all the processes that are running within a container; think of it as a way of filtering the output of the `ps aux` command we ran on the host machine:



As `docker top` is an implementation of the standard `ps` command, any flags you would normally pass to `ps` should work as follows:

```
$ docker top 02multiple_web_3 –aux
$ docker top 02multiple_web_3 -faux
```

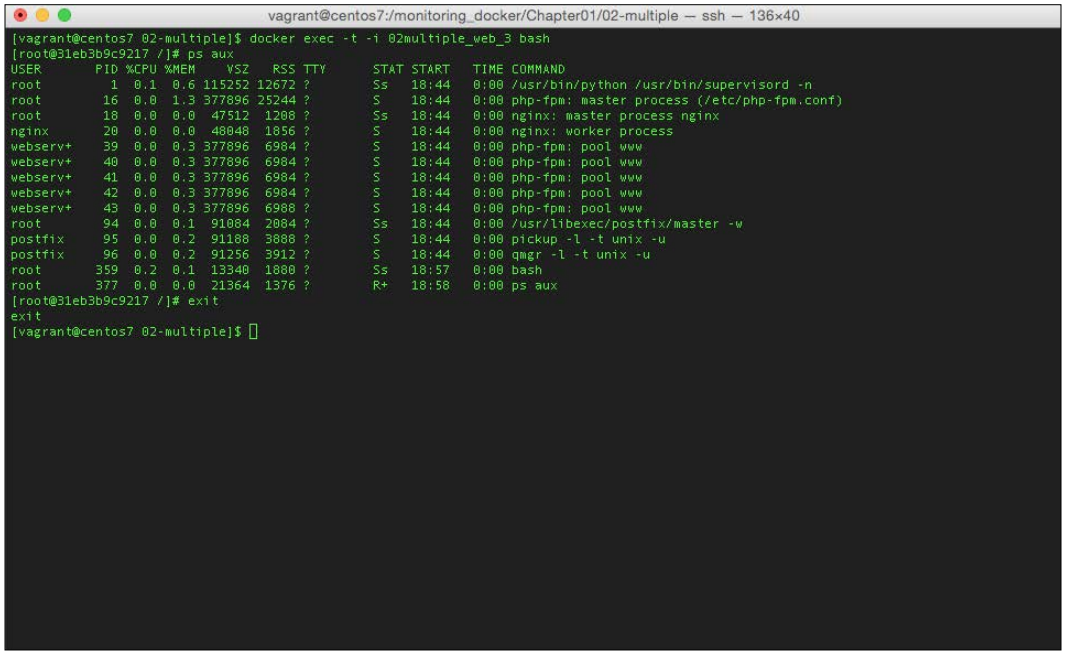
Docker exec

Another way to view what is going on within a container is to enter it. To enable you to do this, Docker introduced the `docker exec` command. This allows you to spawn an additional process within an already running container and then attach to the process; so, if we wanted to look at what is currently running on `02multiple_web_3`, we should use the following command spawn a bash shell within an already running container:

```
docker exec -t -i 02multiple_web_3 bash
```

Once you have an active shell on the container, you will notice that your prompt has changed to the container's ID. Your session is now isolated to the container's environment, meaning that you will only be able to interact with the processes belonging to the container you entered.

From here, you can run the `ps aux` or `top` command as you would do on the host machine, and only see the processes associated with the container you are interested in:



To leave the container, type in `exit`, you should see your prompt change back in your host machine.

Finally, you can stop and remove the containers by running `docker-compose stop` and `docker-compose kill`.

Summary of Module 3 Chapter 2

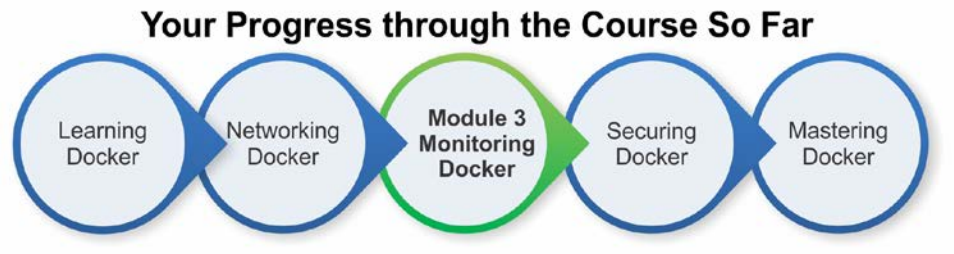
Ankita Thakur

Your Course Guide

In this chapter, we saw how we can get real-time statistics on our running containers and how we can use commands that are familiar to us, to get information on the processes that are launched as part of each container.

On the face of it, docker stats seems like a really basic piece of functionality that isn't really anything more than a tool to help you identify which container is using all the resources while a problem is occurring. However, the Docker command is actually pulling the information from a quite powerful API.

This API forms the basis for a lot of the monitoring tools we will be looking at in the next few chapters.



3
Advanced Container Resource Analysis

In the last chapter, we looked at how you can use the API built into Docker to gain an insight to what resources your containers are running. Now, we are to see how we can take it to the next level by using cAdvisor from Google. In this chapter, you will cover the following topics:

- How to install cAdvisor and start collecting metrics
- Learn all about the web interface and real-time monitoring
- What your options are for shipping metrics to a remote Prometheus database for long-term storage and trend analysis

What is cAdvisor?

Google describes cAdvisor as follows:

> *"cAdvisor (Container Advisor) provides container users an understanding of the resource usage and performance characteristics of their running containers. It is a running daemon that collects, aggregates, processes, and exports information about running containers. Specifically, for each container, it keeps resource isolation parameters, historical resource usage, histograms of complete historical resource usage, and network statistics. This data is exported by a container and is machine-wide."*

The project started off life as an internal tool at Google for gaining an insight into containers that had been launched using their own container stack.

Google's own container stack was called "Let Me Contain That For You" or lmctfy for short. The work on lmctfy has been installed as a Google port functionality over to libcontainer that is part of the Open Container Initiative. Further details on lmctfy can be found at `https://github.com/google/lmctfy/`.

cAdvisor is written in Go (`https://golang.org`); you can either compile your own binary or you can use the pre-compiled binary that are supplied via a container, which is available from Google's own Docker Hub account. You can find this at `http://hub.docker.com/u/google/`.

Once installed, cAdvisor will sit in the background and capture metrics that are similar to that of the `docker stats` command. We will go through these stats and understand what they mean later in this chapter.

cAdvisor takes these metrics along with those for the host machine and exposes them via a simple and easy-to-use built-in web interface.

Running cAdvisor using a container

There are a number of ways to install cAdvisor; the easiest way to get started is to download and run the container image that contains a copy of a precompiled cAdvisor binary.

Before running cAdvisor, let's launch a fresh vagrant host:

```
[russ@mac ~]$ cd ~/Documents/Projects/monitoring-docker/vagrant-centos/
[russ@mac ~]$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==>default: Importing base box 'russmckendrick/centos71'...
==>default: Matching MAC address for NAT networking...
==>default: Checking if box 'russmckendrick/centos71' is up to date...

.....

==>default: => Installing docker-engine ...
==>default: => Configuring vagrant user ...
==>default: => Starting docker-engine ...
==>default: => Installing docker-compose ...
==>default: => Finished installation of Docker
[russ@mac ~]$ vagrantssh
```

Using a backslash

As we have a lot options to pass to the `docker run` command, we are using `\` to split the command over multiple lines so it's easier to follow what is going on.

Once you have access to the host machine, run the following command:

```
docker run \
  --detach=true \
  --volume=/:/rootfs:ro \
  --volume=/var/run:/var/run:rw \
  --volume=/sys:/sys:ro \
  --volume=/var/lib/docker/:/var/lib/docker:ro \
  --publish=8080:8080 \
  --privileged=true \
  --name=cadvisor \
google/cadvisor:latest
```

You should now have a cAdvisor container up and running on your host machine. Before we start, let's look at cAdvisor in more detail by discussing why we have passed all the options to the container.

The cAdvisor binary is designed to run on the host machine alongside the Docker binary, so by launching cAdvisor in a container, we are actually isolating the binary in its down environment. To give cAdvisor access to the resources it requires on the host machine, we have to mount several partitions and also give the container privileged access to let the cAdvisor binary think it is being executed on the host machine.

When a container is launched with `--privileged`, Docker will enable full access to devices on the host machine; also, Docker will configure both AppArmor or SELinux to allow your container the same access to the host machine as a process running outside the container will have. For information on the `--privileged` flag, see this post on the Docker blog at `http://blog.docker.com/2013/09/docker-can-now-run-within-docker/`.

Compiling cAdvisor from source

As mentioned in the previous section, cAdvisor really ought to be executed on the host machine; this means, you may have to use a case to compile your own cAdvisor binary and run it directly on the host.

To compile cAdvisor, you will need to perform the following steps:

1. Install Go and Mercurial on the host machine—version 1.3 or higher of Go is needed to compile cAdvisor.
2. Set the path for Go to work from.
3. Grab the source code for cAdvisor and godep.
4. Set the path for your Go binaries.
5. Build the cAdvisor binary using godep to source the dependencies for us.
6. Copy the binary to `/usr/local/bin/`.
7. Download either an `Upstart` or `Systemd` script and launch the process.

If you followed the instructions in the previous section, you will already have a cAdvisor process running. Before compiling from source, you should start with a clean host; let's log out of the host and launch a fresh copy:

```
$ exit
logout
Connection to 127.0.0.1 closed.
[russ@mac ~]$ vagrant destroy
default: Are you sure you want to destroy the 'default' VM? [y/N] y
==>default: Forcing shutdown of VM...
==>default: Destroying VM and associated drives...
==>default: Running cleanup tasks for 'shell' provisioner...
[russ@mac ~]$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==>default: Importing base box 'russmckendrick/centos71'...
==>default: Matching MAC address for NAT networking...
==>default: Checking if box 'russmckendrick/centos71' is up to date...

.....

==>default: => Installing docker-engine ...
==>default: => Configuring vagrant user ...
```

```
==>default: => Starting docker-engine ...
==>default: => Installing docker-compose ...
==>default: => Finished installation of Docker
[russ@mac ~]$ vagrantssh
```

To build cAdvisor on the CentOS 7 host, run the following command:

```
sudo yum install -y golanggit mercurial
export GOPATH=$HOME/go
go get -d github.com/google/cadvisor
go get github.com/tools/godep
export PATH=$PATH:$GOPATH/bin
cd $GOPATH/src/github.com/google/cadvisor
godep go build .
sudocpcadvisor /usr/local/bin/
sudowgethttps://gist.githubusercontent.com/russmckendrick/
f647b2faad5d92c96771/raw/86b01a044006f85eebbe395d3857de1185ce4701/
cadvisor.service -O /lib/systemd/system/cadvisor.service
sudosystemctl enable cadvisor.service
sudosystemctl start cadvisor
```

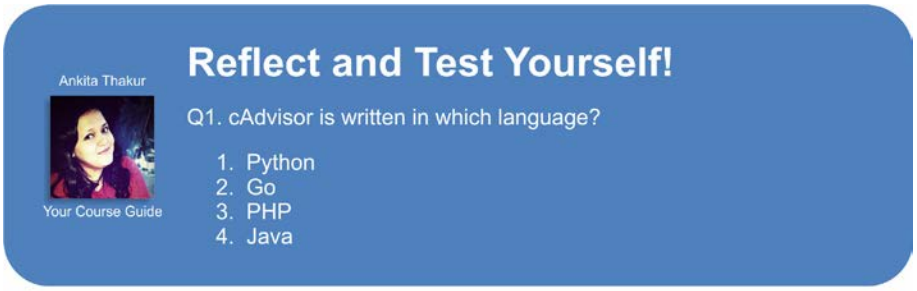
On the Ubuntu 14.04 LTS host, run the following command:

```
sudo apt-get -y install software-properties-common
sudo add-apt-repository ppa:evarlast/golang1.4
sudo apt-get update

sudo apt-get -y install golang mercurial

export GOPATH=$HOME/go
go get -d github.com/google/cadvisor
go get github.com/tools/godep
export PATH=$PATH:$GOPATH/bin
cd $GOPATH/src/github.com/google/cadvisor
godep go build .
sudocpcadvisor /usr/local/bin/
sudowgethttps://gist.githubusercontent.com/russmckendrick/
f647b2faad5d92c96771/raw/e12c100d220d30c1637bedd0ce1c18fb84beff77/
cadvisor.conf -O /etc/init/cadvisor.conf
sudo start cadvisor
```

You should now have a running cAdvisor process. You can check this by running `ps aux | grep cadvisor` and you should see a process with a path of `/usr/local/bin/cadvisor` running.

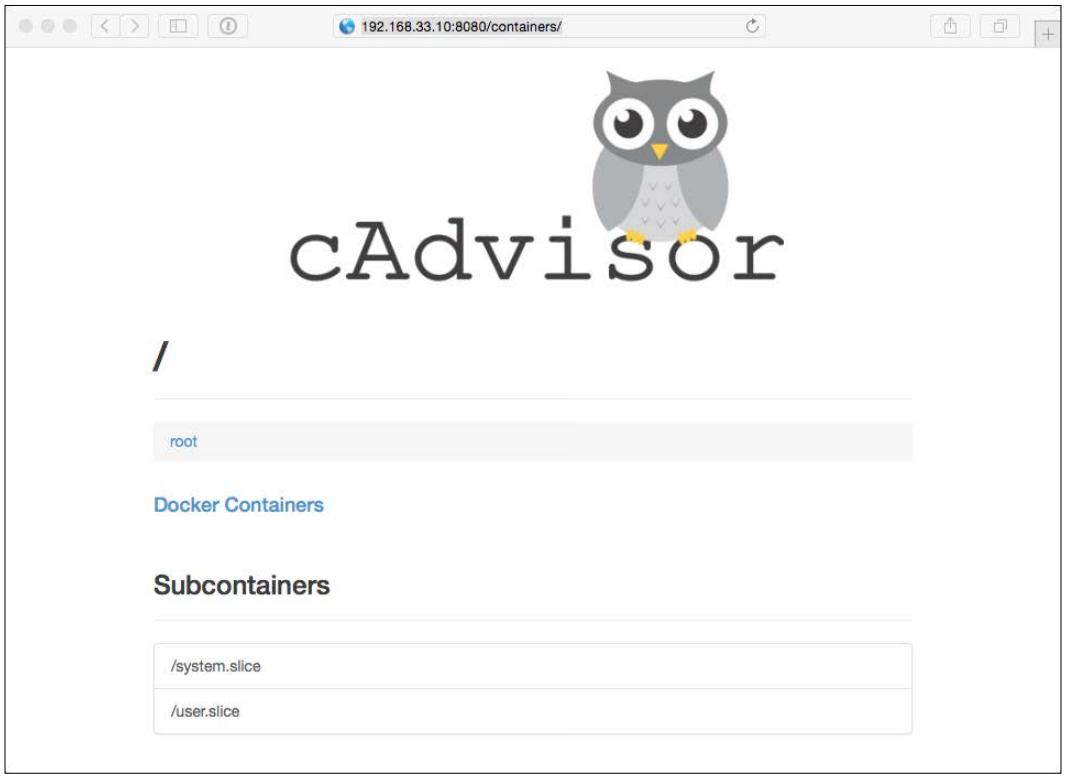
Collecting metrics

Now, you have cAdvisor running; what do you need to do to configure the service in order to start collecting metrics? The short answer is, nothing at all. When you started the cAdvisor process, it instantly started polling your host machine to find out what containers are running and gathered information on both the running containers and your host machine.



The Web interface

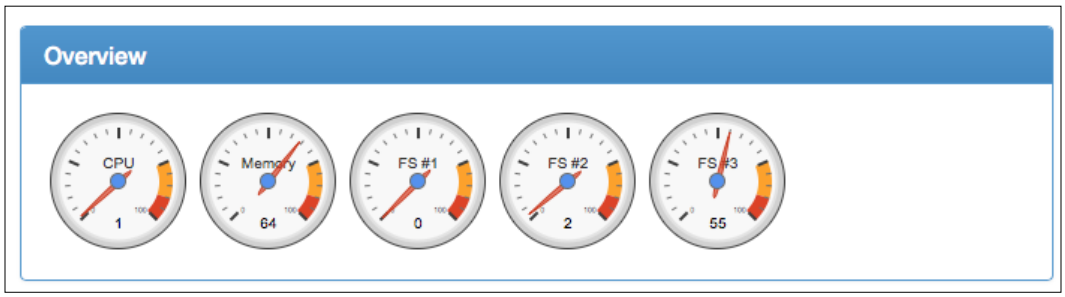
cAdvisor should be running on the `8080` port; if you open `http://192.168.33.10:8080/`, you should be greeted with the cAdvisor logo and an overview of your host machine:



This initial page streams live stats about the host machine, though each section is repeated when you start to drill down and view the containers. To start with, let's look at each section using the host information.

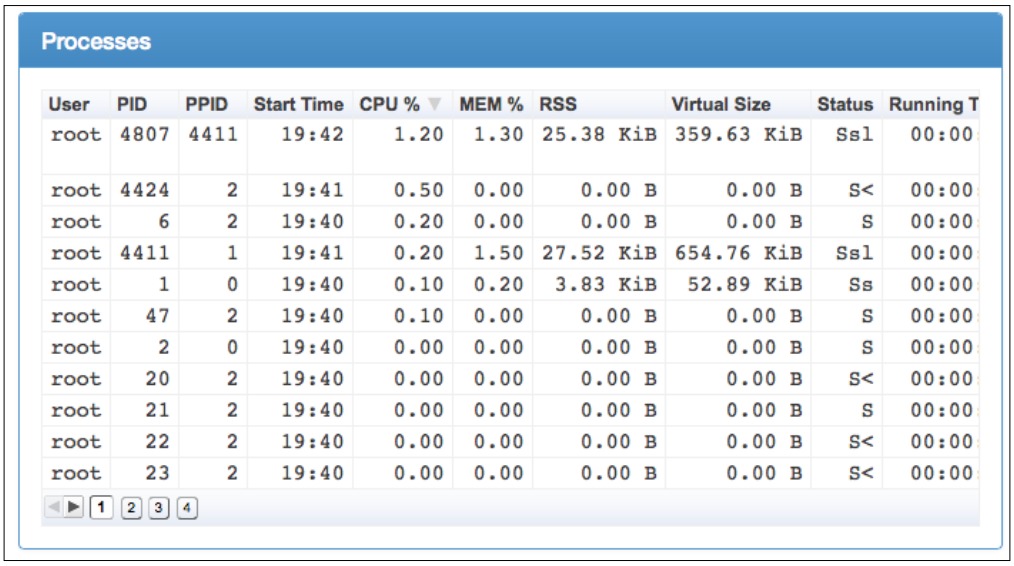
Overview

This overview section gives you a bird's-eye view of your system; it uses gauges so you can quickly get an idea of which resources are reaching their limits. In the following screenshot, there is very little in the way of CPU utilization and the file system usage is relatively low; however, we are using 64% of the available RAM:



Processes

The following screenshot displays a combined view of the output of the `ps aux`, `dockerps` and `top` commands we used in the previous chapter:



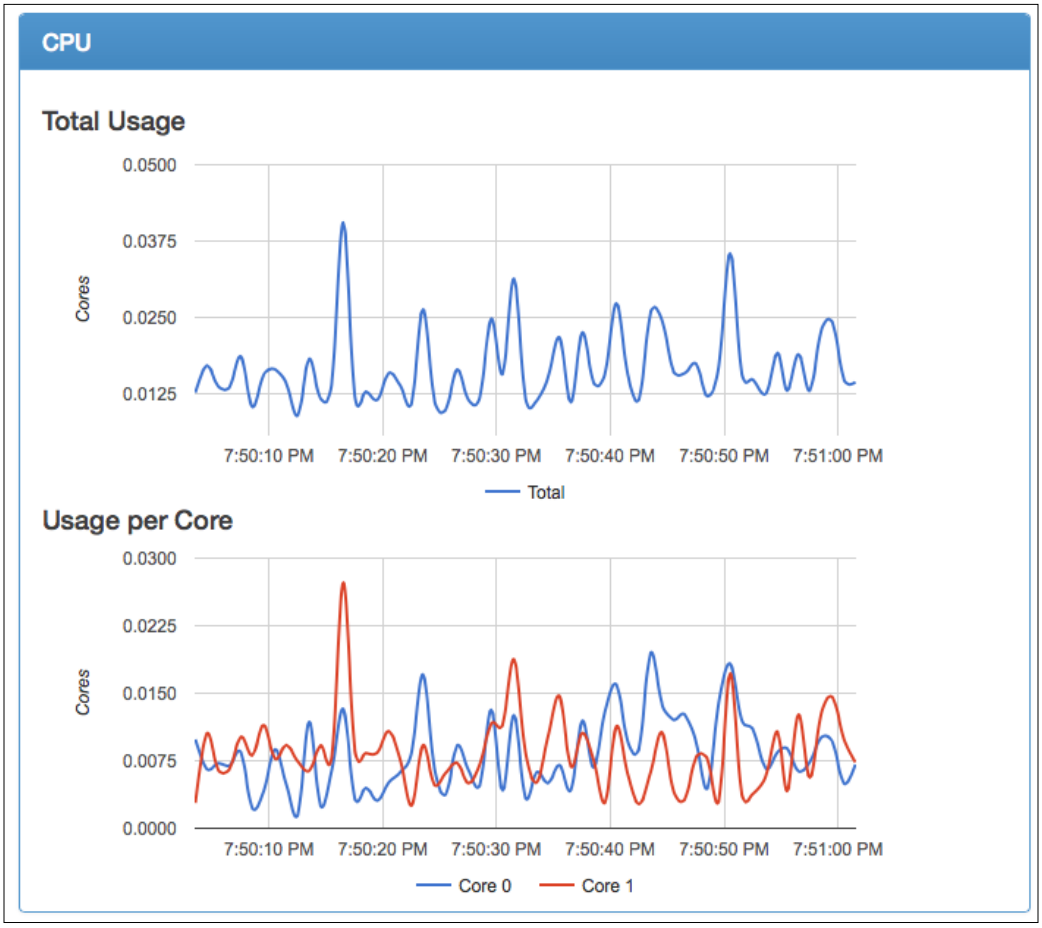
Here is what each column heading means:

- **User**: This shows which user is running the process
- **PID**: This is the unique process ID
- **PPID**: This is the **PID** of the parent process
- **Start Time**: This shows what time the process started
- **CPU %**: This is the percentage of the CPU the process is currently consuming
- **MEM %**: This is the percentage of the RAM the process is currently consuming
- **RSS**: This shows how much of the main memory the process is using
- **Virtual Size**: This shows how much of the virtual memory the process is using
- **Status**: This shows the current status of the process; this are the standard Linux process state codes
- **Running Time**: This shows how long the process has been running
- **Command**: This shows which command the process is running
- **Container**: This shows which container the process is attached to; the container listed as `/` is the host machine

As there could be several hundred processes active, this section is split into pages; you can navigate to these with the buttons on the bottom-left. Also, you can sort the processes by clicking on any of the headings.

CPU

The following graph shows the CPU utilization over the last minute:



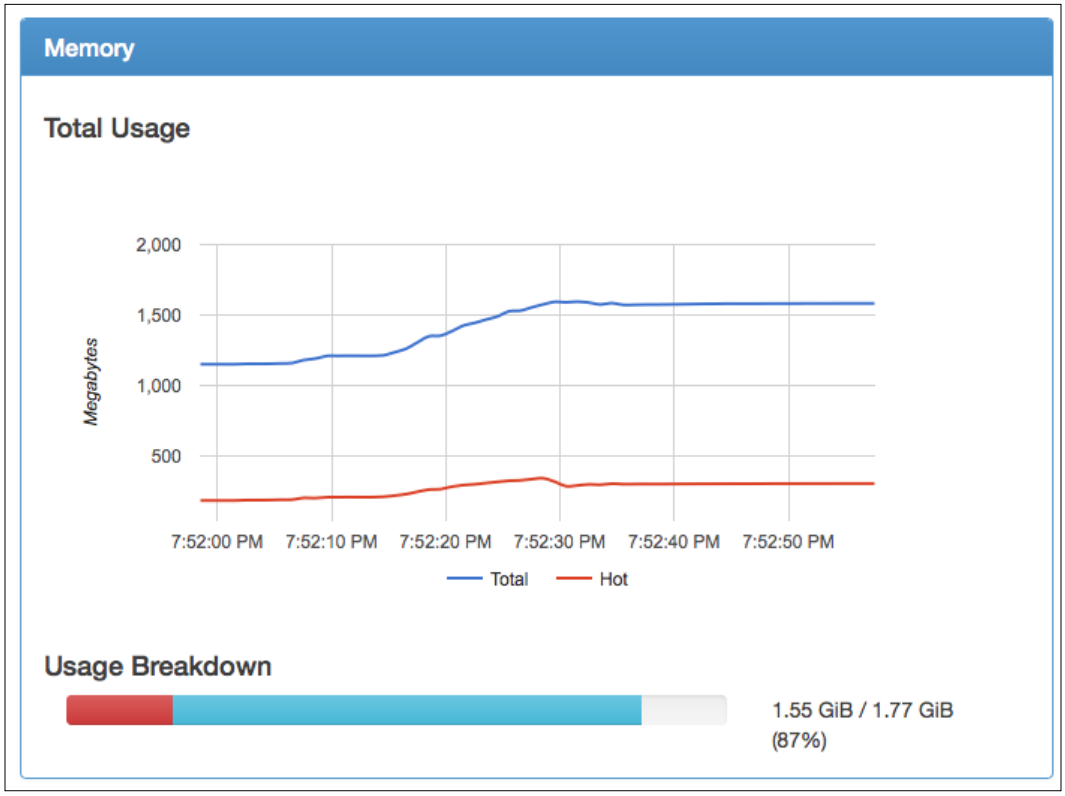
Here is what each term means:

- **Total Usage**: This shows an aggregate usage across all cores
- **Usage per Core**: This graph breaks down the usage per core
- **Usage Breakdown** (not shown in the preceding screenshot): This shows aggregate usage across all cores, but breaks it down to what is being used by the kernel and what is being used by the user-owned processes

Memory

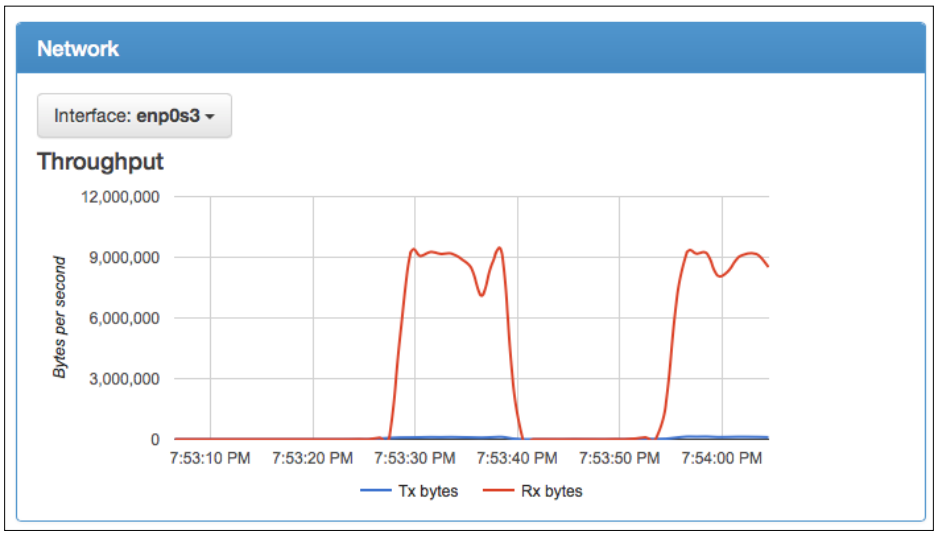
The **Memory** section is split into two parts. The graph tells you the total amount of memory used by all the processes for the host or container; this is the total of the hot and cold memory. The **Hot** memory is the current working set: pages that have been touched by the kernel recently. The **Cold** memory is the page that hasn't been touched for a while and could be reclaimed if needed.

The **Usage Breakdown** gives a visual representation of the total memory in the host machine, or allowance in the container, alongside the total and hot usage:



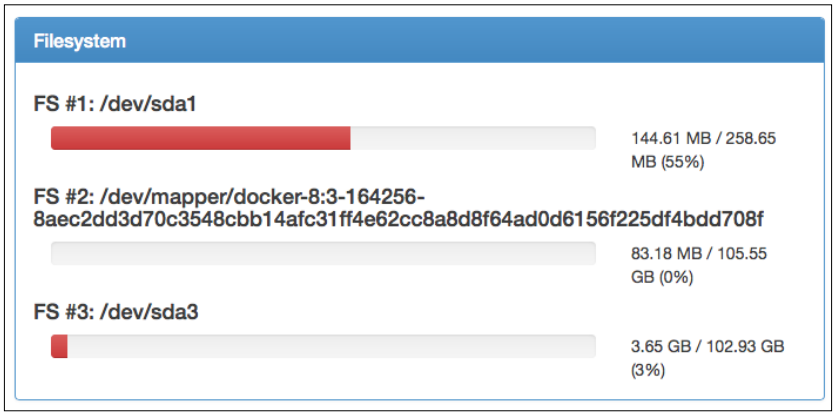
Network

This section shows the incoming and outgoing traffic over the last minute. You can change the interface using the drop-down box on the top-left. There is also a graph that shows any networking errors. Typically, this graph should be flat. If it isn't, then you will be seeing performance issues with your host machine or container:



Filesystem

The final section gives a break down of the filesystem usage. In the following screenshot, `/dev/sda1` is the boot partition, `/dev/sda3` is the main filesystem, and `/dev/mapper/docker-8...` is an aggregate of the write file systems of your running containers:



Viewing container stats

At the top of the page, there is a link of your running containers; you can either click on the link or go directly to `http://192.168.33.10:8080/docker/`. Once the page loads, you should see a list of all your running containers, and also a detailed overview of your Docker process, and finally a list of the images you have downloaded.

Subcontainers

Subcontainers shows a list of your containers; each entry is a clickable link that will take you to a page that will give you the following details:

- Isolation:
    - **CPU**: This shows you the CPU allowances of the container; if you have not set any resource limits, you will see the host's CPU information
    - **Memory**: This shows you the memory allowances of the container; if you have not set any resource limits, your container will show an unlimited allowance
- Usage:
    - **Overview**: This shows gauges so you can quickly see how close to any resource limits you are
    - **Processes**: This shows the processes for just your selected container
    - **CPU**: This shows the CPU utilization graphs isolated to just your container
    - **Memory**: This shows the memory utilization of your container

Driver status

The driver gives the basic stats on your main Docker process, along with the information on the host machine's kernel, host name, and also the underlying operating system.

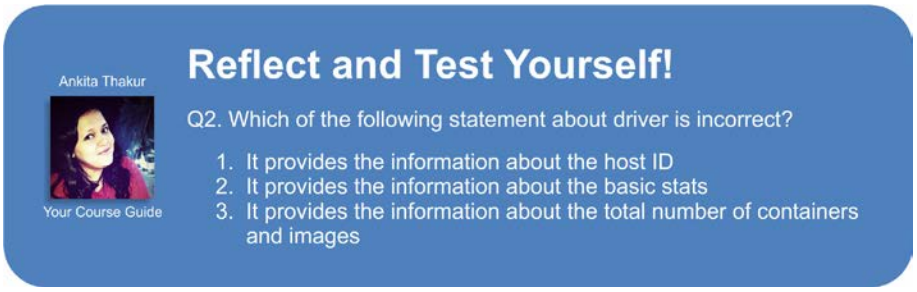
It also gives information on the total number of containers and images. You may notice that the total number of images is a much larger figure than you expected to see; this is because it is counting each file system as an individual image.

For more details on Docker images, see the Docker user guide at `https://docs.docker.com/userguide/dockerimages/`.

It also gives you a detailed breakdown of your storage configuration.

Images

Finally, you get a list of the Docker images which are available on the host machine. It lists the Repository, Tag, Size, and when the image was created, along with the images' unique ID. This lets you know where the image originated from (Repository), which version of the image you have downloaded (Tag) and how big the image is (Size).



This is all great, what's the catch?

So you are maybe thinking to yourself that all of this information available in your browser is really useful; being able to see real-time performance metrics in an easily readable format is a really plus.

The biggest drawback of using the web interface for cAdvisor, as you may have noticed, is that it only shows you one minute's worth of metrics; you can quite literally see the information disappearing in real time.

As a pane of glass gives a real-time view into your containers, cAdvisor is a brilliant tool; if you want to review any metrics that are older than one minute, you are out of luck.

That is, unless you configure somewhere to store all of your data; this is where Prometheus comes in.

Prometheus

So what's Prometheus? Its developers describe it as follows:

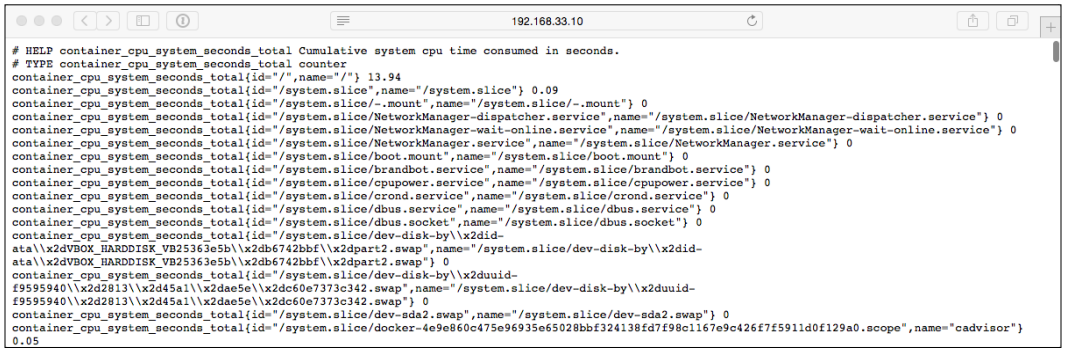
> *Prometheus is an open-source system's monitoring and alerting toolkit built at SoundCloud. Since its inception in 2012, it has become the standard for instrumenting new services at SoundCloud and is seeing growing external usage and contributions.*

OK, but what does that have to do with cAdvisor? Well, Prometheus has quite a powerful database backend that stores the data it imports as a time series of events.

Wikipedia describes a time series as follows:

> *"A time series is a sequence of data points, typically consisting of successive measurements made over a time interval. Examples of time series are ocean tides, counts of sunspots, and the daily closing value of the Dow Jones Industrial Average. Time series are very frequently plotted via line charts."*
>
> `https://en.wikipedia.org/wiki/Time_series`

One of the things cAdvisor does, by default, is expose all the metrics it is capturing on a single page at `/metrics`; you can see this at `http://192.168.33.10:8080/metrics` on our cAdvisor installation. The metrics are updated each time the page is loaded:



As you can see in the preceding screenshot, this is just a single long page of raw text. The way Prometheus works is that you configure it to scrape the `/metrics` URL at a user-defined interval, let's say every five seconds; the text is in a format that Prometheus understands and it is ingested into the Prometheus's time series database.

What this means is that, using Prometheus's powerful built-in query language, you can start to drill down into your data. Let's look at getting Prometheus up and running.

Launching Prometheus

Like cAdvisor there are several ways you can launch Prometheus. To start with, we will launch a container and inject our own configuration file so that Prometheus knows where our cAdvisor endpoint is:

```
docker run \
  --detach=true \
  --volume=/monitoring_docker/Chapter03/prometheus.yml:/etc/prometheus/
prometheus.yml \
  --publish=9090:9090 \
  --name=prometheus \
prom/prometheus:latest
```

Once you have launched the container, Prometheus will be accessible on the following URL: `http://192.168.33.10:9090`. When you first load the URL, you will be taken to a status page; this gives some basic information on the Prometheus installation. The important part of this page is the list of targets. This lists the URL that Prometheus will be scrapping to capture metrics; you should see your cAdvisor URL listed with a state of **HEALTHY**, as shown in the following screenshot:

Another information page contains the following:

- **Runtime information**: This displays how long Prometheus has been up and polling data, if you have configured an endpoint
- **Build information**: This contains the details of the version of Prometheus that you have been running
- **Configuration**: This is a copy of the configuration file we injected into the container when it was launched
- **Rules**: This is a copy of any rules we injected; these will be used for alerting
- **Startup flags**: This shows all the runtime variables and their values

Querying Prometheus

As we only have a few containers up and running at the moment, let's launch one that runs Redis so we can start to look at the query language built into Prometheus.

We will use the official Redis image for this and as we are only going to use this as an example we won't need to pass it any user variables:

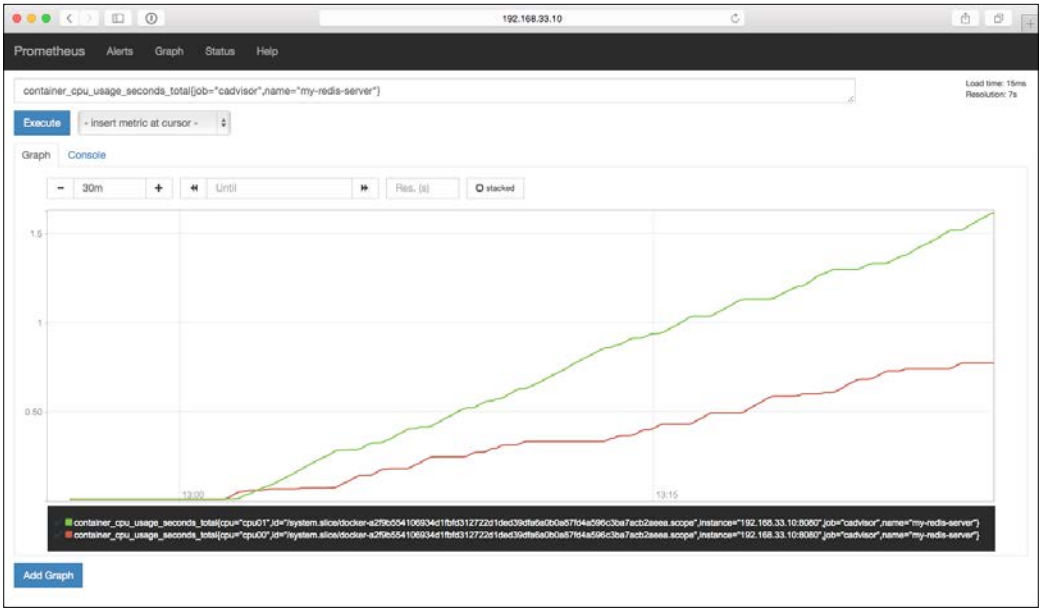
```
docker run --name my-redis-server -d redis
```

We now have a container called `my-redis-server` running. cAdvisor should already be exposing metrics about the container to Prometheus; let's go ahead and see. In the Prometheus web interface, go to the **Graph** link in the menu at the top of the page. Here, you will be presented with a text box into which you can enter your query. To start with, let's look at the CPU usage of the Redis container.

In the box, enter the following:

```
container_cpu_usage_seconds_total{job="cadvisor",name="my-redis-
server"}
```

Then, after clicking on **Execute**, you should have two results returned, listed in the **Console** tab of the page. If you remember, cAdvisor records the CPU usage of each of the CPU cores that the container has access to, which is why we have two values returned, one for "cpu00" and one for "cpu01". Clicking on the **Graph** link will show you results over a period of time:



As you can see in the preceding screenshot, we now have access to the usage graphs for the last 25 minutes, which is about how long ago I launched the Redis instance before generating the graph.

Dashboard

Also, when creating one of the graphs using the query tool in the main application, you can install a separate Dashboard application. This runs in a second container that connects to your main Prometheus container using the API as a data source.

Before we start the Dashboard container, we should initialize a SQLite3 database to store our configuration. To ensure that the database is persistent, we will store this on the host machine in `/tmp/prom/file.sqlite3`:

```
docker run \
  --volume=/tmp/prom:/tmp/prom \
  -e DATABASE_URL=sqlite3:/tmp/prom/file.sqlite3 \
prom/promdash ./bin/rake db:migrate
```

Once we have initialized the database, we can launch the Dashboard application properly:

```
docker run \
  --detach=true \
  --volume=/tmp/prom:/tmp/prom \
  -e DATABASE_URL=sqlite3:/tmp/prom/file.sqlite3 \
  --publish=3000:3000  \
  --name=promdash \
prom/promdash
```

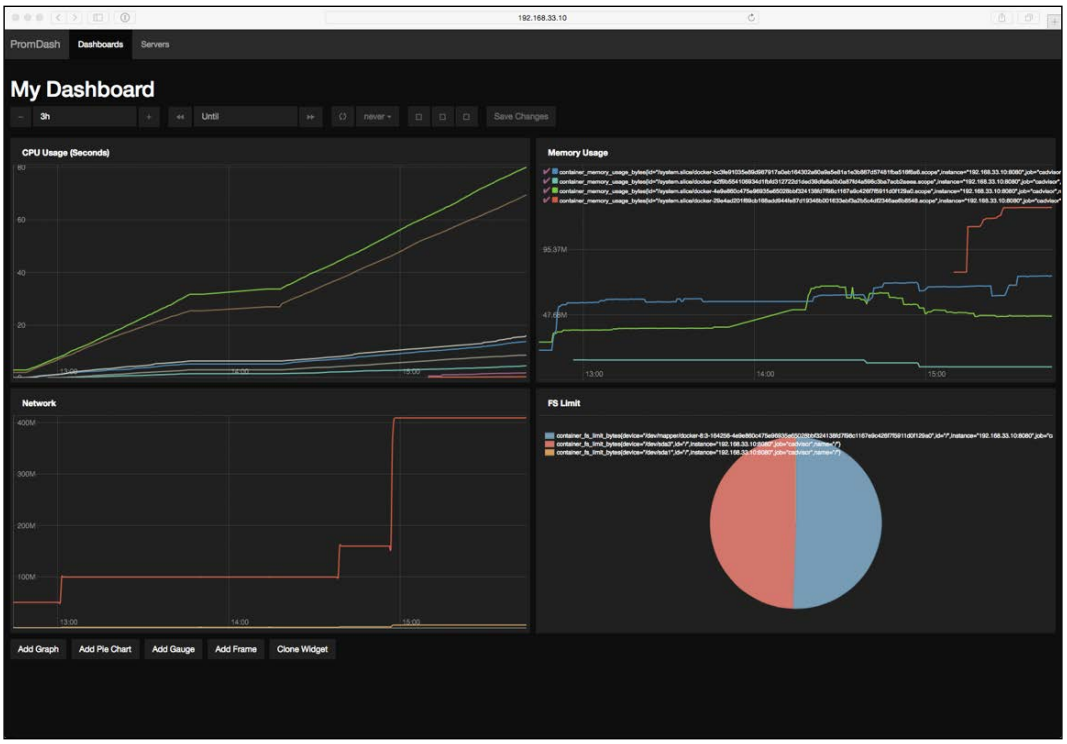
The application should now be accessible at `http://192.168.33.10:3000/`. The first thing we need to do is set up the data source. To do this, click on the **Servers** link at the top of the screen and then click on **New Server**. Here, you will be asked to provide the details of your Prometheus server. Name the server and enter the following URL:

- **Name**: `cAdvisor`
- **URL**: `http://192.168.33.10:9090`
- **Server Type**: `Prometheus`

Once you click on **Create Server**, you should receive a message saying **Server was successfully created**. Next up, you need to create a `directory`; this is where your dashboards will be stored.

Click on the **Dashboards** link in the top menu and then click on **New directory** and create one called `Test directory`. Now, you are ready to start creating Dashboards. Click on **New Dashboard**, call it **My Dashboard**, place it in `Test directory`. Once you click on **Create Dashboard**, you will be taken to the preview screen.

From here, you can build up dashboards using the control in the top right-hand side of each section. To add data, you simply enter the query you would like to see in the dashboard section:



For detailed information on how to create Dashboards, see the **PROMDASH** section of the Prometheus documentation at `http://prometheus.io/docs/visualization/promdash/`.

The next steps

At the moment, we are running Prometheus in a single container and its data is being stored within that same container. This means, if for any reason the container is terminated, our data is lost; it also means that we can't upgrade without loosing out data. To get around this problem, we can create a data volume container.

A data volume container is a special type of container that only exists as storage for other containers. For more details, see the Docker user guide at `https://docs.docker.com/userguide/dockervolumes/#creating-and-mounting-a-data-volume-container`.

First of all, let's make sure we have removed all the running Prometheus containers:

```
docker stop prometheus&&dockerrm Prometheus
```

Next up, let's create a data container called `promdata`:

```
docker create \
  --volume=/promdata \
  --name=promdata \
prom/prometheus /bin/true
```

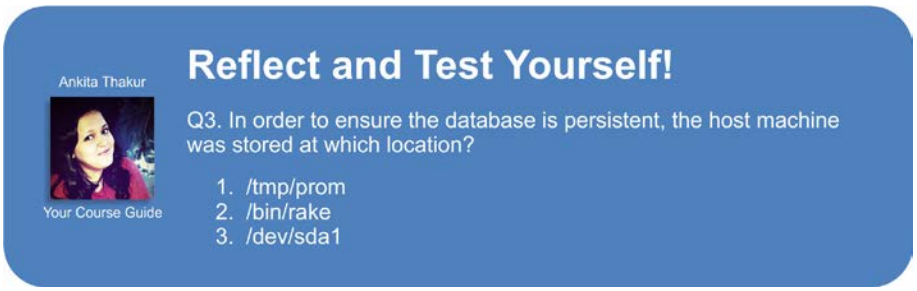
Finally, launch Prometheus again, this time, using the data container:

```
docker run \
  --detach=true \
  --volumes-from promdata \
  --volume=/monitoring_docker/Chapter03/prometheus.yml:/etc/prometheus/
prometheus.yml \
  --publish=9090:9090 \
  --name=prometheus \
prom/prometheus
```

This will ensure that, if you have to upgrade or relaunch your container, the metrics you have been capturing are safe and sound.

We have only touched on the basics of using Prometheus in this section of the module; for further information on the application, I recommend the following links as a good starting point:

- Documentation: `http://prometheus.io/docs/introduction/overview/`
- Twitter: `https://twitter.com/PrometheusIO`
- Project page: `https://github.com/prometheus/prometheus`
- Google groups: `https://groups.google.com/forum/#!forum/prometheus-developers`



Alternatives?

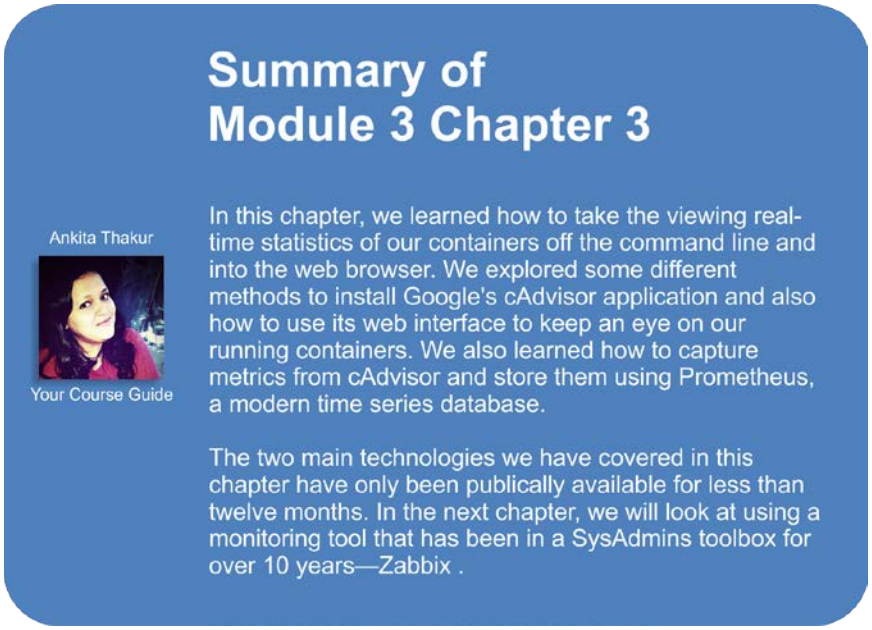
There are some alternatives to Prometheus. One such alternative is InfluxDB that describes itself as follows:

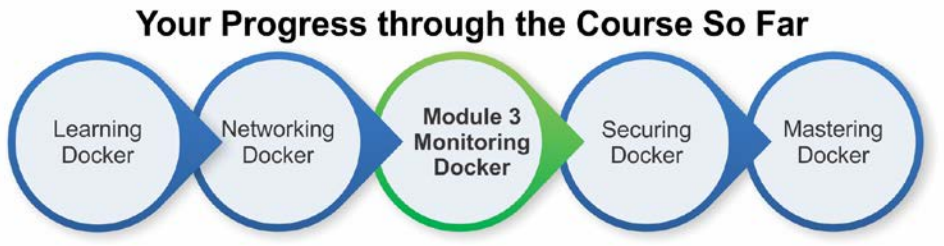
> *An open-source distributed time series database with no external dependencies.*

However, at the time of writing, cAdvisor is not currently compatible with the latest version of InfluxDB. There are patches in the codebase for cAdvisor; however, these are yet to make it through to the Google-maintained Docker Image.

For more details on InfluxDB and it's new visualization complain application Chronograf, see the project website at `https://influxdb.com/` and for more details on how to export cAdvisor statistics to InfluxDB, see the supporting documentation for cAdvisor at `https://github.com/google/cadvisor/tree/master/docs`.


Ankita Thakur
Your Course Guide
Your Coding Challenge
cAdvisor supports exporting the performance matrices to influxdb (http://influxdb.com/). Heapster (https://github.com/GoogleCloudPlatform/heapster) is another project from Google, which allows cluster-wide (Kubernetes) monitoring using cAdvisor. Try researching about Heapster and how to use it.


Summary of
Module 3 Chapter 3
Ankita Thakur
Your Course Guide
In this chapter, we learned how to take the viewing real-time statistics of our containers off the command line and into the web browser. We explored some different methods to install Google's cAdvisor application and also how to use its web interface to keep an eye on our running containers. We also learned how to capture metrics from cAdvisor and store them using Prometheus, a modern time series database.
The two main technologies we have covered in this chapter have only been publically available for less than twelve months. In the next chapter, we will look at using a monitoring tool that has been in a SysAdmins toolbox for over 10 years—Zabbix .


Your Progress through the Course So Far
Learning Docker
Networking Docker
Module 3 Monitoring Docker
Securing Docker
Mastering Docker

4
A Traditional Approach to Monitoring Containers

So far, we have looked at only a few technologies to monitor our containers, so in this chapter, we will be looking more at a traditional tool for monitoring services. By the end of this chapter, you should know your way around Zabbix and the various ways you can monitor your containers. We will cover the following topics in this chapter:

- How to run a Zabbix Server using containers
- How to launch a Zabbix Server on a vagrant machine
- How to prepare our host system for monitoring containers using the Zabbix agent
- How to find your way around the Zabbix web interface

Zabbix

First things first, what is Zabbix and why use it?

I have personally been using it since version 1.2; the Zabbix site describes it as follows:

> *"With Zabbix, it is possible to gather virtually limitless types of data from the network. High-performance real-time monitoring means that tens of thousands of servers, virtual machines, and network devices can be monitored simultaneously. Along with storing the data, visualization features are available (overviews, maps, graphs, screens, and so on), as well as very flexible ways of analyzing the data for the purpose of alerting.*

> *Zabbix offers great performance for data gathering and can be scaled to very large environments. Distributed monitoring options are available with the use of Zabbix proxies. Zabbix comes with a web-based interface, secure user authentication, and a flexible user permission schema. Polling and trapping is supported, with native high-performance agents gathering data from virtually any popular operating system; agent-less monitoring methods are available as well."*

At the time I started using Zabbix, the only real viable options were as follows:

- Nagios: `https://www.nagios.org/`
- Zabbix: `http://www.zabbix.com/`
- Zenoss: `http://www.zenoss.org/`

Out of the these three options, Zabbix seemed to be the most straightforward one at the time. It was doing enough work to manage the several hundred servers I was going to monitor without having to have the extra work of learning the complexities of setting up Nagios or Zenoss; after all, given the task the software had, I needed to be able to trust that I had set it up correctly.

In this chapter, while I am going to go into some detail about the setup and the basics of using Zabbix, we will only be touching on some of the functionalities, which can do a lot more than just monitor your containers. For more information, I would recommend the following as a good starting point:

- Zabbix blog: `http://blog.zabbix.com`
- Zabbix 2.4 manual: `https://www.zabbix.com/documentation/2.4/manual`
- Further reading: `https://www.packtpub.com/all/?search=zabbix`

Installing Zabbix

As you may have noticed from the links in the previous section, there are a lot of moving parts in Zabbix. It leverages several open source technologies, and a production-ready installation needs a little more planning than we can go into in this chapter. Because of this we are going to look at two ways of installing Zabbix quickly rather go into too much detail.

Using containers

At the time of writing, there are over a hundred Docker images available on the Docker Hub (`https://hub.docker.com`) that mentions Zabbix. These range from full server installations to just the various parts, such as the Zabbix agent or proxy services.

Out of the ones listed, there is one that is recommend by Zabbix itself. So, we will look at this one; it can be found at the following URLs:

- Docker Hub: `https://hub.docker.com/u/zabbix/`
- Project page: `https://github.com/zabbix/zabbix-community-docker`

To get the `ZabbixServer` container up and running, we must first launch a database container. Let's start afresh with our vagrant instance by running the following command:

```
[russ@mac ~]$ cd ~/Documents/Projects/monitoring-docker/vagrant-centos/
[russ@mac ~]$ vagrant destroy
default: Are you sure you want to destroy the 'default' VM? [y/N] y
==>default: Forcing shutdown of VM...
==>default: Destroying VM and associated drives...
==>default: Running cleanup tasks for 'shell' provisioner...
[russ@mac ~]$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==>default: Importing base box 'russmckendrick/centos71'...
==>default: Matching MAC address for NAT networking...
==>default: Checking if box 'russmckendrick/centos71' is up to date...

.....

==>default: => Installing docker-engine ...
==>default: => Configuring vagrant user ...
==>default: => Starting docker-engine ...
==>default: => Installing docker-compose ...
==>default: => Finished installation of Docker
[russ@mac ~]$ vagrantssh
```

Now, we have a clean environment and it's time to launch our database container, as follows:

```
docker run \
  --detach=true \
  --publish=3306 \
  --env="MARIADB_USER=zabbix" \
  --env="MARIADB_PASS=zabbix_password" \
  --name=zabbix-db \
million12/mariadb
```

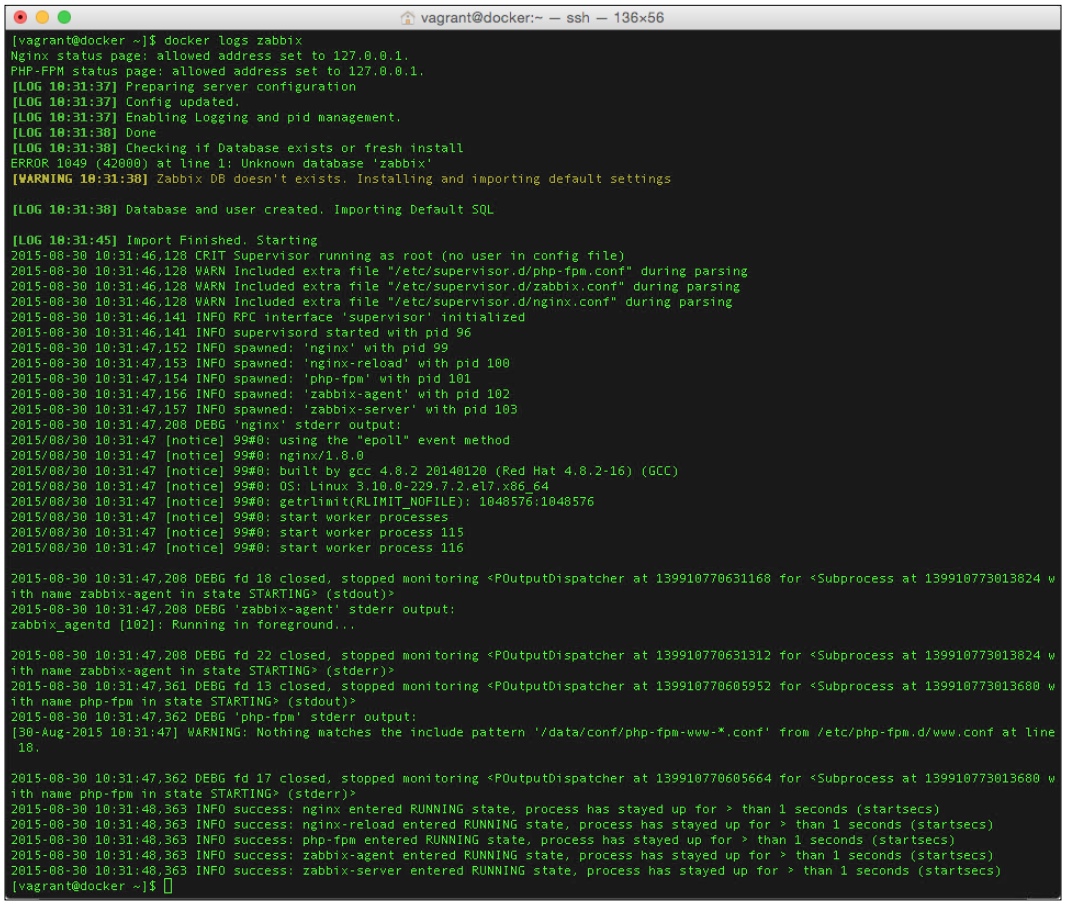
This will download the `million12/mariadb` image from `https://hub.docker.com/r/million12/mariadb/` and launch a container called `zabbix-db`, running MariaDB 10 (`https://mariadb.org`) with a user called `zabbix` who has a password `zabbix_password`. We have also opened the MariaDB port `3306` up on the container, but as we will be connecting to it from a linked container, there is no need to expose that port on the host machine.

Now, we have the database container up and running, we now need to launch our Zabbix Server container:

```
docker run \
  --detach=true \
  --publish=80:80 \
  --publish=10051:10051 \
  --link=zabbix-db:db \
  --env="DB_ADDRESS=db" \
  --env="DB_USER=zabbix" \
  --env="DB_PASS=zabbix_password" \
  --name=zabbix \
zabbix/zabbix-server-2.4
```

This downloads the image, which at the time of writing is over 1 GB so this process could take several minutes depending on your connection, and launches a container called `zabbix`. It maps the web server (port `80`) and the Zabbix Server process (port `10051`) on the host to the container, creates a link to our database container, sets up the alias `db`, and injects the database credentials as environment variables so that the scripts that launch when the container boots can populate the database.

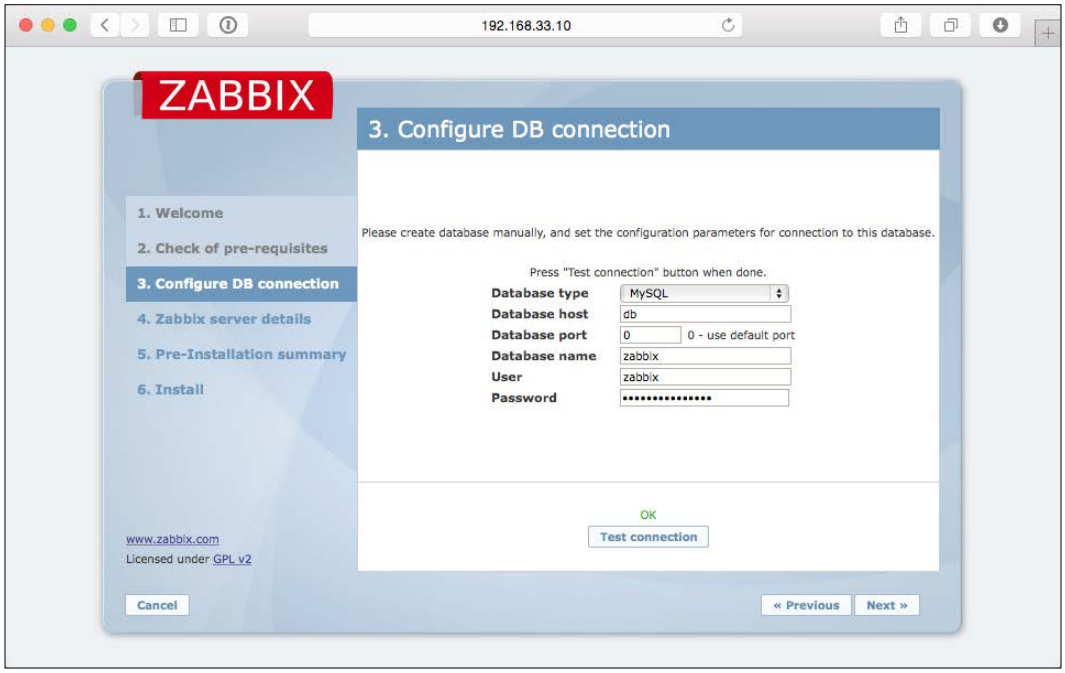
You can verify that everything worked as expected by checking the logs on the container. To do this, enter `docker logs zabbix`. This will print details of what happened when the container launched on screen:



Now, once we have the container up and running, it is time to move to the browser for our first taste of the web interface. Go to `http://192.168.33.10/` in your browser and you will be greeted by a welcome page; before we can start using Zabbix, we need to complete the installation.

On the welcome page, click on **Next** to be taken to the first step. This will verify that everything we need to run a Zabbix Server is installed. As we have launched it in a container, you should see **OK** next to all of the prerequisites. Click on **Next** to move onto the next step.

Now, we need to configure the database connection for the web interface. Here, you should have the same details as you did when you launched the container, as illustrated in the following screenshot:



Once you have entered the details, click on **Test connection** and you should receive an **OK** message; you will not be able to proceed until this test completes successfully. Once you have entered the details and have an **OK** message, click on **Next**.

Next up, are the details on the Zabbix Server that the web interface needs to connect to; click on **Next** here. Next up, you will receive a summary of the installation. To proceed, click on **Next** and you will be get confirmation that the `/usr/local/src/zabbix/frontends/php/conf/zabbix.conf.php` file has been created. Click on **Finish** to be taken to the login page.

Using vagrant

While writing this chapter, I thought a lot about providing another set of installation instructions for the Zabbix Server service. While the module is all about Monitoring Docker containers, having a service as resource intensive as Zabbix running inside a container feels a little counter intuitive. Because of this, there is a vagrant machine that uses Puppet to bootstrap a working installation of Zabbix Server:

```
[russ@mac ~]$ cd ~/Documents/Projects/monitoring-docker/vagrant-zabbix/
[russ@mac ~]$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==>default: Importing base box 'russmckendrick/centos71'...
==>default: Matching MAC address for NAT networking...
==>default: Checking if box 'russmckendrick/centos71' is up to date...

.....

==>default: Debug: Received report to process from zabbix.media-glass.es
==>default: Debug: Evicting cache entry for environment 'production'
==>default: Debug: Caching environment 'production' (ttl = 0 sec)
==>default: Debug: Processing report from zabbix.media-glass.es with
processor Puppet::Reports::Store
```

As you may have noticed, there is a lot of output streamed to the terminal, so what just happened? First of all, a CentOS 7 vagrant instance was launched and then a Puppet agent was installed. Once installed, the installation was handed off to Puppet. Using the Zabbix Puppet module by Werner Dijkerman, Zabbix Server was installed; for more details on the module, see its Puppet Forge page at `https://forge.puppetlabs.com/wdijkerman/zabbix`.

Unlike the containerized version of Zabbix Server, there is no additional configuration required, so you should be able to access the Zabbix login page at `http://zabbix.media-glass.es/` (an IP address of `192.168.33.11` is hardcoded into the configuration).

Preparing our host machine

For the remainder of this chapter, I will assume that you are using the Zabbix Server that is running on its own vagrant instance. This helps to ensure that your environment is consistent with the configuration of the Zabbix agent we will be looking at.

To pass the statistics from our containers to the Zabbix agent, which will then in turn expose them to the Zabbix Server, we will be installing using the `Zabbix-Docker-Monitoring` Zabbix agent module that has been developed by Jan Garaj. For more information on the project, see the following URLs:

- The Project page: `https://github.com/monitoringartist/Zabbix-Docker-Monitoring/`
- The Zabbix share page: `https://share.zabbix.com/virtualization/docker-containers-monitoring`

To get the agent and module installed, configured, and running, we need to execute the following steps:

1. Install the Zabbix package repository.
2. Install the Zabbix agent.
3. Install the prerequisites for the module.
4. Add the Zabbix agent user to the Docker group.
5. Download the auto-discovery bash script.
6. Download the precompiled `zabbix_module_docker` binary.
7. Configure the Zabbix agent with the details of our Zabbix Server and also the Docker module.
8. Set the correct permissions on all the files we have downloaded and created.
9. Start the Zabbix agent.

While the steps remain the same for both CentOS and Ubuntu, the actions taken to do the initial package installation differ slightly. Rather than going through the process of showing the commands to install and configure the agent, there is a script for each of the host operating systems in the `/monitoring_docker/chapter04/` folder. To view the scripts, run the following command from your terminal:

```
cat /monitoring_docker/chapter04/install-agent-centos.sh
cat /monitoring_docker/chapter04/install-agent-ubuntu.sh
```

Now, you have taken a look at the scripts its time to run them, to do this type one of the following commands. If you are running CentOS, run this command:

```
bash /monitoring_docker/chapter04/install-agent-centos.sh
```

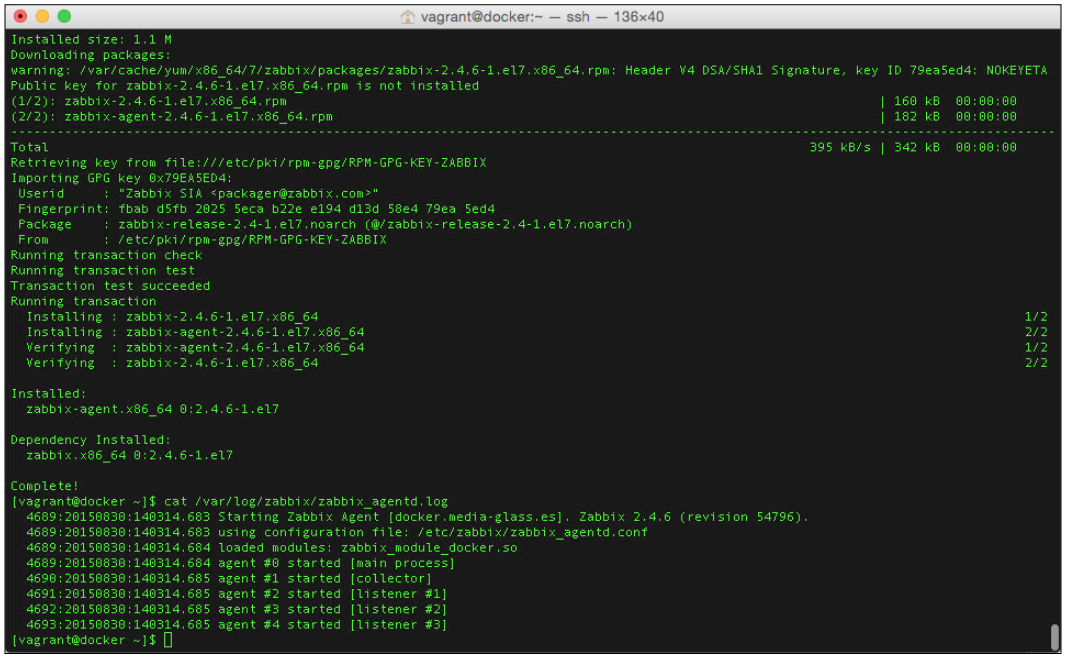
For Ubuntu, run the following command:

```
bash /monitoring_docker/chapter04/install-agent-ubuntu.sh
```

To verify that everything ran as expected, check the Zabbix agent log file by running the following command:

```
cat /var/log/zabbix/zabbix_agentd.log
```

You should see that the end of the file confirms that the agent has started and that the `zabbix_module_docker.so` module has been loaded:



Before we move onto the Zabbix web interface, let's launch a few containers using the `docker-compose` file from *Chapter 2*, *Using the Built-in Tools*:

```
[vagrant@docker ~]$ cd /monitoring_docker/chapter02/02-multiple/
[vagrant@docker 02-multiple]$ docker-compose up -d
[vagrant@docker 02-multiple]$ docker-compose scale web=3
[vagrant@docker 02-multiple]$ docker-compose ps
```

We should now have three web server containers running and a running Zabbix agent on the host.

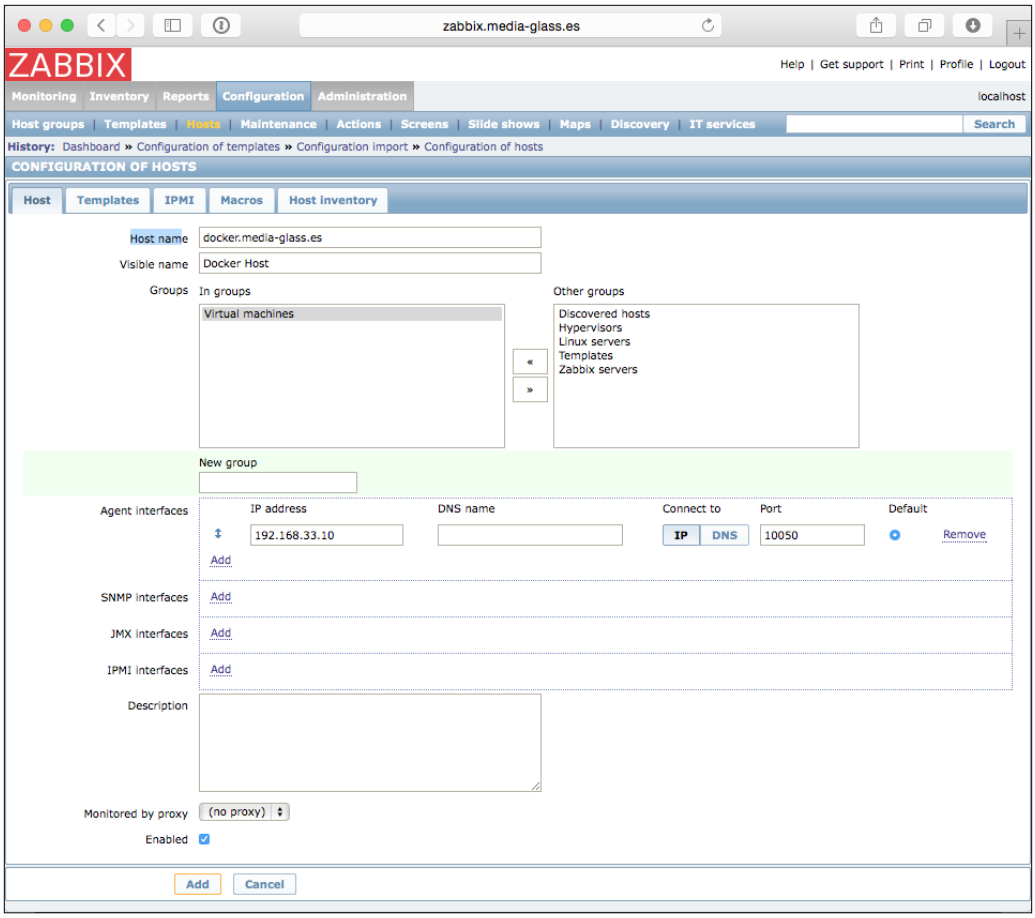
The Zabbix web interface

Once you have Zabbix installed you can open the Zabbix web interface by going to `http://zabbix.media-glass.es/` in your browser, this link will only work when you have the Zabbix vagrant box up and running, if you don't have it running the page will time out. You should be presented with a login screen. Enter the default username and password here, which is `Admin` and `zabbix` (note that the username has a capital *A*), to login.

Once logged in, you will need to add the host templates. These are preconfigured environment settings and will add some context around the statistics that the Zabbix agent is sending to the server, along with the auto-discovery of containers.

To add the templates, go to the **Configuration** tab in the top menu and select **Template**; this will bring up a list of all the templates that are currently installed. Click on the **Import** button in the header and upload a copy of the two template files you can find in the `~/Documents/Projects/monitoring-docker/chapter04/template` folder on your main machine; there is no need to change the rules when uploading the templates.

Once both templates have been successfully imported, it is time to add our Docker host. Again, go to the **Configuration** tab, but this time select **Hosts**. Here, you need to click on **Create host**. Then, enter the following information in the **Host** tab:



Here are the details of the preceding information:

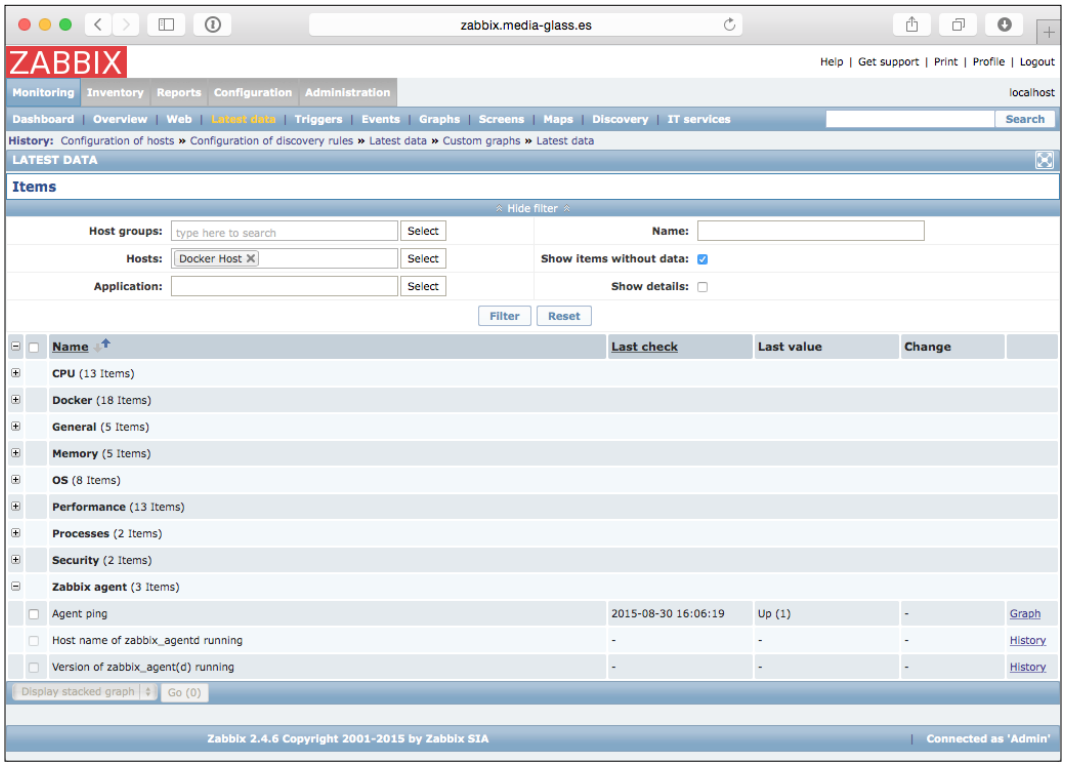
- **Host name**: This is the host name of our Docker host
- **Visible name**: Here, the name server will appear as in Zabbix
- **Groups**: Which group within Zabbix the server you would like the Docker host to be part of
- **Agent Interfaces**: This is the IP address or the DNS name of our Docker host
- **Enabled**: This should be ticked

Before clicking on **Add**, you should click on the **Templates** tab and link the following two templates to the host:

- **Template App Docker**
- **Template OS Linux**

Here is the screenshot of the host:

Once you have added the two templates, click on **Add** to configure and enable the host. To verify that the host has been added correctly, you should go to the **Monitoring** tab and then **Latest data**. From here, click on **Show filter** and enter the host machine in the **Hosts** box. You should then start to see items appearing:



Don't worry if you don't see the **Docker** section immediately, by default, Zabbix will attempt to auto-discover new containers every five minutes.

Docker metrics

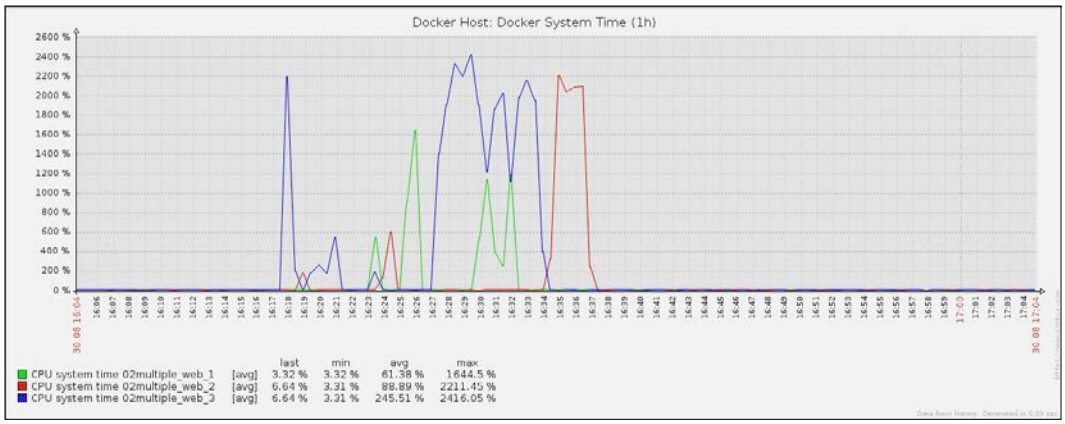
For each container, Zabbix discovers the following metrics that will be recorded:

- Container (your Containers name) is running
- CPU system time
- CPU user time
- Used cache memory
- Used RSS memory
- Used swap

Apart from "Used swap", these are the same metrics recorded by cAdvisor.

Create custom graphs

You can access a time-based graph for any of the metrics collected by Zabbix; you can also create your own custom graphs. In the following graph, I have created a graph that plots all the CPU System stats from the three web containers we launched earlier in the chapter:

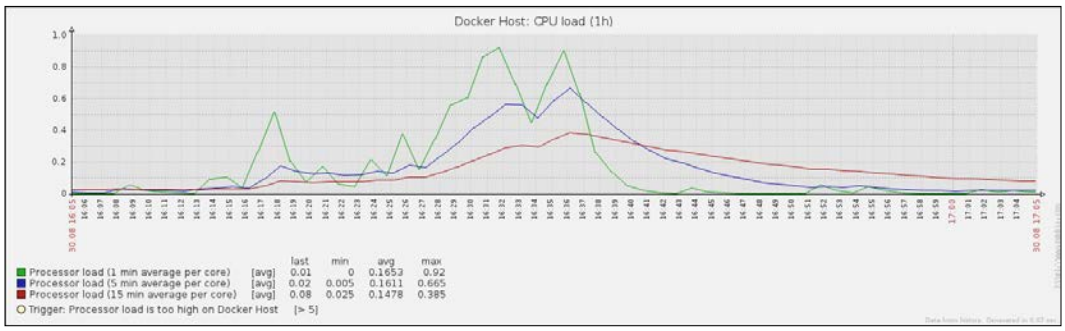


As you can see, I performed a few tests using ApacheBench to make the graph a little more interesting.

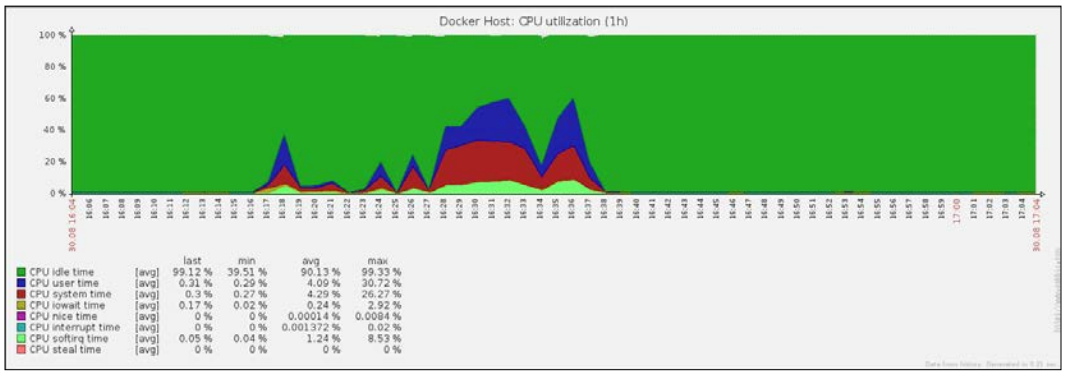
For more information on how to create custom graphs, see the graphs section of the documentation site at `https://www.zabbix.com/documentation/2.4/manual/config/visualisation/graphs`.

Compare containers to your host machine

As we added the Linux OS template and the Docker template to the host and we are also recording quite a lot of information about the system, here we can tell the effect the testing with ApacheBench had on the overall processor load:

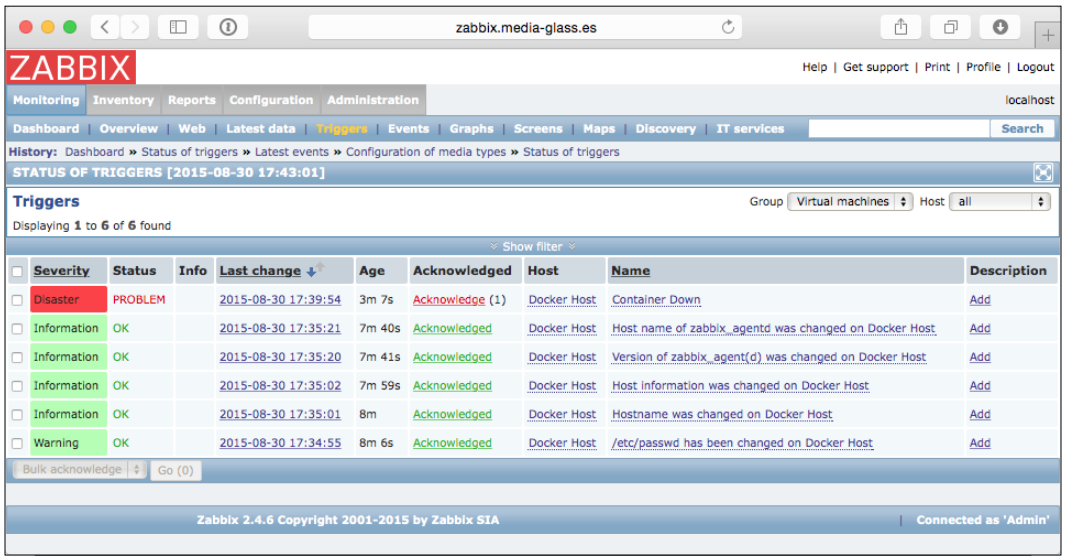


We can drill down further to get information on the overall utilization:

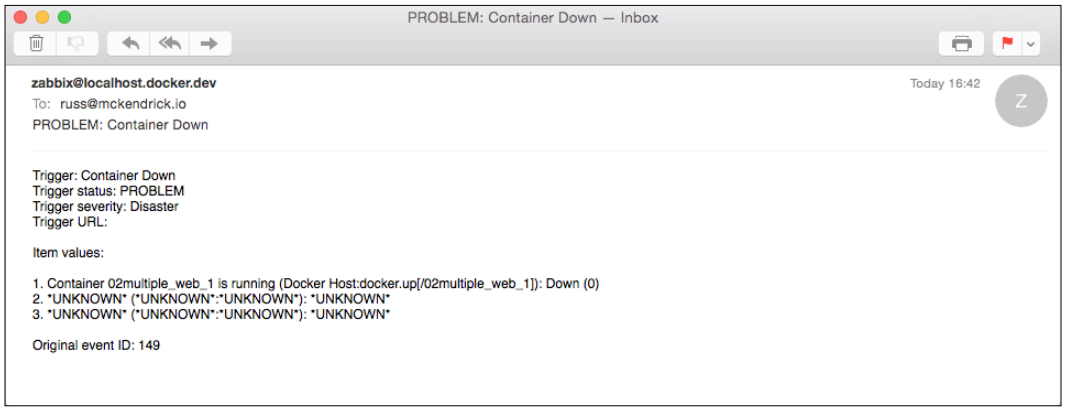


Triggers

Another feature of Zabbix is triggers: you can define actions to happen when a metric meets a certain set of criteria. In the following example, Zabbix has been configured with a trigger called **Container Down**; this changes the status of the monitored item to **Problem** with a severity of **Disaster**:



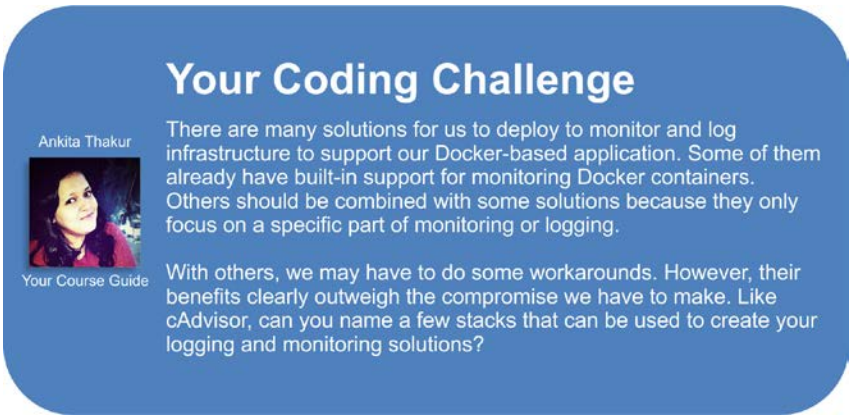
This change in status then triggers an e-mail to inform that, for some reason the container is no longer up and running:



This could have also triggered other tasks, such as running a custom script, sending an instant message via Jabber, or even triggering a third-party service such as PagerDuty (`https://www.pagerduty.com`) or Slack (`https://slack.com`).

For more information on Triggers, Events, and Notifications, see the following sections of the documentation:

- `https://www.zabbix.com/documentation/2.4/manual/config/triggers`
- `https://www.zabbix.com/documentation/2.4/manual/config/events`
- `https://www.zabbix.com/documentation/2.4/manual/config/notifications`



Summary of Module 3 Chapter 4

Ankita Thakur

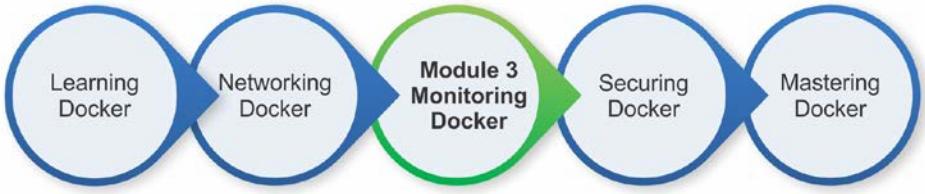
Your Course Guide

So, how does this traditional approach to monitoring fit into a container's lifecycle? Going back to the Pets versus Cattle analogy, at first glance, Zabbix seems to be geared more towards Pets: its feature set is best suited to monitoring services that are static over a long period of time. This means that the same approach to monitoring a pet can also be applied to long-running processes running within your containers.

Zabbix is also the perfect option for monitoring mixed environments. Maybe you have several database servers that are not running as containers, but you have several hosts running Docker, and have equipment such as switches and SANs that you need to monitor. Zabbix can provide you with a single pane of glass showing you metrics for all your environments, along with being able to alert you to problems.

So far, we have looked at using APIs and metrics provided by Docker and LXC, but what about other metrics can we use? In the next chapter, we will look at a tool that hooks straight into the host machine's kernel to gather information on your containers.

Your Progress through the Course So Far



5
Querying with Sysdig

The previous tools we have looked at have all relied on making API calls to Docker or reading metrics from LXC. Sysdig works differently by hooking itself into the hosts machine's kernel while this approach does go against Docker's philosophy of each service being run in its own isolated container, the information you can get by running Sysdig only for a few minutes far outweighs any arguments about not using it.

In this chapter, we will look at the following topics:

- How to install Sysdig and Csysdig on the host machine
- Basic usage and how to query your containers in real time
- How to capture logs so they can be queried later

What is Sysdig?

Before we start to get into Sysdig, let's first understand what it is. When I first heard about the tool, I thought to myself that it sounded too good to be true; the website describes the tool as follows:

> *"Sysdig is open source, system-level exploration: capture system state and activity from a running Linux instance, then save, filter and analyze. Sysdig is scriptable in Lua and includes a command line interface and a powerful interactive UI, csysdig, that runs in your terminal. Think of sysdig as strace + tcpdump + htop + iftop + lsof + awesome sauce. With state of the art container visibility on top."*

This is quite a claim as all the tools that it is claiming to be as powerful were all in a set of goto commands to run when looking into problems, so I was a little skeptical at first.

As any one who has had to try and track down a haywire process of try and track down an issue that isn't being very verbose in its error logs on a Linux server will know that using tools such as strace, lsof, and tcpdump can get complicated very quickly and it normally involves capturing a whole lot of data and then using a combination of several tools to slowly, and manually, trace the problem by reducing the amount of data you captured.

Imagine my delight when Sysdig's claims turned out to be true. It made me wish I had the tool back when I was a front line engineer; it would have made my life a lot easier.

Sysdig comes in two different flavors, first is the Open Source version available at http://www.sysdig.org/; this comes with an ncurses interface so that you can easily access and query data from a terminal-based GUI.

Wikipedia describes **ncurses** (new curses) as a programming library that provides an API that allows the programmer to write text-based user interfaces in a terminal-independent manner. It is a toolkit for developing "GUI-like" application software that runs under a terminal emulator. It also optimizes screen changes in order to reduce the latency experienced when using remote shells.

There is also a commercial service that allows you to stream your Sysdig to their externally hosted service; this version has a web-based interface for viewing and querying your data.

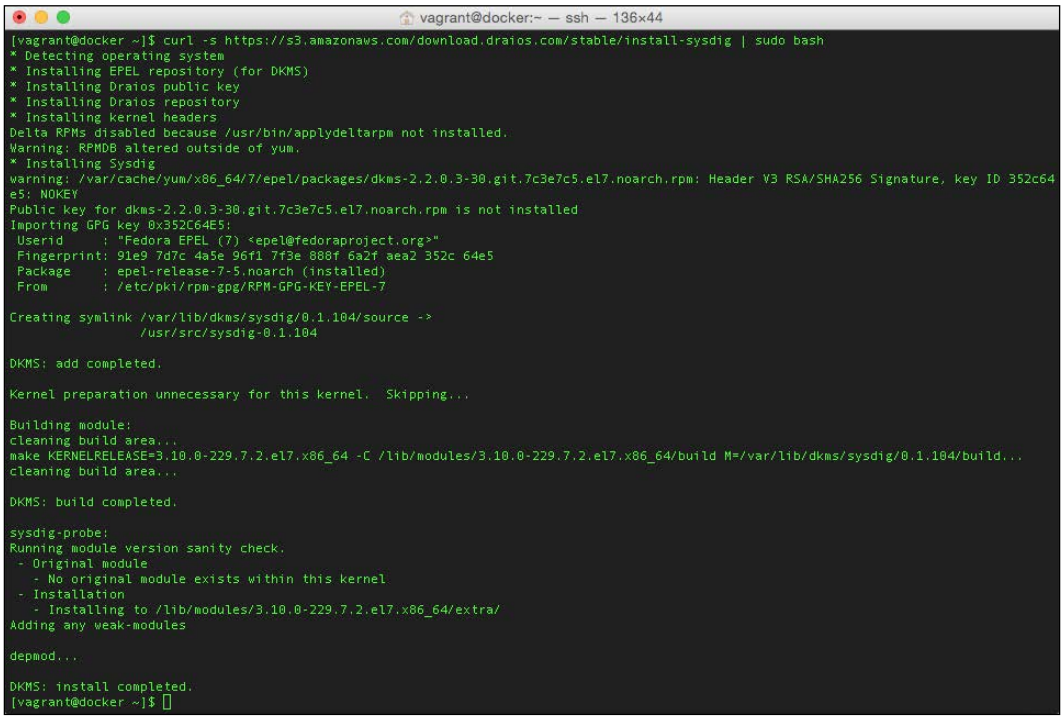
In this chapter, we will be concentrating on the open source version.

Installing Sysdig

Considering how powerful Sysdig is, it has one of the most straightforward installation and configuration processes I have come across. To install Sysdig on either a CentOS or Ubuntu server, type the following command:

```
curl -s https://s3.amazonaws.com/download.draios.com/stable/install-
sysdig | sudo bash
```

After running the preceding command, you will get the following output:



That's it, you are ready to go. There is nothing more to configure or do. There is a manual installation process and also a way of installing the tool using containers to build the necessary kernel modules; for more details, see the installation guide as follows:

`http://www.sysdig.org/wiki/how-to-install-sysdig-for-linux/`

Using Sysdig

Before we look at how to use Sysdig, let's launch a few containers using `docker-compose` by running the following command:

```
cd /monitoring_docker/chapter05/wordpress/
docker-compose up -d
```

This will launch a WordPress installation running a database and two web server containers that are load balanced using an HAProxy container. You will be able to view the WordPress installation at `http://docker.media-glass.es/` once the containers have launched. You will need to enter some details to create the admin user before the site is visible; follow the on-screen prompts to complete these steps.

The basics

At its core, Sysdig is a tool for producing a stream of data; you can view the stream by typing `sudo sysdig` (to quit, press *Ctrl+c*).

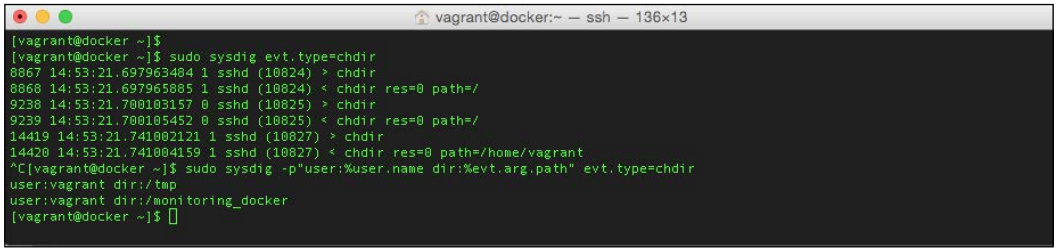
There is a lot information there so let's start to filter the stream down and run the following command:

```
sudo sysdigevt.type=chdir
```

This will display only events in which a user changes directory; to see it in action, open a second terminal and you will see that when you log in, you see some activity in the first terminal. As you can see, it looks a lot like a traditional log file; we can format output to give information such as the username, by running the following command:

```
sudo sysdig -p"user:%user.name dir:%evt.arg.path" evt.type=chdir
```

Then, in your second terminal, change the directory a few times:



As you can see, this is a lot easier to read than the original unformatted output. Press *Ctrl* + *c* to stop filtering.

Capturing data

In the previous section, we looked at filtering data in real time; it is also possible to stream Sysdig data to a file so that you can query the data at a later time. Exit from your second terminal and run the following command on your first one:

```
sudosysdig -w ~/monitoring-docker.scap
```

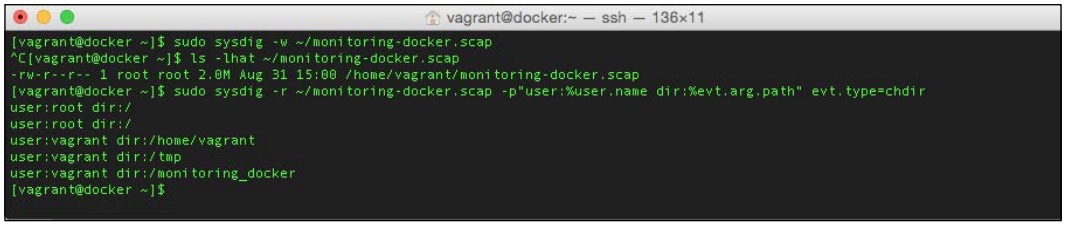
While the command is running on the first terminal, log in to the host on the second one and change the directory a few times. Also, while we are recording, click around the WordPress site we started at the beginning of this section, the URL is `http://docker.media-glass.es/`. Once you have done that, stop the recording by pressing *Crtl* + *c*; you should have now dropped back to a prompt. You can check the size of the file created by Sysdig by running the following:

```
ls -lha ~/monitoring-docker.scap
```

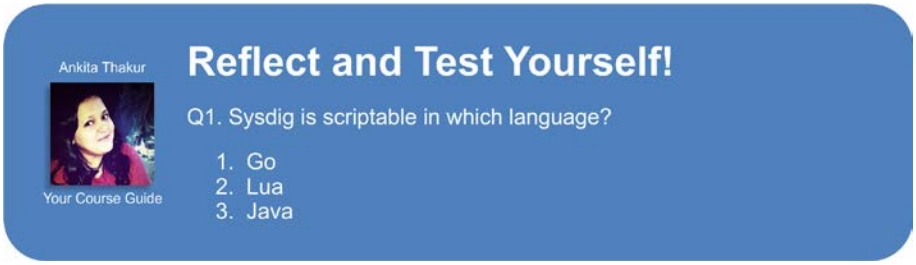
Now, we can use the data that we have captured to apply the same filter as we did when looking at the real-time stream:

```
sudosysdig -r ~/monitoring-docker.scap -p"user:%user.name dir:%evt.arg.
path" evt.type=chdir
```

By running the preceding command, you will get the following output:



Notice how we get similar results to when we were viewing the data in real time.



Containers

One of the things that was recorded in `~/monitoring-docker.scap` was details on the system state; this includes information on the containers we launched at the start of the chapter. Let's use this file to get some stats on the containers. To list the containers that were active during the time, we captured the data file run:

```
sudo sysdig -r ~/monitoring-docker.scap -c lscontainers
```

To see which of the containers utilized the CPU most of the time, we were clicking around the WordPress site run:

```
sudo sysdig -r ~/monitoring-docker.scap -c topcontainers_cpu
```

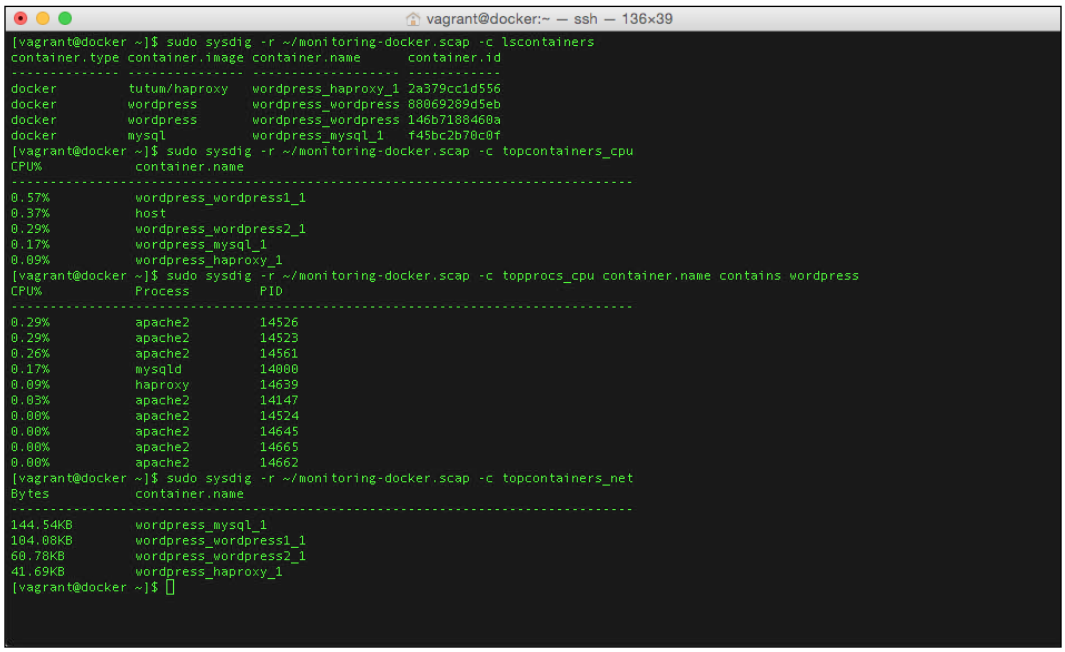
To have a look at the top processes in each of the containers that have "wordpress" in their names (which is all of them in our case), run the following command:

```
sudo sysdig -r ~/monitoring-docker.scap -c topprocs_cpu container.name
contains wordpress
```

Finally, which of our containers transferred the most amount of data?:

```
sudosysdig -r ~/monitoring-docker.scap -c topcontainers_net
```

By running the preceding command, you will get the following output:



As you can see, we have extracted quite a bit of information on our containers from the data we captured. Also, using the file, you can remove the `-r ~/monitoring-docker.scap` part of the command to view the container metrics in real time.

It's also worth pointing out that there are binaries for Sysdig that work on both OS X and Windows; while these do not capture any data, they can be used to read data that you have recorded on your Linux host.

Further reading

From the few basic exercises covered in this section, you should start to get an idea of just how powerful Sysdig can be. There are more examples on the Sysdig website at `http://www.sysdig.org/wiki/sysdig-examples/`. Also, I recommend you to read the blog post at `https://sysdig.com/fishing-for-hackers/`; it was my first exposure to Sysdig and it really demonstrates its usefulness.

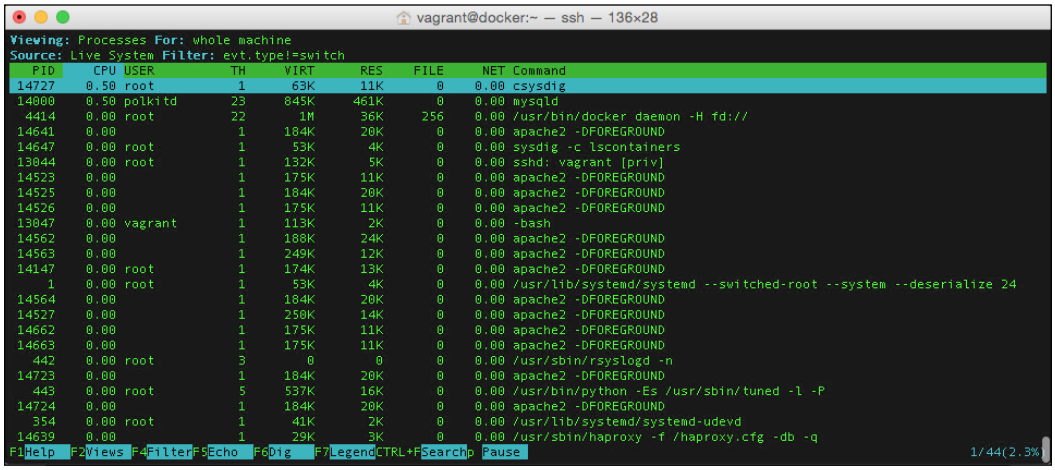
Using Csysdig

As easy as it is to view data captured by Sysdig using the command line and manually filtering the results, it can get more complicated as you start to string more and more commands together. To help make the data captured by Sysdig as accessible as possible, Sysdig ships with a GUI called **Csysdig**.

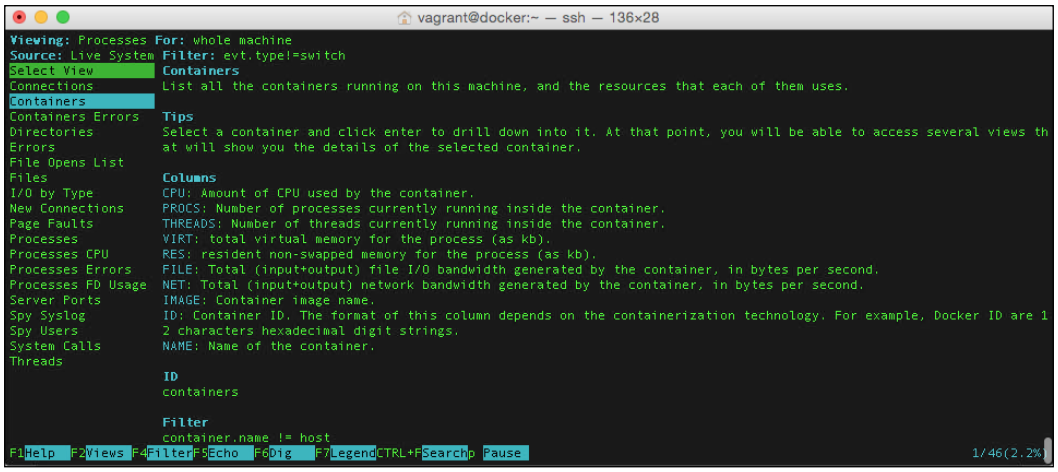
Launching the Csysdig is done with a single command:

```
sudo csysdig
```

Once the process has launched, it should instantly look familiar to anyone who has used top or cAdvisor (minus the graphs); its default view will show you real-time information on the processes that are running:



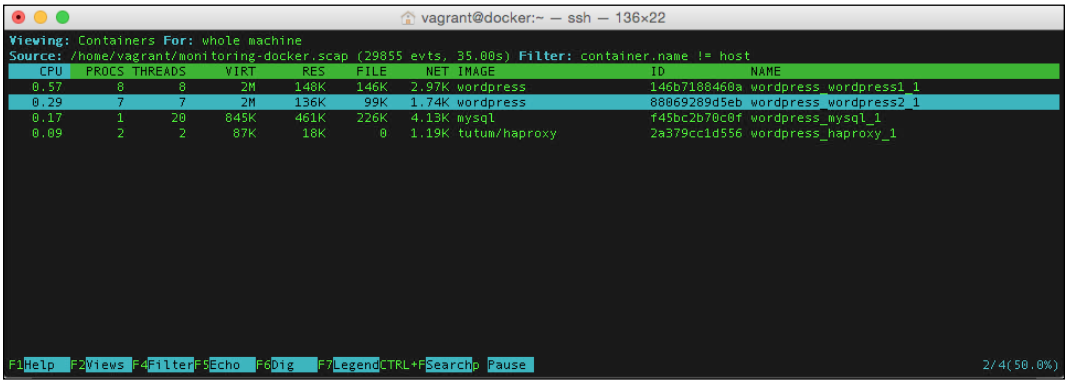
To change this view, known as the **Processes** view, press *F2* to open the **Views** menu; from here, you can use the up and down arrows on your keyboard to select a view. As you may have already guessed, we would like to see the **Containers** view:



However, before we drill down into our containers, let's quit Csysdig by pressing *q* and load up the file we created in the previous section. To do this, type the following command:

```
sudo csysdig -r ~/monitoring-docker.scap
```

Once Csysdig loads, you will notice that **Source** has changed from **Live System** to the file path of our data file. From here, press *F2* and use the up arrow to select containers and then hit *Enter*. From here, you can use the up and down arrows to select one of the two web servers, these would be either `wordpress_wordpress1_1` or `wordpress_wordpress2_1` as shown in the following screen:

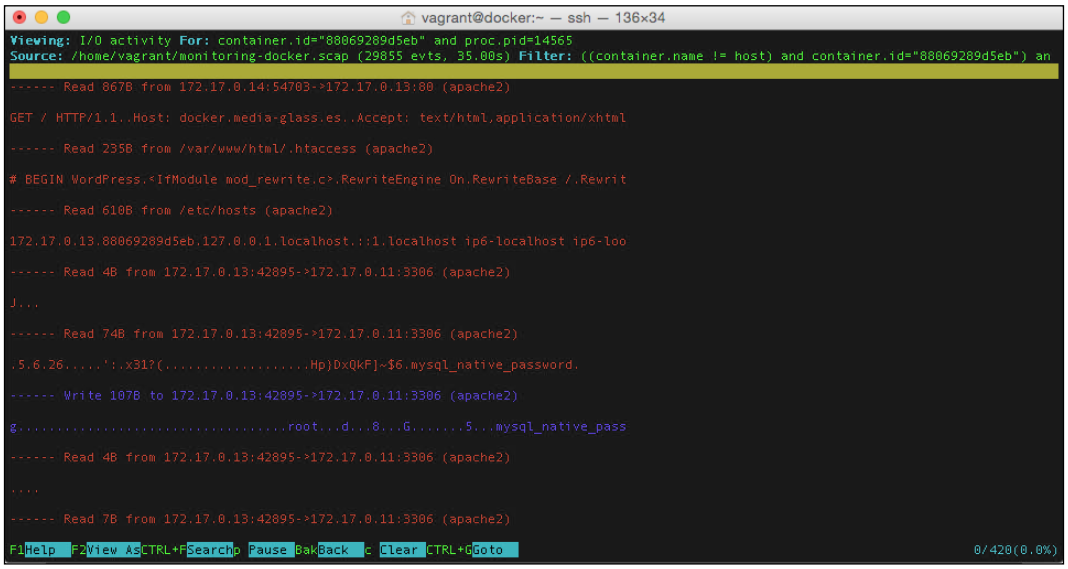


The remaining part of this chapter assumes that you have Csysdig open in-front of you, it will talk you through how to navigate around the tool. Please feel free to explore yourself as well.

Once you have selected a server, hit *Enter* and you will be presented with a list of processes that the container was running. Again, you can use the arrow keys to select a process to drill down further into.

I suggested looking at one of the Apache processes that has a value listed in the **File** column. This time, rather than pressing *Enter* to select the process, let's "Echo" what the process was up to at the time we captured the data; with the process selected, press *F5*.

You can use the up and down arrows to scroll through the output:



To better format the data, press *F2* and select **Printable ASCII**. As you can see from the preceding screenshot, this Apache process performed the following tasks:

- Accepted an incoming connection
- Accessed the `.htaccess` file
- Read the `mod_rewrite` rules
- Got information from the hosts file
- Made a connection to the MySQL container
- Sent the MySQL password

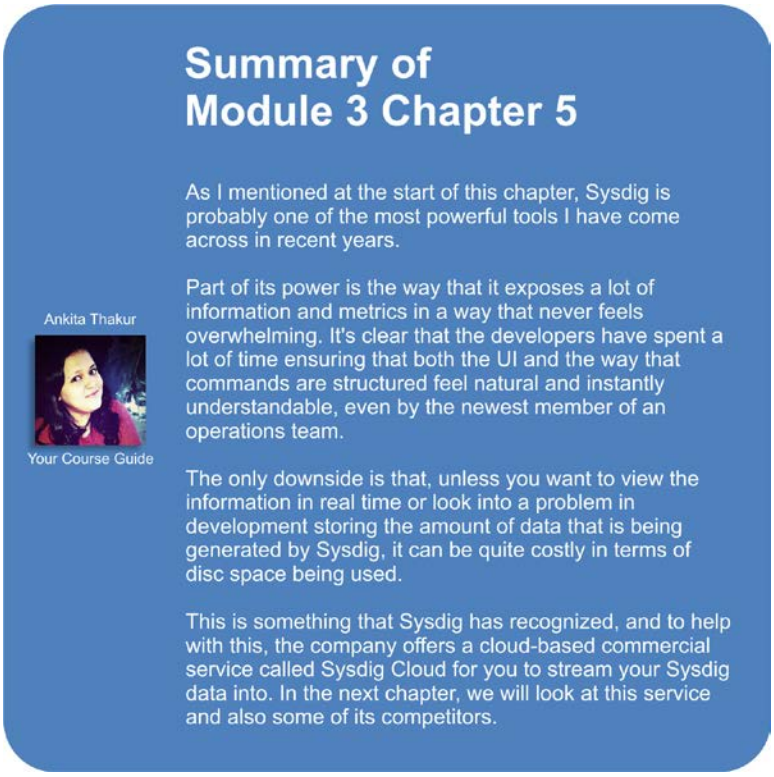
By scrolling through the remainder of the data in the "Echo" results for the process, you should be able to easily follow the interactions with the database all the way through to the page being sent to the browser.

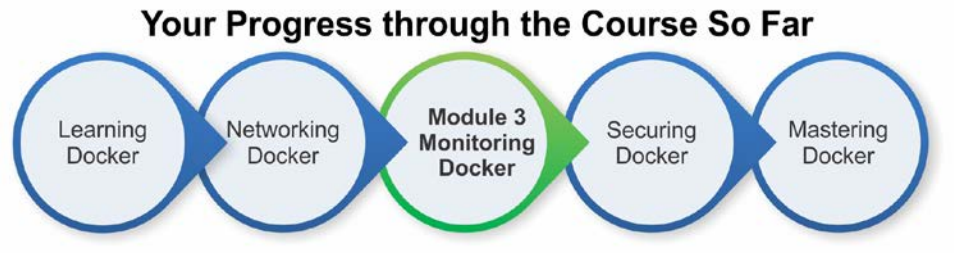
To leave the "Echo" screen, press *Backspace*; this will always take you a level back.

If you want a more detailed breakdown on what the process was doing, then press *F6* to enter the **Dig** view; this will list the files that the process was accessing at the time, along with the network interaction and how it is accessing the RAM.

To view a full list of commands and for more help, you can press *F1* at anytime. Also, to get a breakdown on any columns that are on screen, press *F7*.


Ankita Thakur
Your Course Guide
Your Coding Challenge
Let's test what you've learned so far:
• What is Sysdig?
• What is Csysdig?


Summary of
Module 3 Chapter 5
Ankita Thakur
Your Course Guide
As I mentioned at the start of this chapter, Sysdig is probably one of the most powerful tools I have come across in recent years.
Part of its power is the way that it exposes a lot of information and metrics in a way that never feels overwhelming. It's clear that the developers have spent a lot of time ensuring that both the UI and the way that commands are structured feel natural and instantly understandable, even by the newest member of an operations team.
The only downside is that, unless you want to view the information in real time or look into a problem in development storing the amount of data that is being generated by Sysdig, it can be quite costly in terms of disc space being used.
This is something that Sysdig has recognized, and to help with this, the company offers a cloud-based commercial service called Sysdig Cloud for you to stream your Sysdig data into. In the next chapter, we will look at this service and also some of its competitors.


Your Progress through the Course So Far
Learning Docker
Networking Docker
Module 3 Monitoring Docker
Securing Docker
Mastering Docker

6
Exploring Third Party Options

So far, we have been looking at the tools and services you host yourself. Along with these self-hosted tools, a large amount of cloud-based software has developed around Docker as a service ecosystem. In this chapter, we will look at the following topics:

- Why use a SaaS service over self-hosted or real-time metrics?
- What services are available and what do they offer?
- Installation of agents for Sysdig Cloud, Datadog, and New Relic on the host machines
- Configuration of the agents to ship metrics

A word about externally hosted services

So far, to work through the examples in this module, we have used locally hosted virtual servers that are launched using vagrant. During this chapter, we are going to use services that need to be able to communicate with your host machine, so rather than trying to do this using your local machine, its about time you took your host machine into the cloud.

As we are going to start and stop the remote hosts while we look at the services, it pays to use a public cloud, as we only get charged for what we use.

There are several public cloud services that you can use to evaluate the tools covered in this chapter, which one you choose to use is up to you, you could use:

- Digital Ocean: `https://www.digitalocean.com/`
- Amazon Web Services: `https://aws.amazon.com/`
- Microsoft Azure: `https://azure.microsoft.com/`
- VMware vCloud Air: `http://vcloud.vmware.com/`

Or use your own preferred provider, the only pre-requisite is that your server is publically accessible.

This chapter assumes that you are capable of launching either a CentOS 7 or Ubuntu 14.04 cloud instance and you understand that you will likely incur charges while the cloud instance is up and running.

Deploying Docker in the cloud

Once you have launched your cloud instance, you can bootstrap Docker in the same way that you installed using vagrant. In the `chapter 6` folder of the Git repository, there are two separate scripts to download and install the Docker engine and compose it on your cloud instance.

To install Docker, ensure that your cloud instance is updated by running:

```
sudo yum update
```

For the CentOS instance of your Ubuntu, run the following command:

```
sudo apt-get update
```

Once updated, run the following command to install the software. Due to the differences in the way different cloud environments are configured, it is best to switch over to the root user to run the remainder of the commands, to do this, run:

```
sudo su -
```

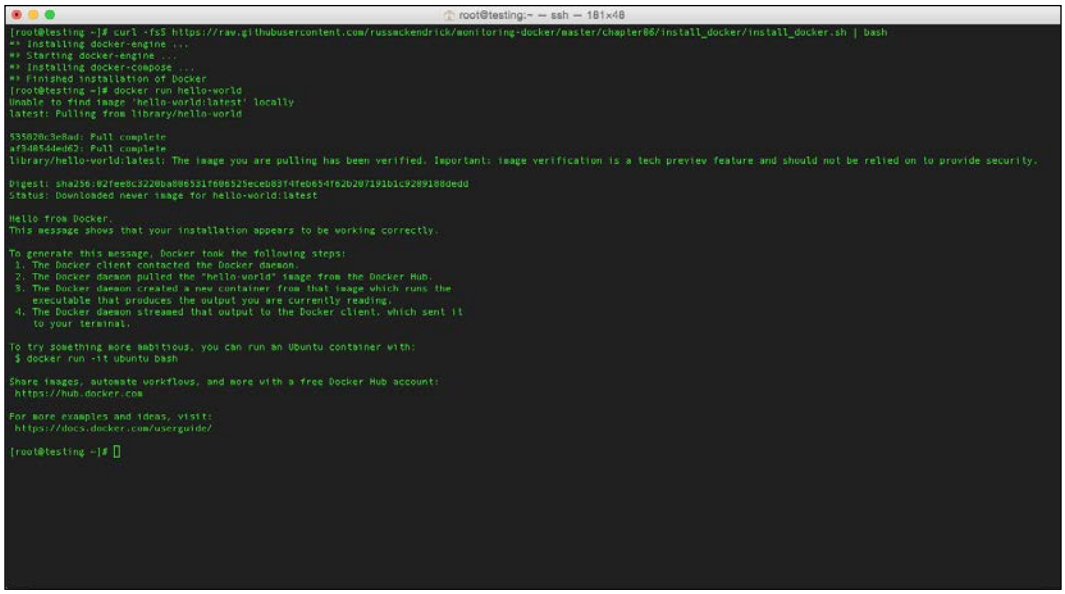
Now you will be able to run the install script using the following command:

```
curl -fsS https://raw.githubusercontent.com/russmckendrick/monitoring-
docker/master/chapter06/install_docker/install_docker.sh | bash
```

To check that everything works as expected, run the following command:

```
docker run hello-world
```

You should see something similar to the terminal output, as shown in the following screenshot:



We can start to look at the SaaS services once you have Docker up and running.

Why use a SaaS service?

You may have noticed while working with the examples in the previous chapters that the tools we have used can potentially use many resources if we needed to start collecting more metrics, especially if the applications we want to monitor are in production.

To help shift this load from both storage and CPU, a number of cloud-based SaaS options have started offering support to record metrics for your containers. Many of these services were already offering services to monitor servers, so adding support for containers seemed a natural progression for them.

These typically require you to install an agent on your host machine, once installed, the agent will sit in the background and report to the services, normally cloud-based and API services.

A few of the services allow you to deploy the agents as Docker containers. They offer containerized agents so that the service can run on stripped down operating systems, such as:

- CoreOS: `https://coreos.com/`
- RancherOS: `http://rancher.com/rancher-os/`
- Atomic: `http://www.projectatomic.io/`
- Ubuntu Snappy Core: `https://developer.ubuntu.com/en/snappy/`

These operating systems differ from traditional ones, as you cannot install services on them directly; their only purpose is to run a service, such as Docker, so that you can launch the services or applications you need to be run as containers.

As we are running full operating systems as our host systems, we do not need this option and will be deploying the agents directly to the hosts.

The SaaS options that we are going to look at in this chapter are as follows:

- Sysdig Cloud: `https://sysdig.com/product/`
- Datadog: `https://www.datadoghq.com/`
- New Relic: `http://newrelic.com`

They all offer free trials and two of them offer free cut-down versions of the main service. On the face of it, they might all appear to offer similar services; however, when you start to use them, you will immediately notice that they are in fact all very different from each other.

Sysdig Cloud

In the previous chapter, we had a look at the open source version of Sysdig. We saw that there is a great ncurses interface called cSysdig and it allows us to navigate through all the data that Sysdig is collecting about our host.

The sheer amount of metrics and data collected by Sysdig means that you have to try to stay on top of it either by shipping your files off the server, maybe to Amazon Simple Storage Service (S3), or to some local shared storage. In addition, you can query the data in the command line on the host itself or on your local machine using an installation of the command-line tools.

This is where Sysdig Cloud comes into play; it offers a web-based interface to the metrics that Sysdig captures along with the options to ship the Sysdig captures off your host machine either to Sysdig's own storage or to your S3 bucket.

Sysdig cloud offers the following functionality:

- ContainerVision™
- Real-Time Dashboard
- Historical Replay
- Dynamic Topology
- Alerting

As well as, the option to trigger a capture on any of your hosts and at any time.

Sysdig describes ContainerVision as:

> *"Sysdig Cloud's patent-pending core technology, ContainerVision, is the only monitoring technology on the market designed specifically to respect the unique characteristics of containers. ContainerVision offers you deep and comprehensive visibility into all aspects of your containerized environment - applications, infrastructures, servers, and networks - all without the need to pollute your containers with any extra instrumentation. In other words, ContainerVision gives you 100% visibility into the activity inside your containers, from the outside."*

Before we delve into Sysdig Cloud any further, I should point out that this is a commercial server and at the time of writing, it costs $25 per host per month. There is also a 14-day fully featured trial available. If you wish to work through the agent installation and follow the example in this chapter, you will need an active account that runs either on the 14-day trial or a paid subscription.

- Sign up for a 14-day free trial: `https://sysdig.com/`
- Details on pricing: `https://sysdig.com/pricing/`
- Introduction to the company: `https://sysdig.com/company/`

Installing the agent

The agent installation is similar to installing the open source version; you need to ensure that your cloud host is running an up-to-date kernel and that you are also booted into the kernel.

Some cloud providers keep a tight control on the kernels you can boot into (for example, Digital Ocean), and they do not allow you to manage your kernel on the host itself. Instead, you need to choose the correct version through their control panel.

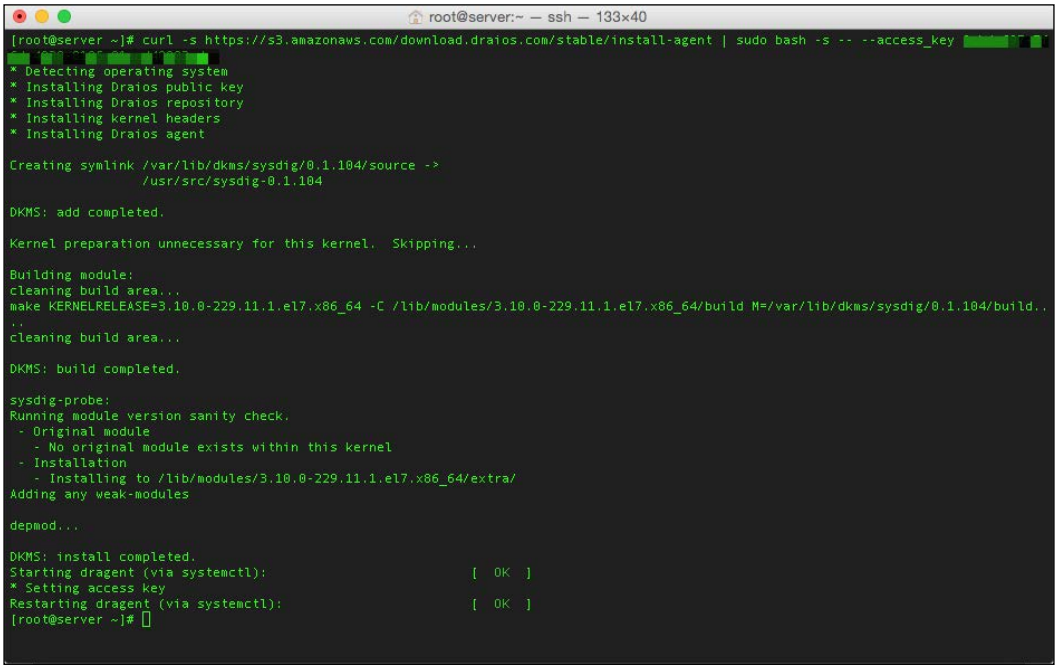
Once you have the correct kernel installed, you should be able to run the following command to install the agent. Ensure that you replace the access key at the end of the command with your own access key, which can be found on your **User Profile** page or on the agent installation pages; you can find these at:

- **User Profile**: `https://app.sysdigcloud.com/#/settings/user`
- **Agent Installation**: `https://app.sysdigcloud.com/#/settings/agentInstallation`

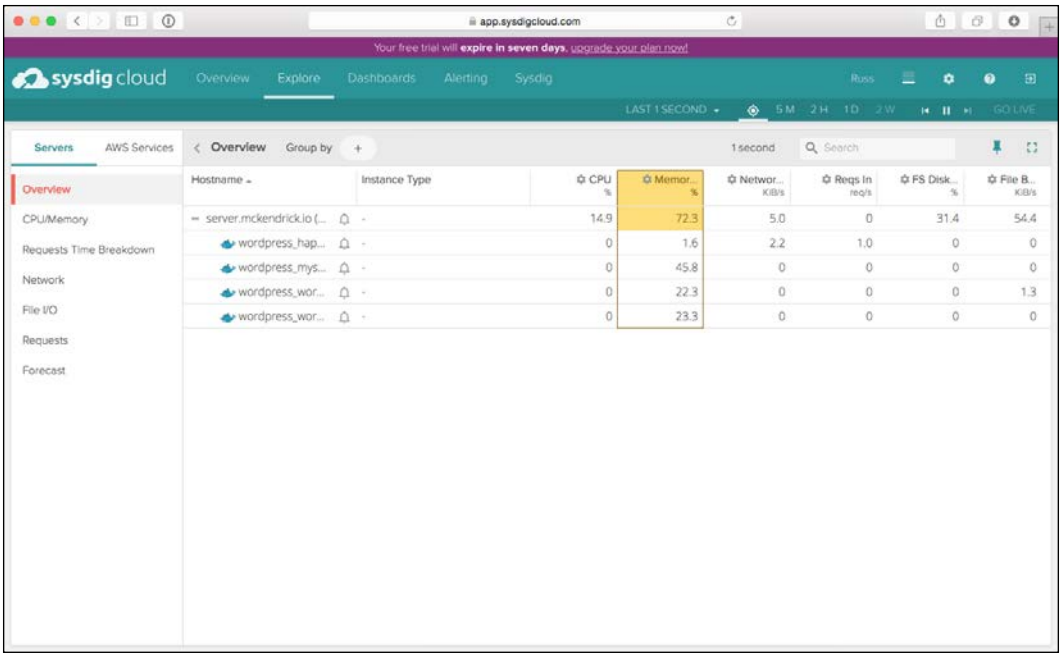
The command to run is:

```
curl -s https://s3.amazonaws.com/download.draios.com/stable/install-agent
| sudo bash -s -- --access_key wn5AYlhjRhgn3shcjW14y3yOT09WsF7d
```

The shell output should look like the following screen:



Once the agent has been installed, it will immediately start to report the data back to Sysdig Cloud. If you click on **Explore**, you will see your host machine and the running containers:



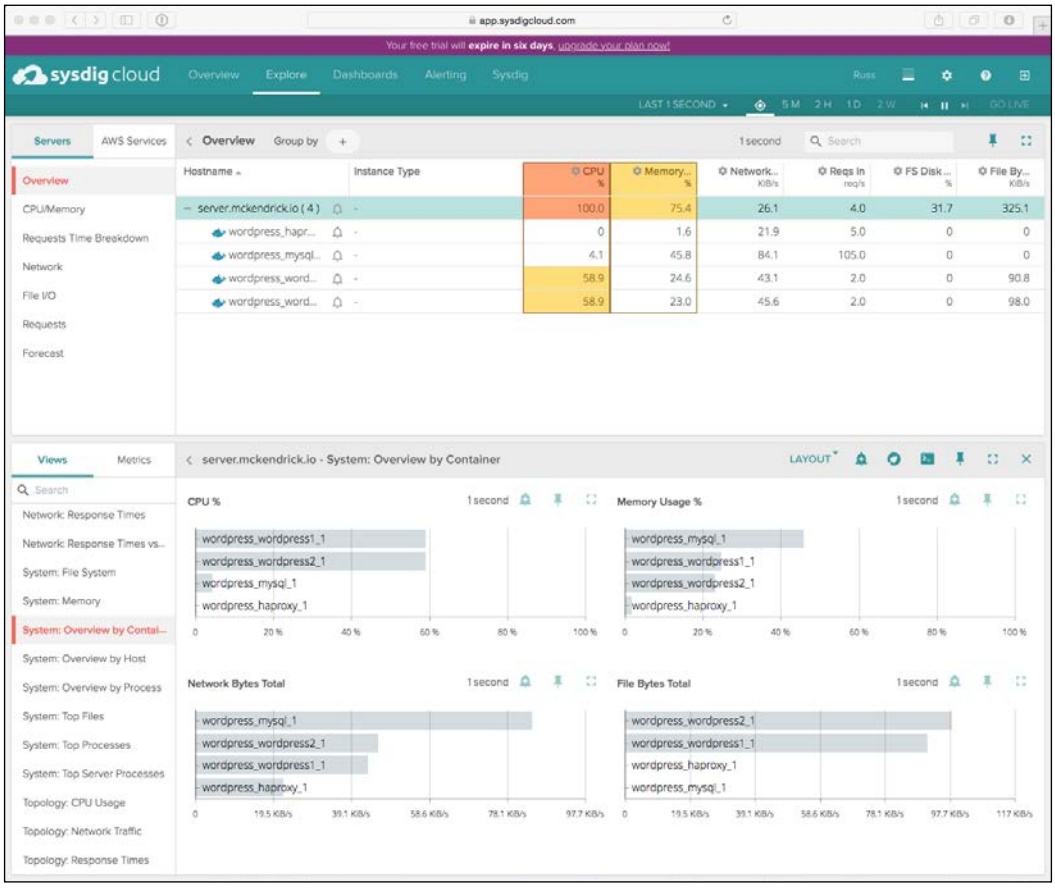
As you can see here, I have my host machine and four containers running a WordPress installation similar to the one we used in the previous chapter. From here, we can start to drill down into our metrics.

To launch the WordPress installation on your cloud-based machine, run the following commands as the root user:

```
sudo su -
mkdir ~/wordpress
curl -L https://raw.githubusercontent.com/russmckendrick/monitoring-docker/master/chapter05/wordpress/docker-compose.yml > ~/wordpress/docker-compose.yml
cd ~/wordpress
docker-compose up -d
```

Exploring your containers

The Sysdig Cloud web interface will feel instantly familiar, as it shares a similar design and overall feeling with cSysdig:

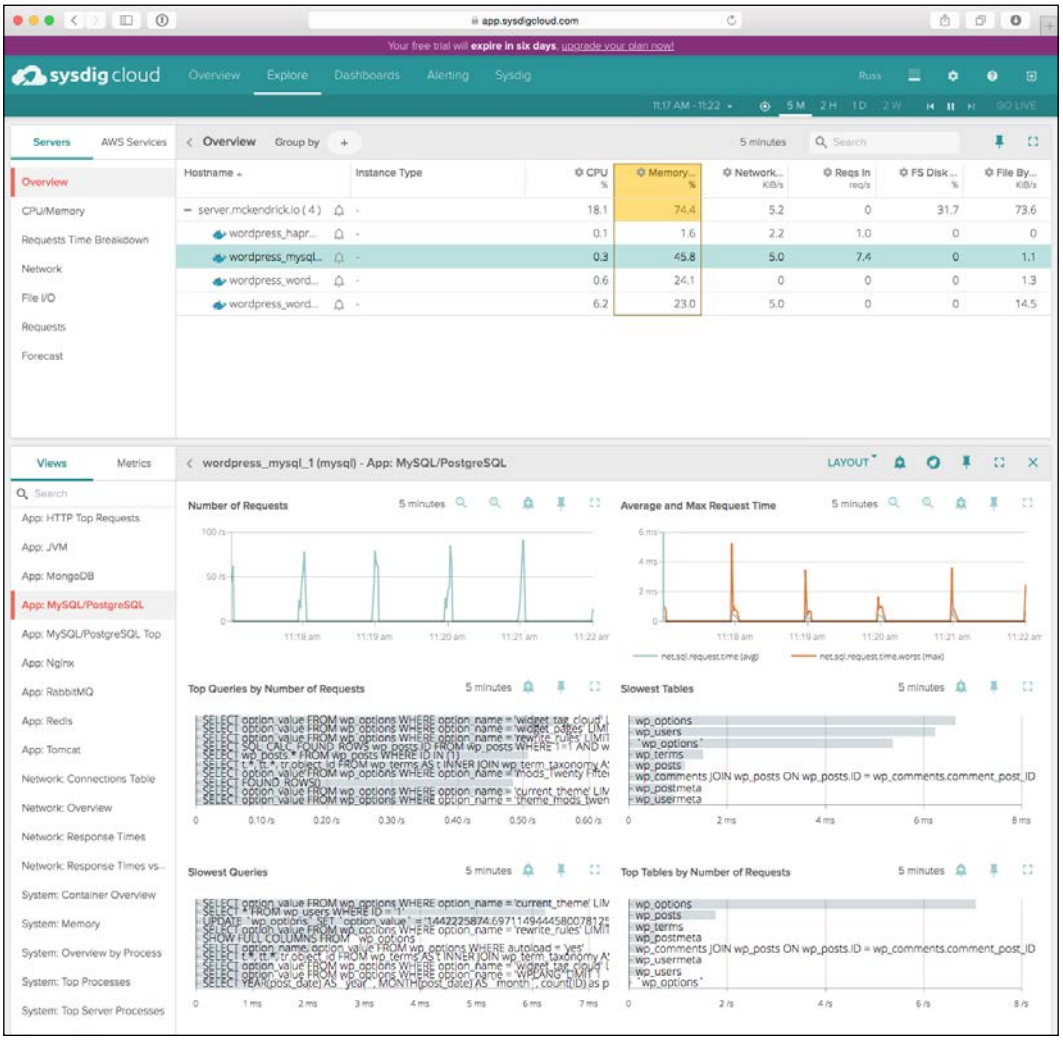


Once you start to drill down, you can see that a bottom pane opens up and this is where you can view the statistics. One of the things I liked about Sysdig Cloud is that it opens up a wealth of metrics and there should be very little that you need to configure from here.

For example, if you want to know what processes have been consuming the most CPU time in the last 2 hours, click on **2H** in the secondary menu and then from the **Views** tab in the bottom-left click on **System: Top Processes**; this will give you a table of the processes, ordered by the ones that have used the most time.

To apply this view to a container, click on a container in the top-section and the bottom-section will be instantly updated to reflect the top CPU utilization for just that container; as most containers will only run one or two processes, this may not be that interesting. So, let's have a deep look at the processes themselves. Let's say, we clicked on our database container and we wanted information on what is happening within MySQL.

Sysdig Cloud comes with application overlays, these when selected give you more granular information on the processes within the container. Selecting the **App: MySQL/PostgreSQL** view gives you an insight into what your MySQL processes are currently doing:



Here, you can see that view in the bottom section has instantly updated to give a wealth of information on what has been happening in the last 5 minutes within MySQL.

Sysdig Cloud supports a number of application views, including:

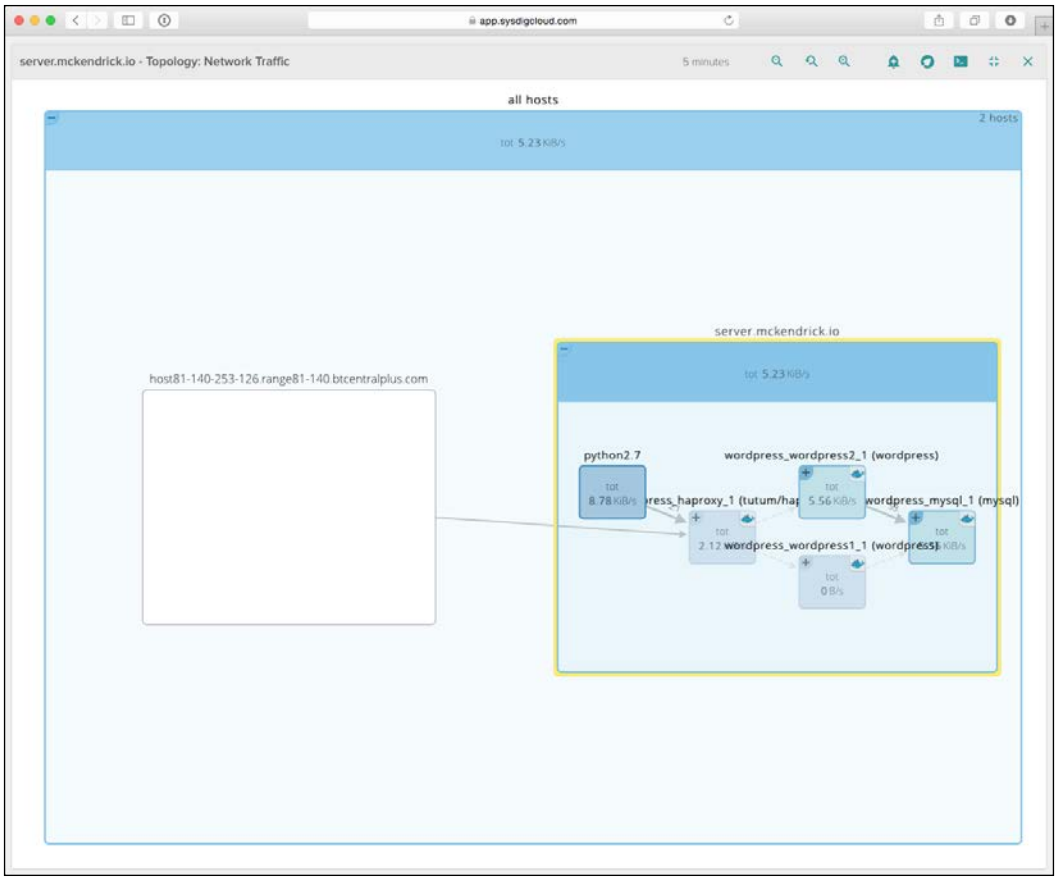
- Apache
- HAProxy
- NGINX
- RabbitMQ
- Redis
- Tomcat

Each one gives you immediate access to metrics, which even the most experienced SysAdmins will find valuable.

You may have noticed that at the top of the second panel there are also a few icons, these allow you to:

- **Add Alert**: Creates an alert based on the view you have open; it lets you tweak the threshold and also choose how you are notified.
- **Sysdig Capture**: Pressing this brings up a dialog, which lets you record a Sysdig session. Once recorded, the session is transferred to Sysdig Cloud or your own S3 bucket. Once the session is available, you download it or explore it within the web interface.
- **SSH Connect**: Gets a remote shell on the server from the Sysdig Cloud web interface; it is useful if you do not have immediate access to your laptop or desktop machine and you want to do some troubleshooting.
- **Pin to dashboard**: Adds the current view to a custom dashboard.

Out these options icons, the "Add Alert" and "Sysdig Capture" options are probably the ones that you will end up using the most. One final view that I found interesting, is the topology one. It gives you a bird's eye view of your host and containers, this is useful too see the interaction between containers and hosts:



Here, you can see me request a page from the WordPress site (it's in the box on the left), this request hits my host machine (the box on the right). Once it's on the host machine, it is routed to the HAProxy container, which then passes the page request to the Wordpress2 container. From here, the Wordpress2 container interacts with the database that is running on the MySQL container.

Summary and further reading

Although Sysdig Cloud is quite a new service, it feels instantly familiar and fully featured as it is built on top of an already established and respected open source technology. If you like, the level of detail you get from the open source version of Sysdig, then Sysdig Cloud is a natural progression for you to start storing your metrics offsite and also to configure alerts. Some good starting points for learning more about Sysdig Cloud are:

- Video Introduction: `https://www.youtube.com/watch?v=p8UVbpw8n24`
- Sysdig Cloud Best Practices: `http://support.sysdigcloud.com/hc/en-us/articles/204872795-Best-Practices`
- Dashboards: `http://support.sysdigcloud.com/hc/en-us/articles/204863385-Dashboards`
- Sysdig blog: `https://sysdig.com/blog/`

If you have launched a cloud instance and are no longer using it, now would be a good time to power the instance down or terminate it altogether. This will ensure that you do not get billed for services that you are not using.

Datadog

Datadog is a full monitoring platform; it supports various servers, platforms, and applications. Wikipedia describes the service as:

> *"Datadog is a SaaS-based monitoring and analytics platform for IT infrastructure, operations and development teams. It brings together data from servers, databases, applications, tools and services to present a unified view of the applications that run at scale in the cloud."*

It uses an agent that is installed on your host machine; this agent sends metrics back to the Datadog service periodically. It also support multiple cloud platforms, such as Amazon Web Services, Microsoft Azure, and OpenStack to name a few.

The aim is to bring all of your servers, applications, and host provider metrics into a single pane of glass; from here, you can create custom dashboards and alerts so that you can be notified of any problem at any level within your infrastructure.

You can sign up for a free trial of the full service at `https://app.datadoghq.com/signup`. You will need at least a trial account to configure the altering, and if your trial has already expired the lite account will do. For more detail on Datadog's pricing structure, please see `https://www.datadoghq.com/pricing/`.

Installing the agent

The agent can be installed either directly on the host machine or as a container. To install directly on the host machine, run the following command and make sure that you use your own unique `DD_API_KEY`:

```
DD_API_KEY=wn5AYlhjRhgn3shcjW14y3yOT09WsF7d bash -c "$(curl -L https://
raw.githubusercontent.com/DataDog/dd-agent/master/packaging/datadog-
agent/source/install_agent.sh)"
```

To run the agent as a container, use the following command and again make sure that you use your own `DD_API_KEY`:

```
sudo docker run -d --name dd-agent -h `hostname` -v /var/run/docker.
sock:/var/run/docker.sock -v /proc/mounts:/host/proc/mounts:ro -v /sys/
fs/cgroup/:/host/sys/fs/cgroup:ro -e API_KEY=wn5AYlhjRhgn3shcjW14y3yOT09W
sF7d datadog/docker-dd-agent
```

Once the agent has been installed, it will call back to Datadog and the host will appear in your account.

If the agent has been installed directly on the host machine then we will need to enable the Docker integration, if you installed the agent using the container then this will have been done for you automatically.

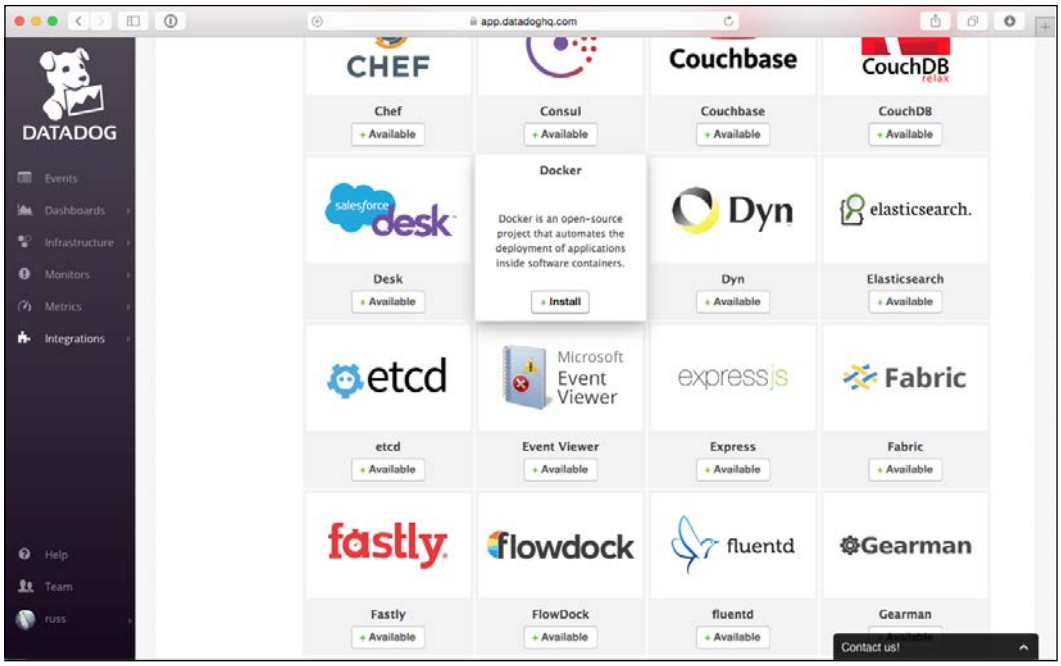
To do this, you first need to allow the Datadog agent access to your Docker installation by adding the `dd-agent` user to the Docker group by running the following command:

```
usermod -a -G docker dd-agent
```

The next step is to create the `docker.yaml` configuration file, luckily the Datadog agent ships with an example configuration file that we can use; copy this in place and then restart the agent:

```
cp -pr /etc/dd-agent/conf.d/docker.yaml.example /etc/dd-agent/conf.d/
docker.yaml
sudo /etc/init.d/datadog-agent restart
```

Now the agent on our host machine has been configured and the final step is to enable the integration through the website. To do this, go to `https://app.datadoghq.com/` and click on **Integrations**, scroll down and then click on install on **Docker**:



Once you click install, you will be presented with an overview of the integration, click on the **Configuration** tab, this gives instructions on how to configure the agent; as we have already done this step, you can click on **Install Integration**.

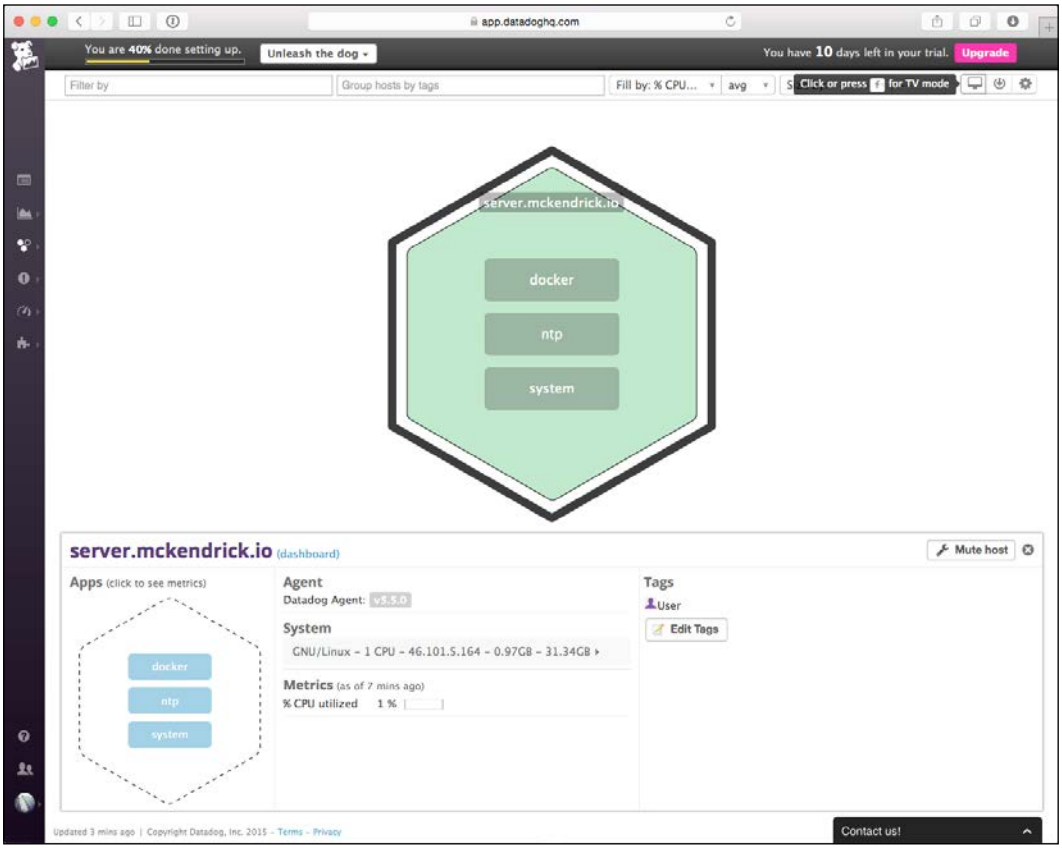
You can find more information on installing the agent and the integrations at the following URLs:

- `https://app.datadoghq.com/account/settings#agent`
- `https://app.datadoghq.com/account/settings#integrations`

Exploring the web interface

Now, you have installed the agent and enabled the Docker integration, you can start to have a look around the web interface. To find your host, click on "Infrastructure" in the left-hand side menu.

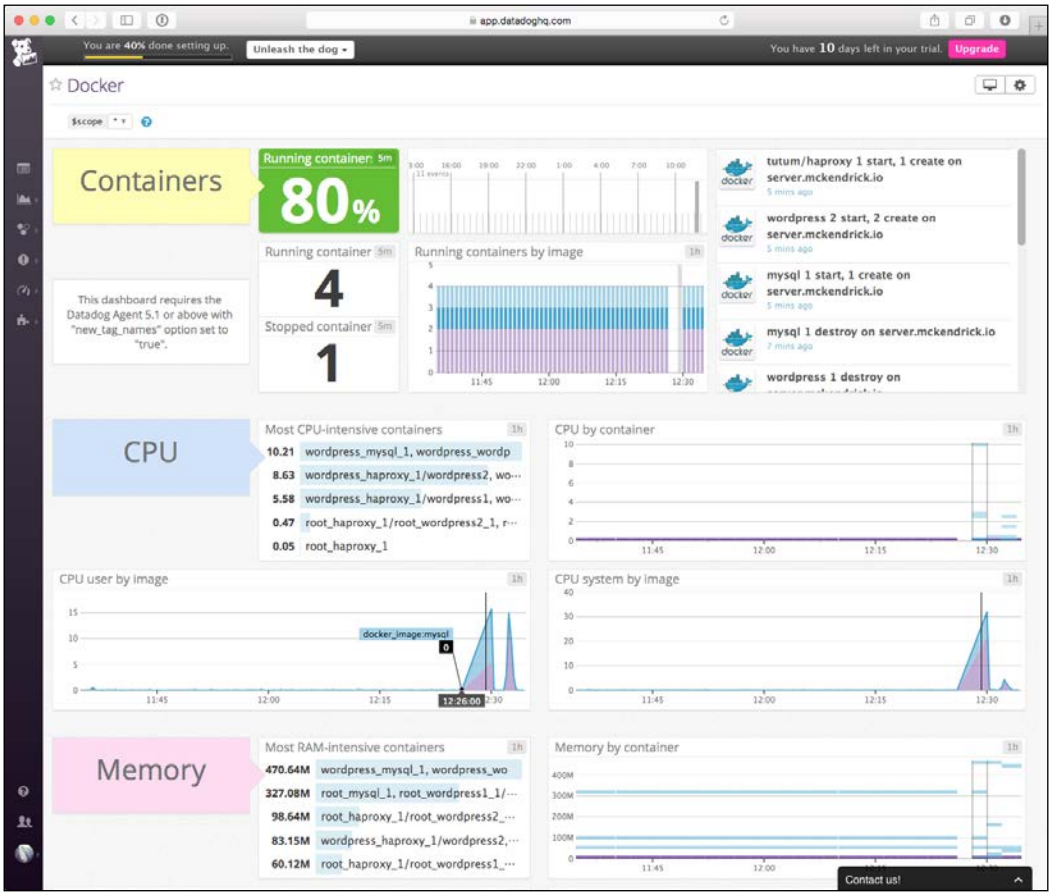
You should be taken to a screen that contains a map of your infrastructure. Like me, you probably only have a single host machine listed, click on it and some basic stats should appear at the bottom of the screen:



If you don't already have the containers launched, now would be a good time to do so, lets launch the WordPress installation again using:

```
sudo su -

mkdir ~/wordpress

curl -L https://raw.githubusercontent.com/russmckendrick/monitoring-
docker/master/chapter05/wordpress/docker-compose.yml > ~/wordpress/
docker-compose.yml

cd ~/wordpress

docker-compose up -d
```

Now, go back to the web interface, from there you can click on any of the services listed on the hexagon. This will bring up some basic metrics for the service you have selected. If you click on **docker**, you will see a link for a Docker Dashboard among the various graphs and so on; clicking this will take you to a more detailed view of your containers:



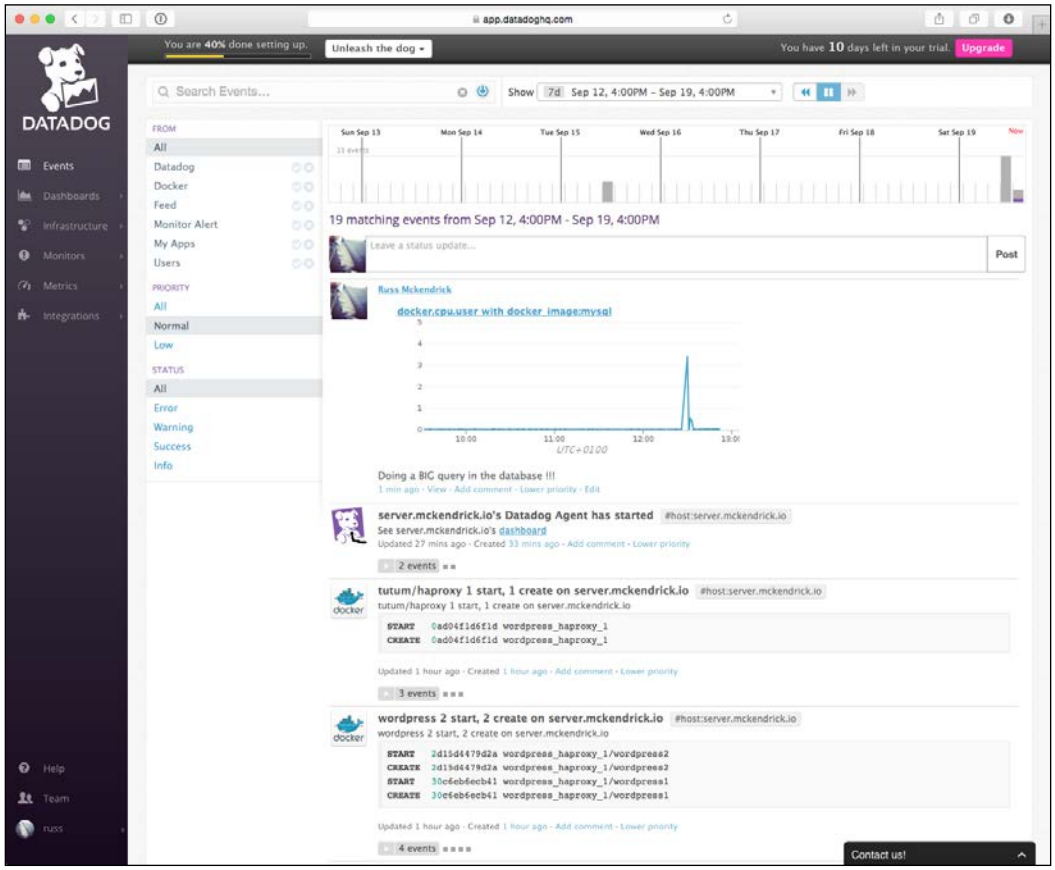
As you can see, this gives us our now familiar break down of the CPU and memory metrics, along with in the top right of the dashboard a breakdown of the container activity on the host machine; this logs events, such as stopping and starting containers.

Datadog currently records the following metrics:

- `docker.containers.running`
- `docker.containers.stopped`

- `docker.cpu.system`
- `docker.cpu.user`
- `docker.images.available`
- `docker.images.intermediate`
- `docker.mem.cache`
- `docker.mem.rss`
- `docker.mem.swap`

From the **Metrics** explorer option in the left-hand side menu, you can start to graph these metrics and once you have the graphs, you can then start to add them to your own custom dashboards or even annotate them. When you annotate a graph, a snapshot is created and the graph shows up in the events queue along with the other events, that have been recorded, such as container stopping and starting:



Also, within the web interface you can configure monitors; these allow you to define triggers, which alert you if your conditions are not met. Alerts can be sent as e-mails or via third party services, such as Slack, Campfire, or PagerDuty.

Summary and further reading

While Datadog's Docker integration only gives you the basic metrics on your containers, it does have a wealth of features and integration with other applications and third parties. If you need to monitor a number of different services alongside your Docker containers, then this service could be for you:

- **Home page**: `https://www.datadoghq.com`
- **Overview**: `https://www.datadoghq.com/product/`
- **Monitoring Docker with Datadog**: `https://www.datadoghq.com/blog/monitor-docker-datadog/`
- **Twitter**: `https://twitter.com/datadoghq`

Please Remember

If you have launched a cloud instance and are no longer using it then now would be a good time to power the instance down or terminate it altogether. This will ensure that you do not get billed for any services you are not using.

New Relic

New Relic could be considered the granddaddy of SaaS monitoring tools, chances are that if you are a developer you will have heard of New Relic. It has been around for a while and it is the standard to which other SaaS tools compare themselves.

New Relic has grown into several products over the year, currently, they offer:

- **New Relic APM**: The main application performance-monitoring tool. This is what most people will know New Relic for; this toll gives you the code level visibility of your application.
- **New Relic Mobile**: A set of libraries to embed into your native mobile apps, giving APM levels of detail for your iOS and android application.
- **New Relic Insights**: A high-level view of all of the metrics collected by other New Relic services.

- **New Relic Servers**: Monitors your host servers, recording metrics around CPU, RAM, and storage utilization.
- **New Relic Browser**: Gives you an insight into what happens with your web-based applications once they leave your servers and enter your end user's browser
- **New Relic Synthetics**: Monitors your applications responsiveness from various locations around the world.

Rather than looking at all of these offerings that give us an insight into what is happening with our Docker-based code, as that's probably a whole module on its own, we are going to take a look at the server product.

The server monitoring service offered by New Relic is available free of charge, you just need an active New Relic account, you can sign up for an account at `https://newrelic.com/signup/` details on New Relics pricing can be found at their homepage at `http://newrelic.com/`.

Installing the agent

Like the other SaaS offerings we have looked at in this chapter, New Relic Servers has a host-based client, which needs to be able to access the Docker binary. To install this on a CentOS machine, run the following:

```
yum install http://download.newrelic.com/pub/newrelic/el5/i386/newrelic-repo-5-3.noarch.rpm
yum install newrelic-sysmond
```

For Ubuntu, run the following command:

```
echo 'deb http://apt.newrelic.com/debian/ newrelic non-free' | sudo tee /etc/apt/sources.list.d/newrelic.list
wget -O- https://download.newrelic.com/548C16BF.gpg | sudo apt-key add -
apt-get update
apt-get install newrelic-sysmond
```

Now that you have the agent installed, you need to configure the agent with your license key. You can do this with the following command and make sure that you add your license, which can be found in your settings page:

```
nrsysmond-config --set license_key= wn5AYlhjRhgn3shcjW14y3yOT09WsF7d
```

Now that the agent is configured, we need to add the `newrelic` user to the `docker` group so that the agent has access to our container information:

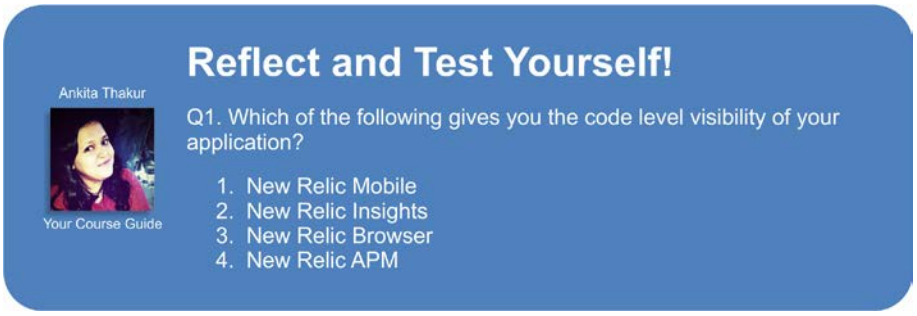
```
usermod -a -G docker newrelic
```

Finally, we need to start the New Relic Server agent and restart Docker:

```
/etc/init.d/newrelic-sysmond restart
/etc/init.d/docker restart
```

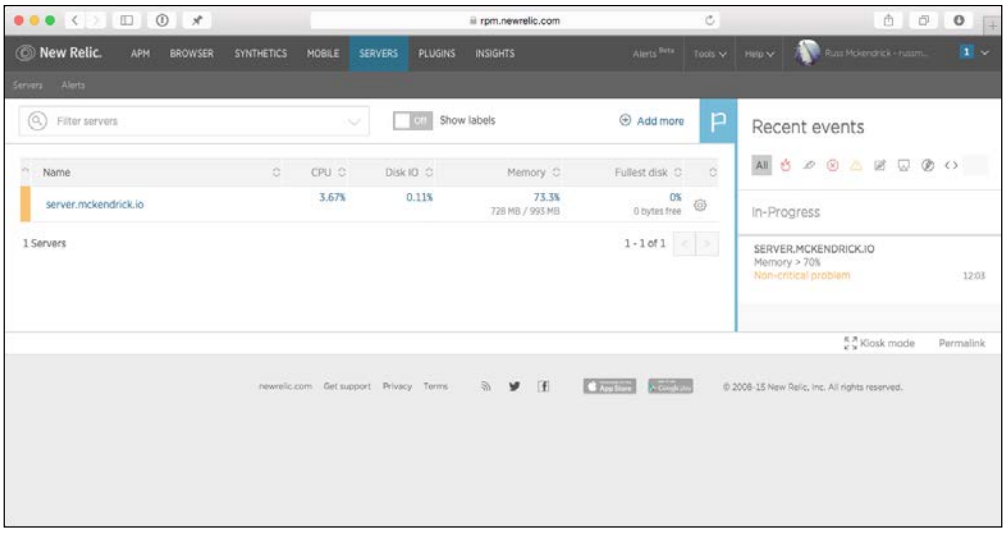
Restarting Docker will stop the running containers that you have; make sure that you make a note of these using `docker ps` and then start them manually and back up once the Docker service restarts.

You should see your server appear on your New Relic control panel after a few minutes.

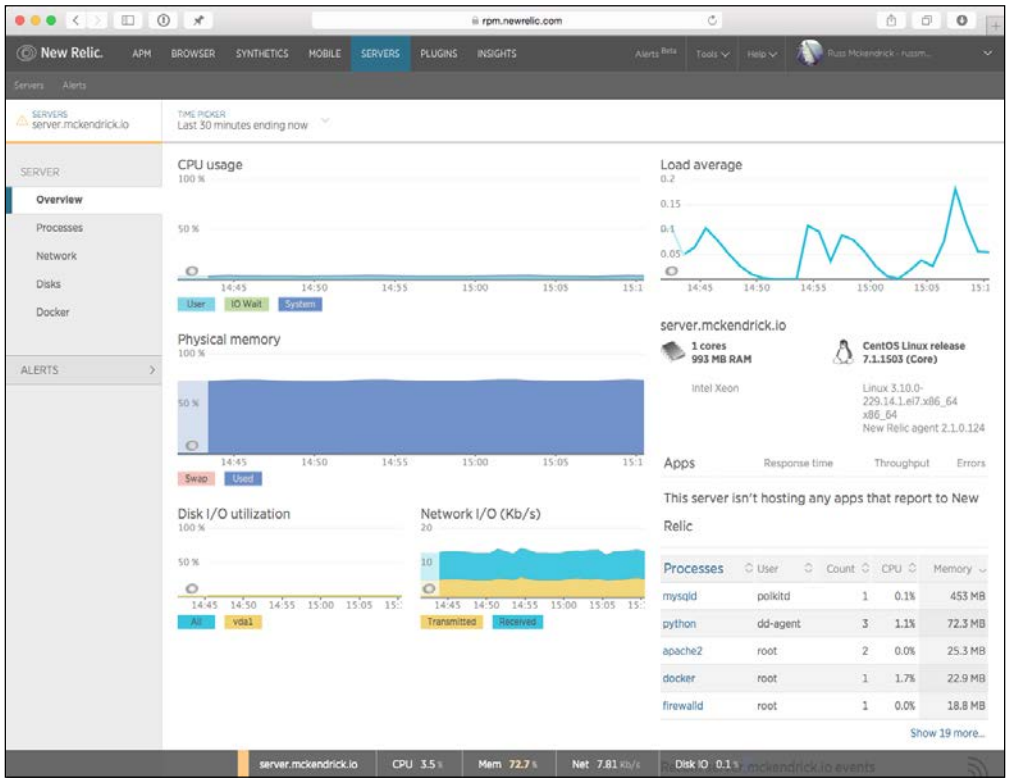


Exploring the web interface

Once you have the New Relic server agent installed, configured, and running on your host machine, you will see something similar to the following screenshot when clicking on **Servers** in the top menu:



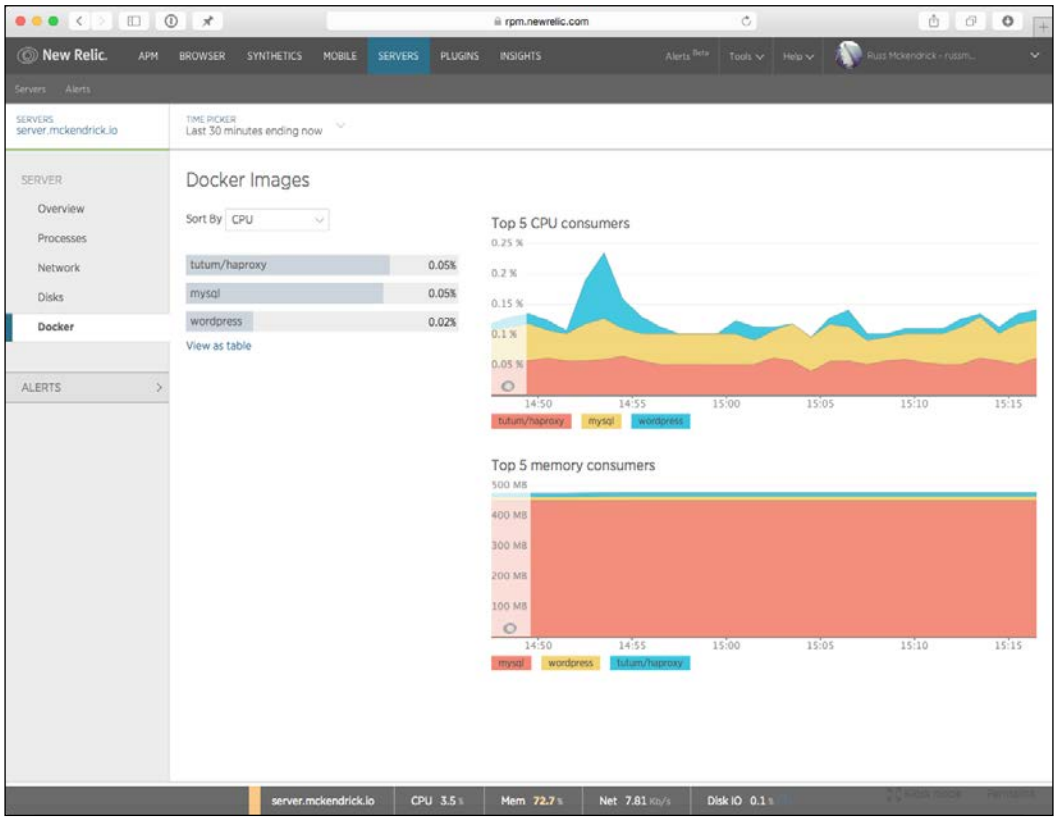
Selecting the server will allow you to start exploring the various metrics that the agent is recording:

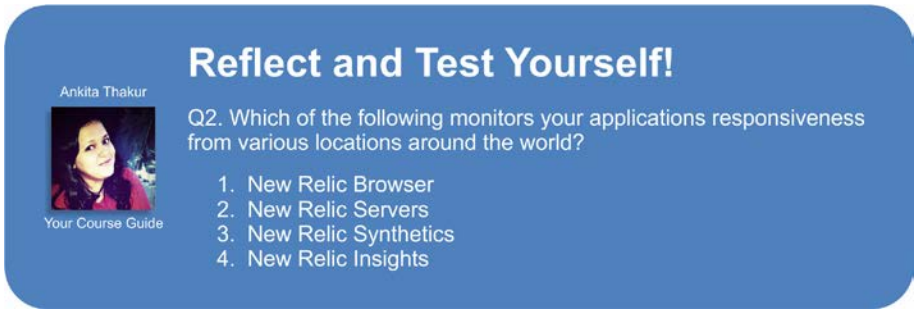


From here, you have the option to drill down further:

- **Overview**: Gives a quick overview of your host machine
- **Processes**: Lists all of the processes that are running both on the host machine and within your containers
- **Network**: Lets you see the network activity for your host machine
- **Disks**: Gives you details on how much space you are using
- **Docker**: Shows you the CPU and memory utilization for your containers

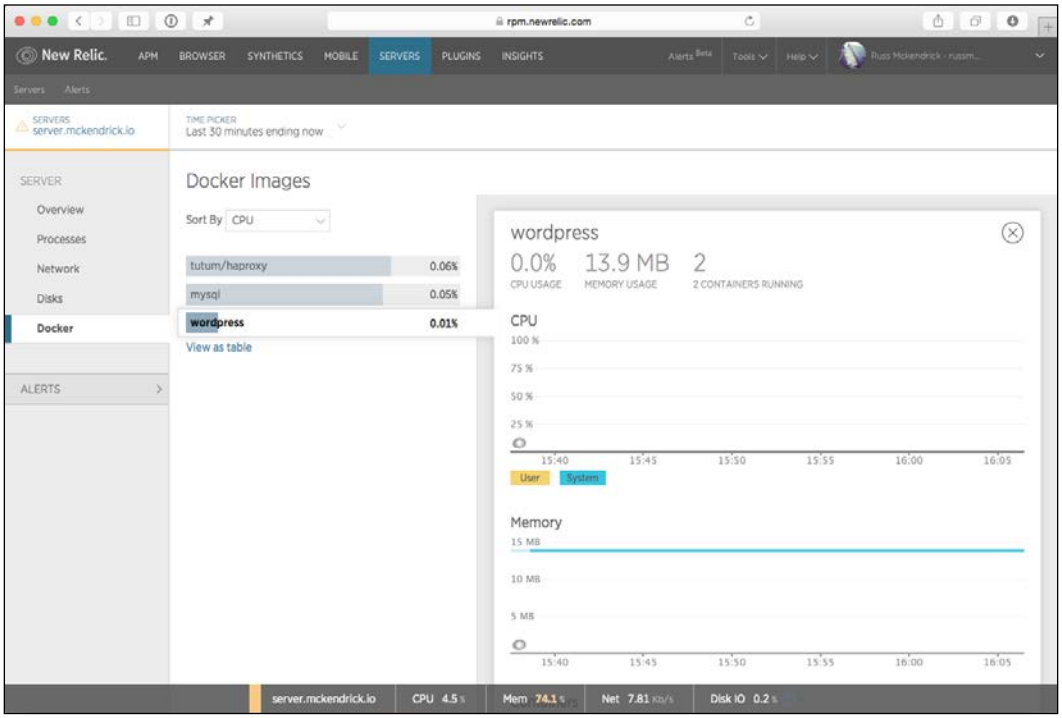
As you may have guessed, we are going to be looking at the **Docker** item next, click on it and you will see a list of your active images:





You may have noticed a difference between New Relic and the other services, as you can see New Relic does not show you the running containers, instead it shows you the utilization by Docker image.

In the preceding screenshot, I have four containers active and running the WordPress installation we have used elsewhere in the module. If I wanted a breakdown per container, then I would be out of luck, as demonstrated by the following screen:



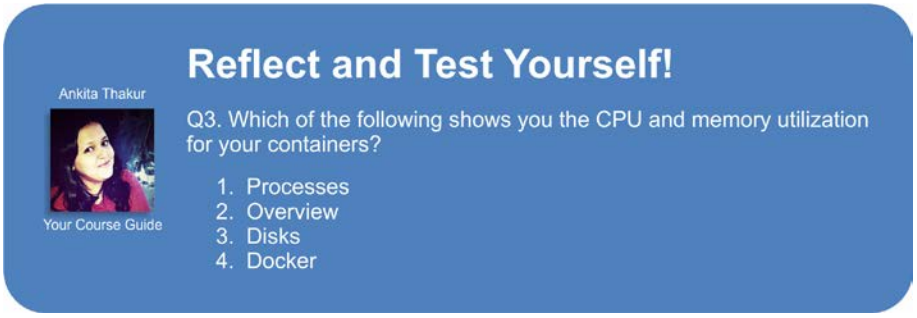
It's a pretty dull screen, but it gives you an idea about what you will see if you are running multiple containers that have been launched using the same image. So how is this useful? Well, coupled with the other services offered by New Relic, it can give you an indication of what your containers were up to when a problem occurred within your application. If you remember the Pets versus Cattle versus Chickens analogy from *Chapter 1, Introduction to Docker Monitoring*, we don't necessarily care which container did what; we just want to see the impact it had during the issue we are looking into.

Summary and further reading

Due to the amount of products it offers, New Relic can be a little daunting at first, but if you work with a development team that actively uses New Relic within their day-to-day workflow, then having all of the information about your infrastructure alongside this data can be both valuable and necessary, especially during an issue:

- New Relic Server monitoring: `http://newrelic.com/server-monitoring`
- New Relic and Docker: `http://newrelic.com/docker/`
- Twitter: `https://twitter.com/NewRelic`

If you have launched a cloud instance and are no longer using it then, now is a good time to power the instance down or terminate it altogether, this will ensure you do not get billed for any services you are not using.



Your Coding Challenge

Ankita Thakur

Your Course Guide

Which SaaS service you choose depends on your circumstances, there are a number of questions you should ask yourself before you start evaluating the SaaS offerings:

- How many containers would you like to monitor?
- How many host machines do you have?
- Is there a non-containerized infrastructure you need to monitor?
- What metrics do you need from the monitoring service?
- How long should the data be retained for?
- Could other departments, such as development and utilize the service?

We covered just three of the available SaaS options in this chapter, there are other options available. Can you list some?

Summary of Module 3 Chapter 6

Ankita Thakur

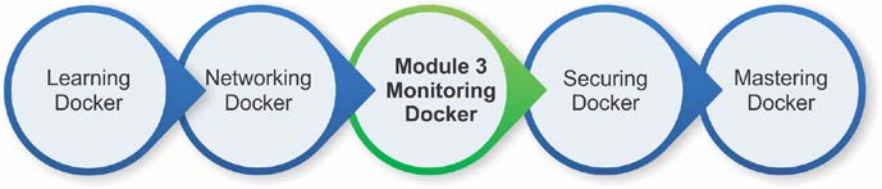
Your Course Guide

Monitoring servers and services are only as good as the metrics you collect, if possible and if your budget allows, you should take full advantage of the services offered by your chosen providers, as more data being recorded by a single provider will only benefit you when it comes to analyzing problems with not only your containerized applications, but also with your infrastructure, code and even your cloud provider.

For example, if you are monitoring your host machine using the same service as you use to monitor your containers, then by using the custom graphing functions, you should be able to create overlay graphs of CPU load spikes of both your host machine and your container. This is a lot more useful than trying to compare two different graphs from different systems side by side.

In the next chapter, we will look at an often-overlooked part of monitoring: shipping your log files away from your containers/hosts to a single location so that they can be monitored and reviewed.

Your Progress through the Course So Far



7
Collecting Application Logs from within the Container

One of the most overlooked parts of monitoring are log files generated by the application or services such as NGINX, MySQL, Apache, and so on. So far we have looked at various ways of recording the CPU and RAM utilization of the processes within your containers are at a point in time, now its time to do the same for the log files.

If you are running your containers as Cattle or Chickens, then the way you deal with the issues to destroy and relaunch your container either manually or automatically is important. While this should fix the immediate problem, it does not help with tracking down the root cause of the issue and if you don't know that then how can you attempt to resolve it so that it does not reoccur.

In this chapter, we will look at how we can get the content of the log files for the applications running within our containers to the central location so that they are available, even if you have to destroy and replace a container. We are going to cover the following topics in this chapter:

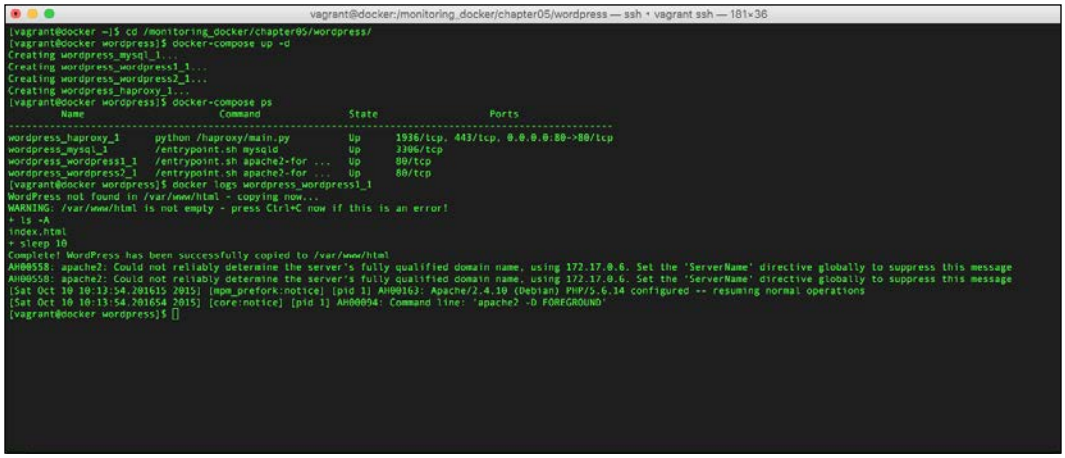
- How to view container logs?
- Deploying an "ELK" stack using a Docker containers stack to ship the logs to
- Reviewing your logs
- What third party options are available?

Viewing container logs

Like the `docker top` command, there is a very basic way of viewing logs. When you use the `docker logs` command, you are actually viewing the `STDOUT` and `STDERR` of the processes that are running within the container.

For more information on Standard Streams, please see `https://en.wikipedia.org/wiki/Standard_streams`.

As you can see from the following screenshot, the simplest thing you have to do is run `docker logs` followed by your container name:



To see this on your own host, let's launch the WordPress installation from `chapter05` using the following commands:

```
cd /monitoring_docker/chapter05/wordpress/
docker-compose up -d
docker logs wordpress_wordpress1_1
```

You can extend the `dockerlogs` command by adding the following flags before your container name:

- `-f` or `--follow` will stream the logs in real time
- `-t` or `--timestamps` will show a timestamp at the start of each line
- `--tail="5"` will show the last *x* number of lines
- `--since="5m00s"` will show only the entries for the last 5 minutes

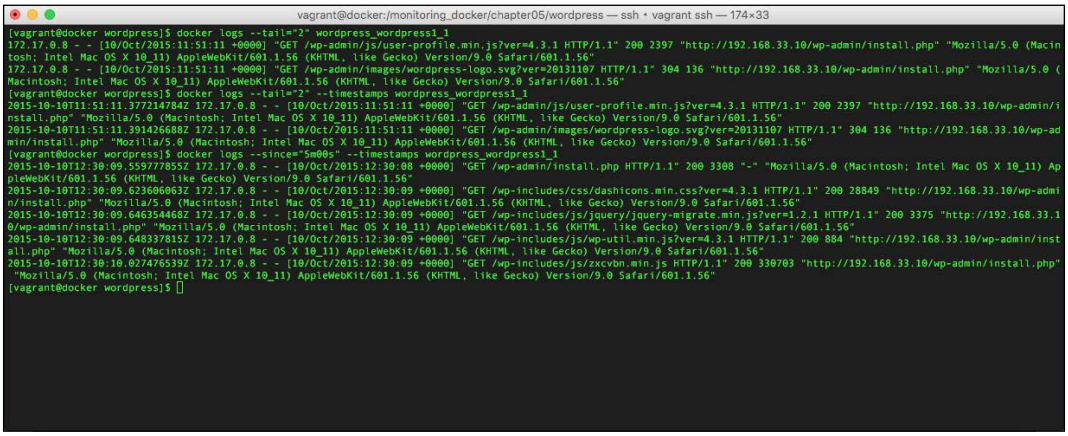
Using the WordPress installation that we have just launched, try running the following commands:

```
docker logs --tail="2" wordpress_wordpress1_1
```

This will show the last two lines of the logs, you can add timestamps using:

```
docker logs --tail="2" –timestamps wordpress_wordpress1_1
```

As you can see in the following terminal output, you can also string commands together to form a very basic query language:



The downside of using `docker logs` is exactly the same as using `docker top`, in that it is only available locally and the logs are only present for the time the container is around, you can view the logs of a stopped container, but once the container is removed, so are the logs.

ELK Stack

Similar to some of the technologies that we have covered in this module, an ELK stack really deserves a module by itself; in fact, there are modules for each of the elements that make an ELK stack, these elements are:

- Elasticsearch is a powerful search server, which has been developed with modern workloads in mind
- Logstash sits between your data source and Elasticsearch services; it transforms your data in real time to a format, which Elasticsearch can understand.
- Kibana is in front of your Elasticsearch services and allows you to query your data in a feature-rich web-based dashboard.

There are a lot of moving parts with an ELK stack, so to simplify things, we will use a prebuilt stack for the purpose of testing; however, you probably don't want to use this stack in production.

Starting the stack

Let's launch a fresh vagrant host on which to run the ELK stack:

```
[russ@mac ~]$ cd ~/Documents/Projects/monitoring-docker/vagrant-centos/
[russ@mac ~]$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'russmckendrick/centos71'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'russmckendrick/centos71' is up to date...

.....

==> default: => Installing docker-engine ...
==> default: => Configuring vagrant user ...
==> default: => Starting docker-engine ...
==> default: => Installing docker-compose ...
==> default: => Finished installation of Docker
[russ@mac ~]$ vagrant ssh
```

Now, we have a clean host that is up and running, we can start the stack by running the following commands:

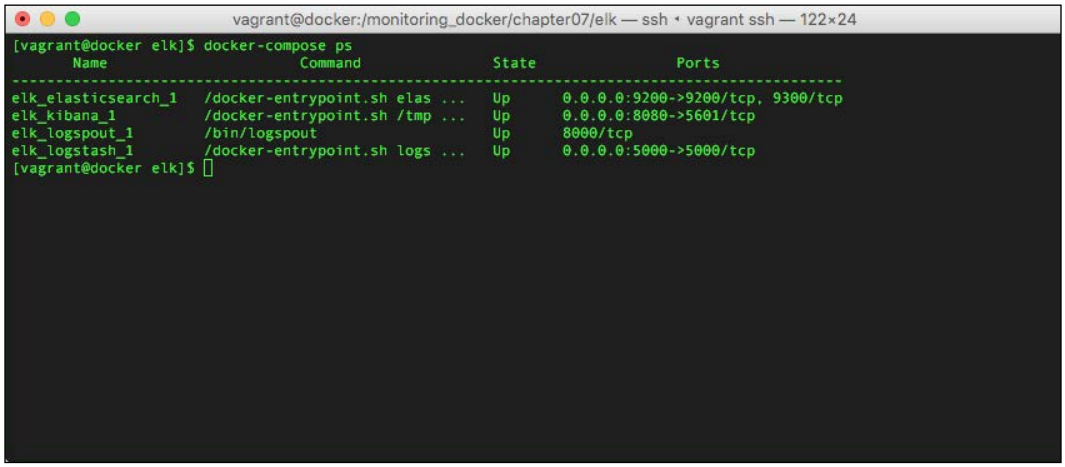
```
[vagrant@docker ~]$ cd /monitoring_docker/chapter07/elk/
[vagrant@docker elk]$ docker-compose up -d
```

As you may have noticed, it did more that just pull down some images; what happened was:

- An Elasticsearch container was launched using the official image from `https://hub.docker.com/_/elasticsearch/`.
- A Logstash container was launched using the official image from `https://hub.docker.com/_/logstash/`, it was also launched with our own configuration, which means that our installation listens for logs sent from Logspout (more about that in a minute).

- A custom Kibana image was built using the official image from `https://hub.docker.com/_/kibana/`. All it did was add a small script to ensure that Kibana doesn't start until our Elasticsearch container is fully up and running. It was then launched with a custom configuration file.
- A custom Logspout container was built using the official image from `https://hub.docker.com/r/gliderlabs/logspout/` and then we added a custom module so that Logspout could talk to Logstash.

Once `docker-compose` has finished building and launching the stack you should be able to see the following when running `docker-compose ps`:



We now have our ELK stack up and running, as you may have noticed, there is an additional container running and giving us an ELK-L stack, so what is Logspout?

Logspout

If we were to launch Elasticsearch, Logstash, and Kibana containers, we should have a functioning ELK stack but we will have a lot of configuration to do to get our container logs into Elasticsearch.

Since Docker 1.6, you have been able to configure logging drivers, this meant that it is possible to launch a container and have it send its `STDOUT` and `STDERR` to a Syslog Server, which will be Logstash in our case; however, this means that you will have to add something similar to the following options each time we launch a container:

```
--log-driver=syslog --log-opt syslog-address=tcp://elk_logstash_1:5000
```

This is where Logspout comes in, it has been designed to collect all of the `STDOUT` and `STDERR` messages on a host machine by intercepting the messages that are being collected by the Docker process and then it routes them to our Logstash instance in a format that is understood by Elasticsearch.

Just as the log-driver, it supports Syslog out of the box; however, there is a third party module that transforms the output to JSON, which Logstash understands. As a part of our build we downloaded, compiled and configured the module.

You can find out more about Logspout and logging drivers at the following:

- Official Logspout image: `https://hub.docker.com/r/gliderlabs/logspout/`
- Logspout Project page: `https://github.com/gliderlabs/logspout`
- Logspout Logstash module: `https://github.com/looplab/logspout-logstash`
- Docker 1.6 release notes: `https://blog.docker.com/2015/04/docker-release-1-6/`
- Docker Logging Drivers: `https://docs.docker.com/reference/logging/overview/`

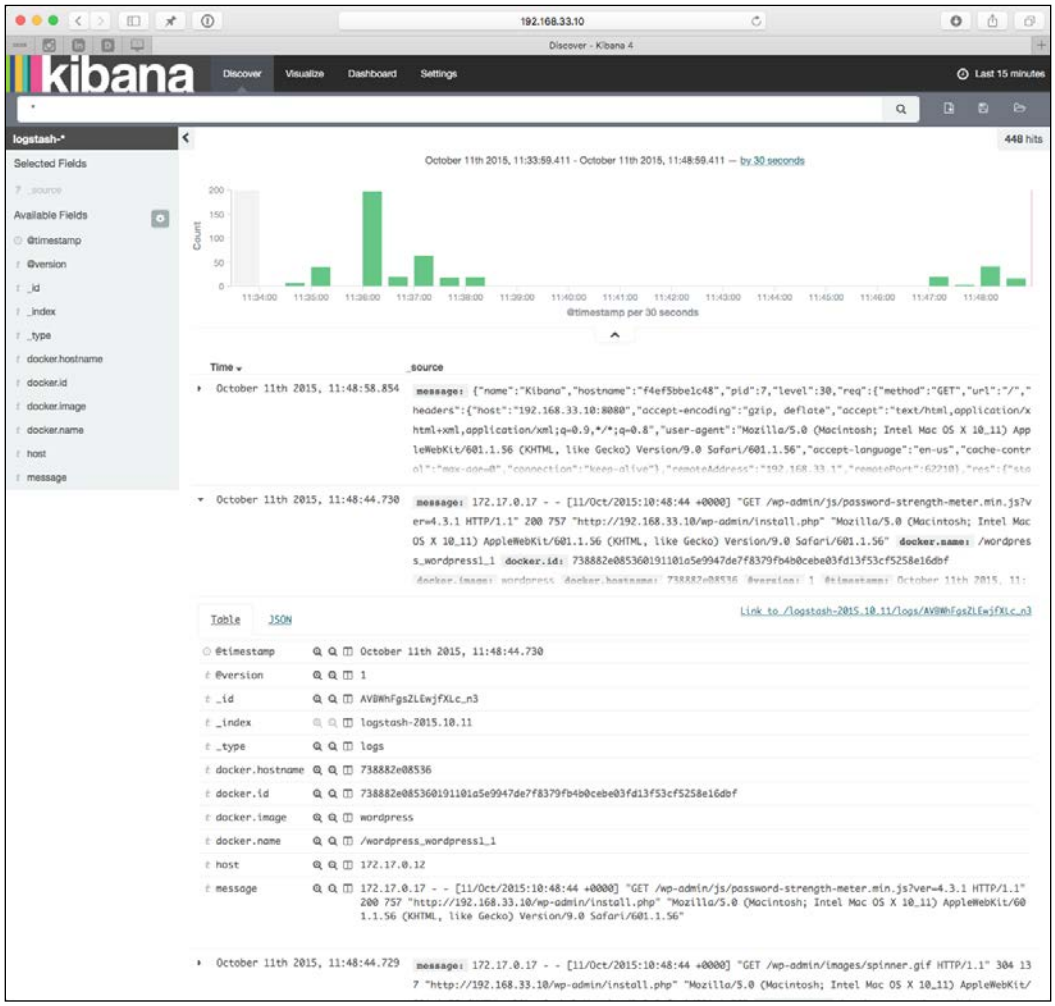
Reviewing the logs

So now, we have our ELK running and a mechanism in place to stream all of the `STDOUT` and `STDERR` messages generated by our containers into Logstash, which in turn routes the data into Elasticsearch. Now its time to view the logs in Kibana. To access Kibana go to `http://192.168.33.10:8080/` in your browser; when you access the page, you will be asked to **Configure an index pattern**, the default index pattern will be fine for our needs so just click the **Create** button.

Once you do, you will see a list of the index patterns, these are taken directly from the Logspout output, and you should notice the following items in the index:

- `docker.name`: The name of container
- `docker.id`: The full container ID
- `docker.image`: The name of the image used to launch the image

From here, if you were to click on **Discover** in the top menu you would see something similar to the following page:



In the screenshot, you will see that I have recently launched the WordPress stack and we have been using it throughout the module, using the following commands:

```
[vagrant@docker elk]$ cd /monitoring_docker/chapter05/wordpress/
[vagrant@docker wordpress]$ docker-compose up -d
```

To give you an idea of what is being logged, here is the raw JSON taken from Elasticseach for running the WordPress installation script:

```
{
  "_index": "logstash-2015.10.11",
  "_type": "logs",
  "_id": "AVBW8ewRnBVdqUV1XVOj",
  "_score": null,
  "_source": {
    "message": "172.17.0.11 - - [11/Oct/2015:12:48:26 +0000]
\"POST /wp-admin/install.php?step=1 HTTP/1.1\" 200 2472
\"http://192.168.33.10/wp-admin/install.php\" \"Mozilla/5.0
(Macintosh; Intel Mac OS X 10_11) AppleWebKit/601.1.56 (KHTML, like
Gecko) Version/9.0 Safari/601.1.56\"",
    "docker.name": "/wordpress_wordpress1_1",
    "docker.id":
"0ba42876867f738b9da0b9e3adbb1f0f8044b7385ce9b3a8a3b9ec60d9f5436c",
    "docker.image": "wordpress",
    "docker.hostname": "0ba42876867f",
    "@version": "1",
    "@timestamp": "2015-10-11T12:48:26.641Z",
    "host": "172.17.0.4"
  },
  "fields": {
    "@timestamp": [
      1444567706641
    ]
  },
  "sort": [
    1444567706641
  ]
}
```

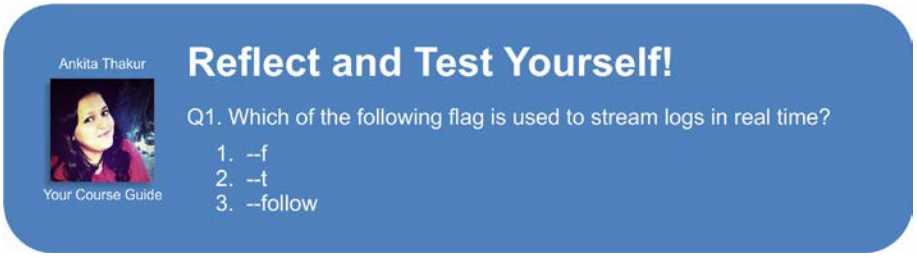
From here, you can start to use the free text search box and build up some quite complex queries to drill down into your container's `STDOUT` and `STDERR` logs.

What about production?

As mentioned at the top of this section, you probably don't want to run your production ELK stack using the `docker-compose` file, which accompanies this chapter. First of all, you will want your Elasticsearch data to be stored on a persistent volume and you more than likely want your Logstash service to be highly available.

There are numerous guides on how to configure a highly available ELK stack, as well as, the hosted services from Elastic, which is the creator of Elasticsearch, and also Amazon Web Services, which offers an Elasticsearch service:

- **ELK tutorial**: `https://www.youtube.com/watch?v=ge8uHdmtb1M`
- **Found from Elastic**: `https://www.elastic.co/found`
- **Amazon Elasticsearch Service**: `https://aws.amazon.com/elasticsearch-service/`



Looking at third party options

There are a few options when it comes to hosting central logging for your containers external to your own server instances. Some of these are:

- **Log Entries**: `https://logentries.com/`
- **Loggly**: `https://www.loggly.com/`

Both of these services offer a free tier. Log Entries also offers a "Logentries DockerFree" account that you can find out more about at `https://logentries.com/docker/`

As recommended in the *Exploring Third-Party Options* chapter, it is best to use a cloud service when evaluating third party services. The remainder of this chapter assumes that you are running a cloud host.

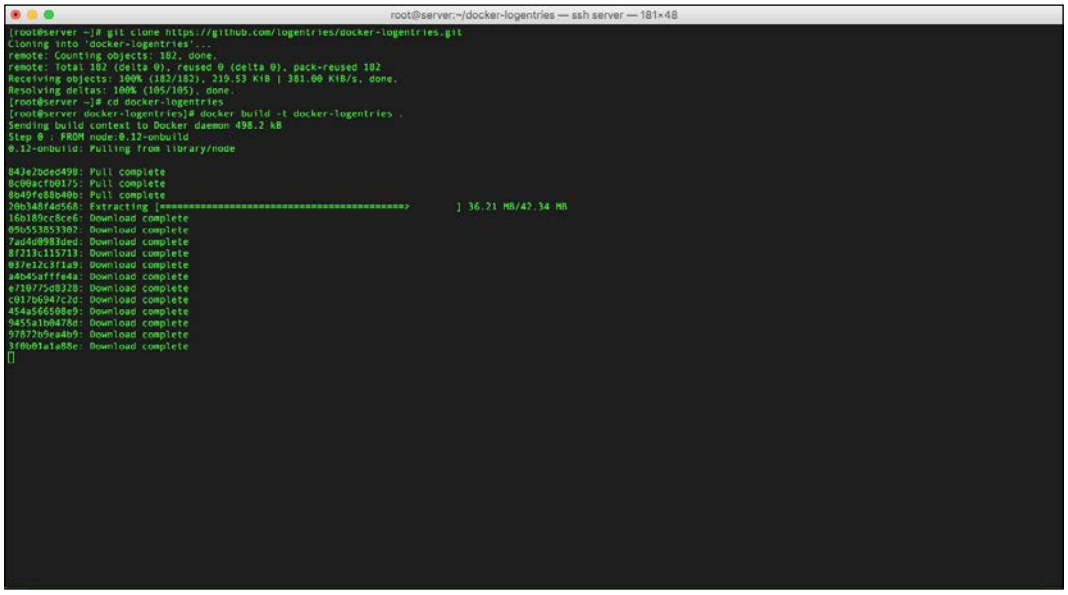
Let's look at configuring the Log Entries on an external server, first of all you need to have signed up for an account at `https://logentries.com/`. Once you have signed up, you should be taken to a page in which your logs will eventually be displayed.

To start, click on the **Add new log** button in the top-right corner of the page and then click the Docker logo in the **Platforms** section.

You have to name your set of logs in the **Select set** section, so give a name to your log set. You now have the choice of building your own container locally using the Docker file from `https://github.com/logentries/docker-logentries`:

```
git clone https://github.com/logentries/docker-logentries.git
cd docker-logentries
docker build -t docker-logentries .
```

After running the preceding command, you will get the following output:



Before you start your container, you will need to generate an access token for your log set by clicking on **Generate Log Token**. Once you have this, you can launch your locally built containers using the following command (replace the token with the one you have just generated):

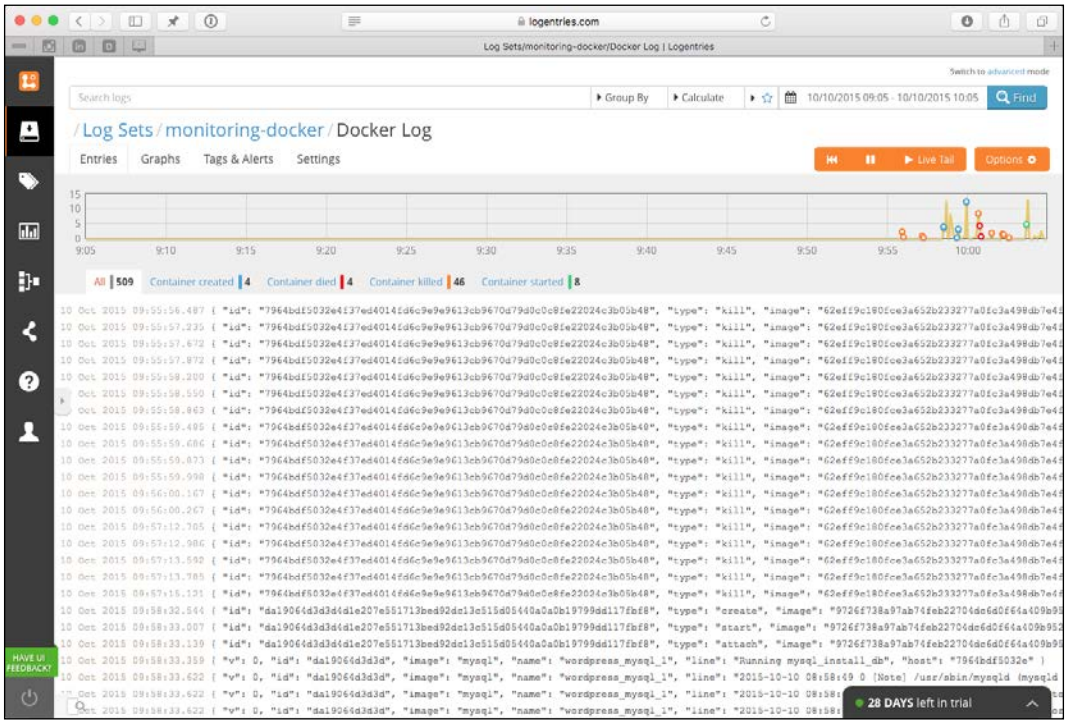
```
docker run -d -v /var/run/docker.sock:/var/run/docker.sock docker-
logentries -t wn5AYlh-jRhgn3shc-jW14y3yO-T09WsF7d -j
```

You can download the image straight from the Docker hub by running:

```
docker run -d -v /var/run/docker.sock:/var/run/docker.sock logentries/
docker-logentries -t wn5AYlh-jRhgn3shc-jW14y3yO-T09WsF7d –j
```

It's worth pointing out that the automatically generated instructions given by Log Entries launches the container in the foreground, rather than detaching from the container once it has been launched like the preceding instructions.

Once you have the `docker-logentries` container up and running, you should start to see logs from your container streamed in real-time to your dashboard:



From here, you will be able to query your logs, create dashboards, and create alerts depending on the account option you go for.

Summary of Module 3 Chapter 7

In this chapter, we have covered how to query the STDOUT and STDERR output from your containers using the tool built into Docker, how to ship the messages to an external source, our ELK stack, and how to store the messages even after the container has been terminated. Finally, we have looked at a few of the third-party services who offer services to which you can stream your logs.

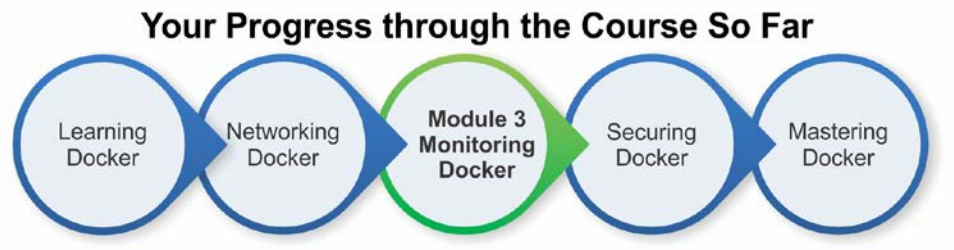
So why go to all of this effort? Monitoring isn't just about keeping and querying CPU, RAM, HDD, and Network utilization metrics; there is no point in knowing if there was a CPU spike an hour ago if you don't have access to the log files to see if any errors were being generated at that time.

The services we have covered in this chapter offer the quickest and most efficient insights into what can quickly become a complex dataset.

In the next chapter, we will look at all of the services and concepts we have covered in the book and apply them to some real world scenarios.

Ankita Thakur

Your Course Guide



8
What Are the Next Steps?

In this final chapter, we will look at the next steps you can take to monitor your containers, by talking about the benefits of adding alerts to your monitoring. Also, we will cover some different scenarios and also which type of monitoring is appropriate for each of them:

- Common problems (performance, availability, and so on) and which type of monitoring is best for your situation.
- What are the benefits of alerting on the metrics you are collecting and what are the options?

Some scenarios

To look at which type of monitoring you might want to implement for your container-based applications, we should work through a few different example configurations that your container-based applications could be deploying into. First, let's remind ourselves about Pets, Cattle, Chickens, and Snowflakes.

Pets, Cattle, Chickens, and Snowflakes

Back in the *Chapter 1*, *Introduction to Docker Monitoring*, we spoke about Pets, Cattle, Chickens, and Snowflakes; in that chapter, we described what each term meant when it was applied to modern cloud deployments. Here, we will go into a little more detail about how the terms can be applied to your containers.

Pets

For your containers to be considered a Pet, you will be more than likely to be running either a single or a small number of fixed containers on a designated host.

Each one of these containers could be considered a single point of failure; if any one of them goes down, it will more than likely result in errors for your application. Worst still, if the host machine goes down for any reason, your entire application will be offline.

This is a typical deployment method for most of our first steps with Docker, and in no way should it be considered bad, frowned upon, or not recommend; as long as you are aware of the limitations, you will be fine.

This pattern can also be used to describe most development environments, as you are constantly reviewing its health and tuning as needed.

You will more than likely be hosting the machine on your local computer or on a hosting service such as DigitalOcean (`https://www.digitalocean.com/`).

Cattle

For the bulk of production or business critical deployments, you should aim to launch your containers in a configuration that allows them to automatically recover themselves after a failure, or, when more capacity is needed, additional containers are launched and then terminated when the scaling event is over.

You will more than likely be using a public cloud-based service as follows:

- **Amazon EC2 Container Service**: `https://aws.amazon.com/ecs/`
- **Google Container Engine**: `https://cloud.google.com/container-engine/`
- Joyent Triton: `https://www.joyent.com/blog/understanding-triton-containers/`

Alternatively, you will be hosting on your own servers using a Docker-friendly and cluster-aware operating system as follows:

- **CoreOS**: `https://coreos.com/`
- **RancherOS**: `http://rancher.com/rancher-os/`

You won't care so much as to where a container is launched within your cluster of hosts, as long as you can route traffic to it. To add more capacity to the cluster, you will be bringing up additional hosts when needed and removing them from the cluster when not needed in order to save on costs.

Chickens

Its more than likely you will be using containers to launch, process data, and then terminate. This can happen anytime from once a day to several times a minute. You will be using a distributed scheduler as follows:

- **Kubernetes by Google**: `http://kubernetes.io/`
- **Apache Mesos**: `http://mesos.apache.org/`

Because of this, you will have a large number of containers launching and terminating within your cluster; you definitely won't care about where a container is launched or even how traffic is routed to it, as long as your data is processed correctly and passed back to your application.

Like the cluster described in the *Cattle* section's description, hosts will be added and removed automatically, probably in response to scheduled peaks such as end of month reporting or seasonal sales and so on.

Snowflakes

I hope one of the things you took away from the first chapter is that if you have any servers or services that you consider being Snowflakes, then you should do something to retire them as soon as possible.

Luckily, due to the way the containerizing of your applications works, you should never be able to create a snowflake using Docker, as your containerized environment should always be reproducible, either because you have the Docker file (everyone makes backups right?) or you have a working copy of the container image because you have exported the container as a whole using the built-in tools.

Sometimes it may not be possible to create a container using a Docker file. Instead, you can backup or migrate your containers by using the export command. For more information on exporting your containers, see the following URL:

`https://docs.docker.com/reference/commandline/export/`

If you find yourself in this position, let me be the first to congratulate you on mitigating a future disaster by promoting your Snowflake into a Pet or even Cattle ahead of any problems.

Still running a Snowflake?

If you find yourself still running a Snowflake server or service, I cannot stress enough that you look at documenting, migrating, or updating the Snowflake as soon as possible. There is no point in monitoring a service that may be impossible for you to recover. Remember that there are containers for old technologies, such as PHP4, if you really need to run them.

Scenario one

You are running a personal WordPress website using the official containers from the Docker Hub; the containers have been launched using a Docker Compose file like the one we have used several times throughout this module.

You have the Docker Compose file stored in a GitHub repository and you can take snapshots of the host machine as a backup. As it's your own blog, you are fine running it on a single cloud-based host.

A suitable monitoring will be as follows:

- Docker stats
- Docker top
- Docker logs
- cAdvisor
- Sysdig

As you are running a single host machine that you are treating as a backup, there is no real need for you to ship your log files to a central location as odds are your host machines; like the containers, its hosting will be online for months or possibly even years.

It is unlikely that you will need to dig too deeply into your containers' historical performance stats, as most of the tuning and troubleshooting will be done in real time as problems occur.

With the monitoring tools suggested, you will be able to get a good insight into what is happening within your containers in real time, and to get more than enough information on processes that are consuming too much RAM and CPU, along with any error messages from within the containers.

You may want to enable a service such as Pingdom (`https://www.pingdom.com/`) or Uptime Robot (`http://uptimerobot.com/`). These services poll your website every few minutes to ensure that the URL you configure them to, check whether its loading within a certain time or at all. If they detect any slowdown or failures with the page loading, they can be configured to send an initial alert to notify you that there is a potential issue, such as both the services mentioned have a free tier.

Scenario two

You are running a custom e-commerce application that needs to be highly available and also scale during your peak times. You are using a public cloud service and the toolset that comes with it to launch containers and route traffic to them.

A suitable monitoring will be as follows:

- cAdvisor + Prometheus
- Zabbix
- Sysdig Cloud
- New Relic Server Monitoring
- Datadog
- ELK + Logspout
- Log Entries
- Loggly

With this scenario, there is a business need to not only be notified about container and host failures, but also to hold your monitoring data and logs away from your host servers so that you can properly review historical information. You may also need to keep logs for PCI compliance or internal auditing for a fixed period of time.

Depending on your budget, you can achieve this by hosting your own monitoring (Zabbix and Prometheus) and central logging (ELK) stacks somewhere within your infrastructure.

You can also choose to run a few different third-party tools such as combining tools that monitor performance, for example, Sysdig Cloud or Datadog, with a central logging service, such as Log Entries or Loggly.

If appropriate, you can also run a combination of self-hosted and third-party tools.

While the self-hosted option may appear to be the most budget-friendly option, there are some considerations to take into account, as follows:

- Your monitoring needs to be hosted away from your application. There is no point in having your monitoring installed on the same host as your application; what will alert you if the host fails?
- Your monitoring needs to be highly available; do you have the infrastructure to do this? If your application needs to be highly available, then so does your monitoring.
- You need to have enough capacity. Do you have the capacity to be able to store log files and metrics going back a month, 6 months, or a year?

If you are going to have to invest in any of the preceding options, then it will be worth weighing up the costs of investing in both the infrastructure and the management of your own monitoring solution against using a third-party that will offer the preceding options as a service.

If you are using a container-only operating system such as CoreOS or RancherOS, then you will need to choose a service whose agent or collector can be executed from within a container, as you will not be able to install the agent binaries directly on the OS.

You will also need to ensure that your host machine is configured to start the agents/collectors on boot. This will ensure that as soon as the host machine joins a cluster (which is typically when containers will start to popup on the host), it is already sending metrics to your chosen monitoring services.

A little more about alerting

A lot of the tools we have looked at in this module offer at least some sort of basic alerting functionality; the million-dollar question is should you enable it?

A lot of this is dependent on the type of application you are running and how the containers have been deployed. As we have already mentioned a few times in this chapter, you should never really have a Snowflake container; this leaves us with Pets, Cattle, and Chickens.

Chickens

As already discussed in the previous section, you probably don't need to worry about getting alerts for RAM, CPU, and hard drive performance on a cluster that is configured to run Chickens.

Your containers should not be up long enough to experience any real problems; however, should there be any unexpected spikes, your scheduler will probably have enough intelligence to distribute your containers to hosts that have the most available resources at that time.

You will need to know if any of your containers have been running longer than you expect them to be up; for example, a process in a container that normally takes no more than 60 seconds is still running after 5 minutes.

This not only means that there is a potential problem, it also means that you find yourself running hosts that only contain stale containers.

Cattle and Pets

When it comes to setting up alerts on Cattle or Pets, you have a few options.

You will more than likely want to receive alerts based on CPU and RAM utilization for both the host machine and the containers, as this could indicate a potential problem that could cause slow down within the application and also loss of business.

As mentioned previously, you will probably also want to be alerted if your application starts to serve the content that is unexpected. For example, a host and a container will quite happily sit there serving an application error.

You can use a service such as Pingdom, Zabbix, or New Relic to load a page and check for the content in the footer; if this content is missing, then an alert can be sent.

Depending on how fluid your infrastructure is, in a Cattle configuration, you will probably want to be alerted when containers spin up and down, as this will indicate periods of high traffic/transactions.

Sending alerts

Sending alerts differs for each tool, for example, an alert could be as simple as sending an email to inform you that there is an issue to the sounding of an audible alert in a **Network Operations Center** (**NOC**) when the CPU load of a container goes above five, or the load on the host goes above 10.

For those of you who require an on-call team to be alerted, most of the software we have covered has some level of integration alert aggregation services such as PagerDuty (`https://www.pagerduty.com`).

These aggregation services either intercept your alert emails or allow services to make API calls to them. When triggered, they can be configured to place phone calls, send SMS messages, and even escalate to secondary on-call technician if an alert has not been flagged down within a definable time.

I can't think of any cases where you shouldn't look at enabling alerting, after all, it's always best to know about anything that could effect your application before your end users do.

How much alerting you enable is really down to what you are using your containers for; however, I would recommend that you review all your alerts regularly and also actively tune your configuration.

The last thing you want is a configuration that produces too many false positives or one that is too twitchy, as you do not want the team who receives your alerts to become desensitized to the alerts that you are generating.

For example, if a critical CPU alert is triggered every 30 minutes because of a scheduled job, then you will probably need to review the sensitivity of the alert, otherwise it is easy for the engineer to simply dismiss a critical alert without thinking about it, as "this alert comes every half an hour and will be ok in a few minutes", when your entire application could be unresponsive.

Keeping up

While Docker has been built on top of well-established technologies such as **Linux Containers** (**LXC**), these have traditionally been difficult to configure and manage, especially for non-system administrators.

Docker removes almost all the barriers to entry, allowing everyone with a small amount of command-line experience to launch and manage their own container-based applications.

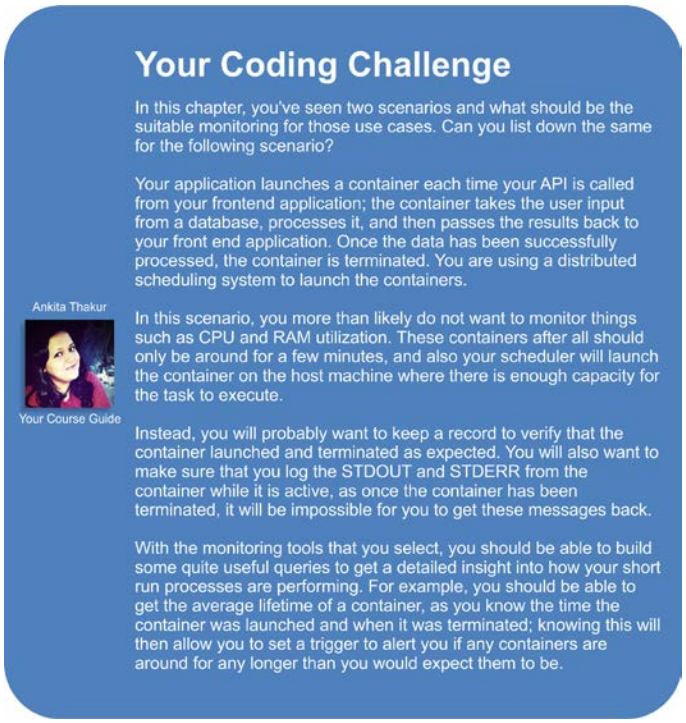
This has forced a lot of the supporting tools to also lower their barrier to entry. Software that once required careful planning to deploy, such as some of the monitoring tools we covered in this module, can now be deployed and configured in minutes rather than hours.

Docker is also a very fast-moving technology; while it has been considered production-ready for a while, new features are being added and existing features are improved with regular updates.

So far, in 2015, there have been 11 releases of Docker Engine; of these, only six have been minor updates that fix bugs, and the rest have all been major updates. Details of each release can be found in the project's Changelog, which can be found at `https://github.com/docker/docker/blob/master/CHANGELOG.md`.

Because of the pace of development of Docker, it is import that you also update any monitoring tools you deploy. This is not only to keep up with new features, but also to ensure that you don't loose any functionality due to changes in the way in which Docker works.

This attitude of updating monitoring clients/tools can be a bit of a change for some administrators who maybe in the past would have configured a monitoring agent on a server and then not thought about it again.



Summary of Module 3 Chapter 8

In this chapter, we have looked at different ways to implement the technologies that have been discussed in the previous chapters of this book. By now, you should have an idea of which approach is appropriate to monitor your containers and host machines, for both your application and for the way the application has been deployed using Docker.

No matter which approach you chose to take, it is important that you stay up-todate with Docker's development and also the new monitoring technologies as they emerge, the following links are good starting points to keep yourself informed:

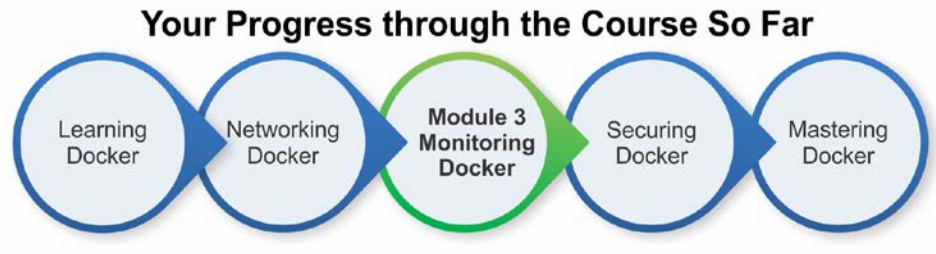
- **Docker Engineering Blog**: http://blog.docker.com/category/engineering/
- **Docker on Twitter**: https://twitter.com/docker
- **Docker on Reddit**: https://www.reddit.com/r/docker
- **Docker on Stack Overflow**: http://stackoverflow.com/questions/tagged/docker

One of the reasons why the Docker project has been embraced by developers, system administrators and even enterprise companies is because it is able to move at a quick pace, while adding more features and very impressively maintaining its ease of use and flexibility.

In coming years, the technology is set to be even more widespread; the importance of ensuring that you are capturing useful performance metrics and logs from your containers will become more critical and I hope that this module has helped you start your journey into monitoring Docker.

Ankita Thakur

Your Course Guide



Course Module 4
Securing Docker

Course Module 1: Learning Docker
Chapter 1: Getting Started with Docker
Chapter 2: Up and Running
Chapter 3: Container Image Storage
Chapter 4: Working with Docker Containers and Images
Chapter 5: Publishing Images
Chapter 6: Running Your Private Docker Infrastructure
Chapter 7: Running Services in a Container
Chapter 8: Sharing Data with Containers
Chapter 9: Docker Machine
Chapter 10: Orchestrating Docker
Chapter 11: Docker Swarm
Chapter 12: Testing with Docker
Chapter 13: Debugging Containers

Course Module 2: Networking Docker
Chapter 1: Docker Networking Primer
Chapter 2: Docker Networking Internals
Chapter 3: Building Your First Docker Network
Chapter 4: Networking in a Docker Cluster
Chapter 5: Next Generation Networking Stack for Docker – libnetwork

Course Module 3: Monitoring Docker
Chapter 1: Introduction to Docker Monitoring
Chapter 2: Using the Built-in Tools
Chapter 3: Advanced Container Resource Analysis
Chapter 4: A Traditional Approach to Monitoring Containers
Chapter 5: Querying with Sysdig
Chapter 6: Exploring Third Party Options
Chapter 7: Collecting Application Logs from within the Container
Chapter 8: What Are the Next Steps?

Course Module 4
Securing Docker

Ankita Thakur

Your Course Guide

Become efficient in securing your Docker apps with Course Module 4, Securing Docker

Course Module 4

Ankita Thakur

Your Course Guide

Security is a very crucial thing and it's important to know the best practices of securing our Docker containers. In this module, you'll learn how to keep your environments secure irrespective of the threats out there. The module starts by sharing the techniques to configure Docker components securely and explore the different security measures/methods one can use to secure the kernel. Furthermore, we will cover the best practices of reporting Docker security findings and will show you how you can safely report any security findings you come across. Finally, we will list the internal and third-party tools that can help you immunize your Docker environment.

By the end of this module, you will have a complete understanding of Docker security and will be able to protect your container-based applications.

1
Securing Docker Hosts

Welcome to the *Securing Docker* module! We are glad you decided to pick up the module and we want to make sure that the resources you are using are being secured in proper ways to ensure system integrity and data loss prevention. It is also important to understand why you should care about the security. If data loss prevention doesn't scare you already, thinking about the worst possible scenario—a full system compromise and the possibility of your secret designs being leaked or stolen by others—might help to reinforce security. Throughout this module, we will be covering a lot of topics to help get your environment set up securely so that you can begin to start deploying containers with peace of mind knowing that you took the right steps in the beginning to fortify your environment. In this chapter, we will be taking a look at securing Docker hosts and will be covering the following topics:

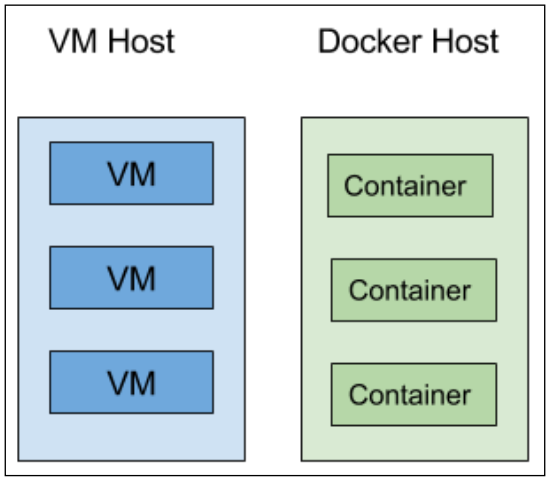
- Docker host overview
- Discussing Docker host
- Virtualization and isolation
- Attack surface of Docker daemon
- Securing Docker hosts
- Docker Machine
- SELinux and AppArmor
- Auto-patching hosts

Docker host overview

Before we get in depth and dive in, let's first take a step back and review exactly what the Docker host is. In this section, we will look at the Docker host itself to get an understanding of what we are referring to when we are talking about the Docker host. We will also be looking at the virtualization and isolation techniques that Docker uses to ensure security.

Discussing Docker host

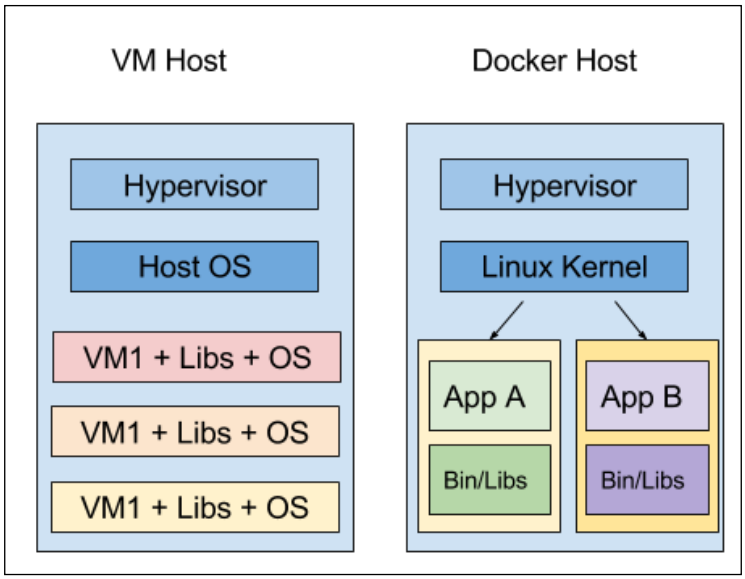
When we think of a Docker host, what comes to our mind? If you put it in terms of virtual machines that almost all of us are familiar with, let's take a look at how a typical VM host differs from a Docker host. A **VM host** is what the virtual machines actually run on top of. Typically, this is something like **VMware ESXi** if you are using VMware or **Windows Server** if you are using **Hyper-V**. Let's take a look at how they are as compared so that you can get a visual representation of the two, as shown in the following diagram:



The preceding image depicts the similarities between a **VM host** and **Docker host**. As stated previously, the host of any service is simply the system that the underlying virtual machines or containers in Docker run on top of. Therefore, a host is the operating system or service that contains and operates the underlying systems that you install and set up a service on, such as web servers, databases, and more.

Virtualization and isolation

To understand how Docker hosts can be secured, we must first understand how the **Docker host** is set up and what items are contained in the **Docker host**. Again, like VM hosts, they contain the operating system that the underlying service operates on. With VMs, you are creating a whole new operating system on top of this **VM host** operating system. However, on Docker, you are not doing that and are sharing the **Linux Kernel** that the **Docker host** is using. Let's take a look at the following diagram to help us represent this:



As we can see from the preceding image, there is a distinct difference between how items are set up on a **VM host** and on a **Docker host**. On a **VM host**, each virtual machine has all of its own items inclusive to itself. Each containerized application brings its own set of libraries, whether it is Windows or Linux. Now, on the **Docker host**, we don't see that. We see that they share the **Linux Kernel** version that is being used on the **Docker host**. That being said, there are some security aspects that need to be addressed on the **Docker host** side of things. Now, on the **VM host** side, if someone does compromise a virtual machine, the operating system is isolated to just that one virtual machine. Back on the **Docker host** side of things, if the kernel is compromised on the **Docker host**, then all the containers running on that host are now at high risk as well.

So, now you should see how important it is that we focus on security when it comes to Docker hosts. Docker hosts do use some isolation techniques that will help protect against kernel or container compromises in a few ways. Two of these ways are by implementing **namespaces** and **cgroups**. Before we can discuss how they help, let's first give a definition for each of them.

Kernel namespaces, as they are commonly known as, provide a form of isolation for the containers that will be running on your hosts. What does this mean? This means that each container that you run on top of your Docker hosts will be given its own network stack so that it doesn't get privileged access to another container's socket or interfaces. However, by default, all Docker containers are sitting on the bridged interface so that they can communicate with each other easily. Think of the bridged interface as a network switch that all the containers are connected to.

Namespaces also provide isolation for processes and mount isolation. Processes running in one container can't affect or even see processes running in another Docker container. Isolation for mount points is also on a container by container basis. This means that mount points on one container can't see or interact with mount points on another container.

On the other hand, control groups are what control and limit resources for containers that will be running on top of your Docker hosts. What does this boil down to, meaning how will it benefit you? It means that cgroups, as they will be called going forward, help each container get its fair share of memory disk I/O, CPU, and much more. So, a container cannot bring down an entire host by exhausting all the resources available on it. This will help to ensure that even if an application is misbehaving that the other containers won't be affected by this application and your other applications can be assured uptime.

Attack surface of Docker daemon

While Docker does ease some of the complicated work in the virtualization world, it is easy to forget to think about the security implications of running containers on your Docker hosts. The largest concern you need to be aware of is that Docker requires root privileges to operate. For this reason, you need to be aware of who has access to your Docker hosts and the Docker daemon as they will have full administrative access to all your Docker containers and images on your Docker host. They can start new containers, stop existing ones, remove images, pull new images, and even reconfigure running containers as well by injecting commands into them. They can also extract sensitive information like passwords and certificates from the containers. For this reason, make sure to also separate important containers if you do need to keep separate controls on who has access to your Docker daemon. This is for containers where people have a need for access to the Docker host where the containers are running. If a user needs API access then that is different and separation might not be necessary. For example, keep containers that are sensitive on one Docker host, while keeping normal operation containers running on another Docker host and grant permissions for other staff access to the Docker daemon on the unprivileged host. If possible, it is also recommended to drop the setuid and setgid capabilities from containers that will be running on your hosts. If you are going to run Docker, it's recommended to only use Docker on this server and not other applications. Docker also starts containers with a very restricted set of capabilities, which works in your favor to address security concerns.

To drop the setuid or setgid capabilities when you start a Docker container, you will need to do something similar to the following:

```
$ docker run -d --cap-drop SETGID --cap-drop SETUID nginx
```

This would start the `nginx` container and would drop the `SETGID` and `SETUID` capabilities for the container.

Docker's end goal is to map the root user to a non-root user that exists on the Docker host. They also are working towards allowing the Docker daemon to run without requiring root privileges. These future improvements will only help facilitate how much focus Docker does take when they are implementing their feature sets.

Protecting the Docker daemon

To protect the Docker daemon even more, we can secure the communications that our Docker daemon is using. We can do this by generating certificates and keys. There are are few terms to understand before we dive into the creation of the certificates and keys. A **Certificate Authority** (**CA**) is an entity that issues certificates. This certificate certifies the ownership of the public key by the subject that is specified in the certificate. By doing this, we can ensure that your Docker daemon will only accept communication from other daemons that have a certificate that was also signed by the same CA.

Now, we will be looking at how to ensure that the containers you will be running on top of your Docker hosts will be secure in a few pages; however, first and foremost, you want to make sure the Docker daemon is running securely. To do this, there are some parameters you will need to enable for when the daemon starts. Some of the things you will need beforehand will be as follows:

1. Create a CA.

```
$ openssl genrsa -aes256 -out ca-key.pem 4096
Generating RSA private key, 4096 bit long modulus
.....................................................................
.....................................................................
.....................................................................
.............++
.....................................................................
..++
e is 65537 (0x10001)
Enter pass phrase for ca-key.pem:
Verifying - Enter pass phrase for ca-key.pem:
```

You will need to specify two values, `pass phrase` and `pass phrase`. This needs to be between `4` and `1023` characters. Anything less than `4` or more than `1023` won't be accepted.

```
$ openssl req -new -x509 -days <number_of_days> -key ca-key.pem
-sha256 -out ca.pem

Enter pass phrase for ca-key.pem:

You are about to be asked to enter information that will be
incorporated

into your certificate request.

What you are about to enter is what is called a Distinguished Name
or a DN.

There are quite a few fields but you can leave some blank

For some fields there will be a default value,

If you enter '.', the field will be left blank.

-----

Country Name (2 letter code) [AU]:US

State or Province Name (full name) [Some-State]:Pennsylvania

Locality Name (eg, city) []:

Organization Name (eg, company) [Internet Widgits Pty Ltd]:

Organizational Unit Name (eg, section) []:

Common Name (e.g. server FQDN or YOUR name) []:

Email Address []:
```

There are a couple of items you will need. You will need `pass phrase` you entered earlier for `ca-key.pem`. You will also need the `Country`, `State`, `city`, `Organization Name`, `Organizational Unit Name`, **fully qualified domain name** (**FQDN**), and `Email Address` to be able to finalize the certificate.

2. Create a client key and signing certificate.

```
$ openssl genrsa -out key.pem 4096

$ openssl req -subj '/CN=<client_DNS_name>' -new -key key.pem -out
client.csr
```

3. Sign the public key.

```
$ openssl x509 -req -days <number_of_days> -sha256 -in client.csr
-CA ca.pem -CAkey ca-key.pem -CAcreateserial -out cert.em
```

4. Change permissions.

```
$ chmod -v 0400 ca-key.pem key.pem server-key.em
$ chmod -v 0444 ca.pem server-cert.pem cert.em
```

Now, you can make sure that your Docker daemon only accepts connections from the other Docker hosts that you provide the signed certificates to:

```
$ docker daemon --tlsverify --tlscacert=ca.pem --tlscert=server-
certificate.pem --tlskey=server-key.pem -H=0.0.0.0:2376
```

Make sure that the certificate files are in the directory you are running the command from or you will need to specify the full path to the certificate file.

On each client, you will need to run the following:

```
$ docker --tlsverify --tlscacert=ca.pem --tlscert=cert.pem --tlskey=key.
pem -H=<$DOCKER_HOST>:2376 version
```

Again, the location of the certificates is important. Make sure to either have them in a directory where you plan to run the preceding command or specify the full path to the certificate and key file locations.

You can read more about using **Transport Layer Security** (**TLS**) by default with your Docker daemon by going to the following link:

`http://docs.docker.com/engine/articles/https/`

For more reading on **Docker Secure Deployment Guidelines**, the following link provides a table that can be used to gain insight into some other items you can utilize as well:

`https://github.com/GDSSecurity/Docker-Secure-Deployment-Guidelines`

Some of the highlights from that website are:

- Collecting security and audit logs
- Utilizing the privileged switch when running Docker containers
- Device control groups
- Mount points
- Security audits

Securing Docker hosts

Where do we start to secure our hosts? What tools do we need to start with? We will take a look at using Docker Machine in this section and how to ensure the hosts that we are creating are being created in a secure manner. Docker hosts are like the front door of your house, if you don't secure them properly, then anybody can just walk right in. We will also take a look at **Security-Enhanced Linux** (**SELinux**) and **AppArmor** to ensure that you have an extra layer of security on top of the hosts that you are creating. Lastly, we will take a look at some of the operating systems that support and do auto patching of their operating systems when a security vulnerability is discovered.

Docker Machine

Docker Machine is the tool that allows you to install the Docker daemon onto your virtual hosts. You can then manage these Docker hosts with Docker Machine. Docker Machine can be installed either through the **Docker Toolbox** on Windows and Mac. If you are using Linux, you will install Docker Machine through a simple `curl` command:

```
$ curl -L https://github.com/docker/machine/releases/download/v0.6.0/
docker-machine-`uname -s`-`uname -m` > /usr/local/bin/docker-machine && \
$ chmod +x /usr/local/bin/docker-machine
```

The first command installs Docker Machine into the `/usr/local/bin` directory and the second command changes the permissions on the file and sets it to executable.

We will be using Docker Machine in the following walkthrough to set up a new Docker host.

Docker Machine is what you should be or will be using to set up your hosts. For this reason, we will start with it to ensure your hosts are set up in a secure manner. We will take a look at how you can tell if your hosts are secure when you create them using the Docker Machine tool. Let's take a look at what it looks like when you create a Docker host using Docker Machine, as follows:

```
$ docker-machine create --driver virtualbox host1


Running pre-create checks...
Creating machine...
Waiting for machine to be running, this may take a few minutes...
Machine is running, waiting for SSH to be available...
```

```
Detecting operating system of created instance...
Provisioning created instance...
Copying certs to the local machine directory...
Copying certs to the remote machine...

Setting Docker configuration on the remote daemon...
```

From the preceding output, as the create is running, Docker Machine is doing things such as creating the machine, waiting for SSH to become available, performing actions, copying the certificates to the correct location, and setting up the Docker configuration, we will see how to connect Docker to this machine as follows:

```
$ docker-machine env host1

export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/scottpgallagher/.docker/machine/machines/
host1"
export DOCKER_MACHINE_NAME="host1"
# Run this command to configure your shell:
# eval "$(docker-machine env host1)"
```

The preceding command output shows the commands that were run to set this machine up as the one that Docker commands will now run against:

```
 eval "$(docker-machine env host1)"
```

We can now run the regular Docker commands, such as `docker info`, and it will return information from `host1`, now that we have set it as our environment.

We can see from the preceding highlighted output that the host is being set up securely from the start from two of the export lines. Here is the first highlighted line by itself:

```
export DOCKER_TLS_VERIFY="1"
```

From the other highlighted output, `DOCKER_TLS_VERIFY` is being set to `1` or `true`. Here is the second highlighted line by itself:

```
export DOCKER_HOST="tcp://192.168.99.100:2376"
```

We are setting the host to operate on the secure port of `2376` as opposed to the insecure port of `2375`.

We can also gain this information by running the following command:

```
$ docker-machine ls
NAME      ACTIVE   DRIVER       STATE     URL
SWARM
host1               *        virtualbox     Running
tcp://192.168.99.100:2376
```

Make sure to check the TLS switch options that can be used with Docker Machine if you have used the previous instructions to set up your Docker hosts and Docker containers to use TLS. These switches would be helpful if you have existing certificates that you want to use as well. These switches can be found in the highlighted section by running the following command:

```
$ docker-machine --help


Options:
  --debug, -D      Enable debug mode
  -s, --storage-path "/Users/scottpgallagher/.docker/machine"
  Configures storage path [$MACHINE_STORAGE_PATH]
  --tls-ca-cert      CA to verify remotes against [$MACHINE_TLS_CA_CERT]
  --tls-ca-key      Private key to generate certificates [$MACHINE_TLS_
CA_KEY]
  --tls-client-cert     Client cert to use for TLS [$MACHINE_TLS_CLIENT_
CERT]
  --tls-client-key       Private key used in client TLS auth [$MACHINE_
TLS_CLIENT_KEY]
  --github-api-token     Token to use for requests to the Github API
[$MACHINE_GITHUB_API_TOKEN]
  --native-ssh      Use the native (Go-based) SSH implementation.
[$MACHINE_NATIVE_SSH]
  --help, -h      show help
  --version, -v      print the version
```

You can also regenerate TLS certificates for a machine using the `regenerate-certs` subcommand in the event that you want that peace of mind or that your keys do get compromised. An example command would look similar to the following command:

```
$ docker-machine regenerate-certs host1


Regenerate TLS machine certs?  Warning: this is irreversible. (y/n): y
```

```
Regenerating TLS certificates
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
```

SELinux and AppArmor

Most Linux operating systems are based on the fact that they can leverage SELinux or AppArmor for more advanced access controls to files or locations on the operating system. With these components, you can limit a container's ability to execute a program as the root user with root privileges.

Docker does ship a security model template that comes with AppArmor and Red Hat comes with SELinux policies as well for Docker. You can utilize these provided templates to add an additional layer of security on top of your environments.

For more information about SELinux and Docker, I would recommend visiting the following website:

`https://www.mankier.com/8/docker_selinux`

While, on the other hand, if you are in the market for some more reading on AppArmor and Docker, I would recommend visiting the following website:

`https://github.com/docker/docker/tree/master/contrib/apparmor`

Here you will find a `template.go` file, which is the template that Docker ships with its application that is the AppArmor template.

Auto-patching hosts

If you really want to get into advanced Docker hosts, then you could use **CoreOS** and **Amazon Linux AMI**, which perform auto-patching, both in a different way. While CoreOS will patch your operating system when a security update comes out and will reboot your operating system, the Amazon Linux AMI will complete the updates when you reboot. So, when choosing which operating system to use when you are setting up your Docker hosts, make sure to take into account the fact that both of these operating systems implement some form of auto-patching, but in a different way. You will want to make sure you are implementing some type of scaling or failover to address the needs of something that is running on CoreOS so that there is no downtime when a reboot occurs to patch the operating system.

Ankita Thakur

Your Course Guide

Your Coding Challenge

Here are some questions for you to know whether you have understood the concepts:

- What is difference between virtualization and isolation?
- What is certificate authority?

Summary of Module 4 Chapter 1

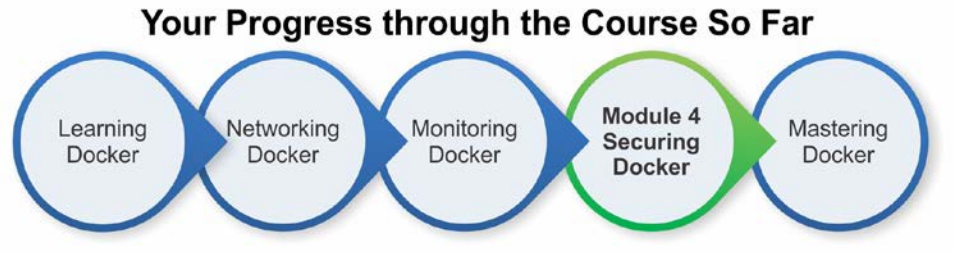
In this chapter, we looked at how to secure our Docker hosts. The Docker hosts are the first line of defense as they are the starting point where your containers will be running and communicating with each other and end users. If these aren't secure, then there is no purpose of moving forward with anything else. You learned how to set up the Docker daemon to run securely running TLS by generating the appropriate certificates for both the host and the clients. We also looked at the virtualization and isolation benefits of using Docker containers, but make sure to remember the attack surface of the Docker daemon too.

Other items included how to use Docker Machine to easily create Docker hosts on secure operating systems with secure communication and ensure that they are being set up using secure methods when you use it to set up your containers. Using items such as SELinux and AppArmor also help to improve your security footprint as well. Lastly, we covered some Docker host operating systems that you can use for autopatching as well, such as CoreOS and Amazon Linux AMI.

In the next chapter, we will be looking at securing the components of Docker. We will focus on securing the components of Docker such as the registry you can use, containers that run on your hosts, and how to sign your images.

Ankita Thakur

Your Course Guide


Your Progress through the Course So Far
Learning Docker
Networking Docker
Monitoring Docker
Module 4 Securing Docker
Mastering Docker

2
Securing Docker Components

In this chapter, we will be taking a look at securing some Docker components using things such as image signing tools. There are tools that will help secure the environments where we are storing our images, whether they are signed or not. We will also look at using tools that come with commercial level support. Some of the tools (image signing and commercial level support tools) we will be looking at are:

- **Docker Content Trust**: Software that can be used to sign your images. We will look at all the components and go through an example of signing an image.
- **Docker Subscription**: Subscription is an all inclusive package that includes a location to store your images, and Docker Engine to run your containers, all while providing commercial level support for all those pieces, plus for the applications and their life cycle that you plan to use.
- **Docker Trusted Registry** (**DTR**): Trusted Registry gives you a secure location to store and manage your images either on premises or in the cloud. It also has a lot of integration into your current infrastructure as well. We will take a look at all the pieces available.

Docker Content Trust

Docker Content Trust is a means by which you can securely sign your Docker images that you have created to ensure that they are from who they say they are from, that being you! In this section, we will take a look at the components of **Notary** as well as an example of signing images. Lastly, we will take a peek at the latest means of using Notary with regards to hardware signing capabilities that are now available. It is a very exciting topic, so let's dive in head first!

Docker Content Trust components

To understand how Docker Content Trust works it is beneficial to be familiar with all the components that make up its ecosystem.

The first part of that ecosystem is **The Update Framework** (**TUF**) piece. TUF, as we will refer to it from now on, is the framework that Notary is built upon. TUF solves the problem with software update systems in that they can often be hard to manage. It enables users to ensure that all applications are secure and can survive any key compromises. However, if an application is insecure by default, it won't help to secure that application until it has been brought up to a secure compliance. It also enables trusted updates over untrusted sources and much more. To learn more about TUF, please visit the website:

```
http://theupdateframework.com/
```

The next piece of the Content Trust ecosystem is that of Notary. Notary is the key underlying piece that does the actual signing using your keys. Notary is open source software, and can be found here:

```
https://github.com/docker/notary
```

This has been produced by those at Docker for the use of publishing and verifying content. Notary consists of a server piece as well as a client piece. The client piece resides on your local machine and handles the storing of the keys locally as well as the communication back with the Notary server to match up timestamps as well.

Basically, there are three steps to the Notary server.

1. Compile the server
2. Configure the server
3. Run the server

Since the steps may change in the future, the best location for that information would be on the GitHub page for Docker Notary. For more information about compiling and setting up the server side of Notary, please visit:

```
https://github.com/docker/notary#compiling-notary-server
```

Docker Content Trust utilizes two distinct keys. The first is that of a tagging key. The tagging key is generated for every new repository that you publish. These are keys that can be shared with others and exported to those who need the ability to sign content on behalf of the registry. The other key, the offline key, is the important key. This is the key that you want to lock away in a vault and never share with anyone… *ever*! Like the name implies, this key should be kept offline and not stored on your machine or anything on a network or cloud storage. The only times you need the offline key are if you are rotating it out for a new one or if you are creating a new repository.

So, what does all this mean and how does it truly benefit you? This helps in protecting against three key, no pun intended, scenarios.

- Protects against image forgery, for instance if someone decides to pretend one of your images is from you. Without that person being able to sign that image with the repository specific key, remember the one you are to keep *offline*!, they won't be able to pass it off as actually coming from you.
- Protects against replay attacks; replay attacks are ones in which a malicious user tries to pass off an older version of an application, which has been compromised, as the latest legitimate version. Due to the way timestamps are utilized with Docker Content Trust, this will ultimately fail and keep you and your users safe.
- Protects against key compromise. If a key is compromised, you can utilize that offline key to do a key rotation. That key rotation can only be done by the one with the offline key. In this scenario, you will need to create a new key and sign it with your offline key.

The major take away from all of this is that the offline key is meant to be kept offline. Never store it on your cloud storage, on GitHub, or even a system that is always connected to the Internet such as that of your local machine. It would be best practice to store it on a thumb drive that is encrypted and keep that thumb drive stored in a a secure location.

To learn more about Docker Content Trust, please visit the following blog post:

`http://blog.docker.com/2015/08/content-trust-docker-1-8/`

Signing images

Now that we have covered all the components of Docker Content Trust, let's take a look at how we can sign an image and what all steps are involved. These instructions are just for development purposes. If you are going to want to run a Notary server in production, you will want to use your own database and compile Notary yourself using the instructions at their website:

`https://github.com/docker/notary#compiling-notary-server`

This will allow you to use your own keys for your situation to your own backend registry. If you are using the Docker Hub, it is very simple to use Docker Content Trust.

```
$ export DOCKER_CONTENT_TRUST=1
```

The most important piece is that you need to put a tag on all images you push, which we see in the next command:

```
$ docker push scottpgallagher/ubuntu:latest


The push refers to a repository [docker.io/scottpgallagher/ubuntu] (len:
1)
f50e4a66df18: Image already exists
a6785352b25c: Image already exists
0998bf8fb9e9: Image already exists
0a85502c06c9: Image already exists
latest: digest: sha256:98002698c8d868b03708880ad2e1d36034c79a6698044b495a
c34c4c16eacd57 size: 8008
Signing and pushing trust metadata
You are about to create a new root signing key passphrase. This
passphrase
will be used to protect the most sensitive key in your signing system.
Please
choose a long, complex passphrase and be careful to keep the password and
the
key file itself secure and backed up. It is highly recommended that you
use a
password manager to generate the passphrase and keep it safe. There will
be no
way to recover this key. You can find the key in your config directory.
Enter passphrase for new root key with id d792b7a:
Repeat passphrase for new root key with id d792b7a:
```

```
Enter passphrase for new repository key with id docker.io/
scottpgallagher/ubuntu (46a967e):
Repeat passphrase for new repository key with id docker.io/
scottpgallagher/ubuntu (46a967e):
Finished initializing "docker.io/scottpgallagher/ubuntu"
```

The most important line from the code above is:

```
latest: digest: sha256:98002698c8d868b03708880ad2e1d36034c79a6698044b495a
c34c4c16eacd57 size: 8008
```

This gives you the SHA hash that is used to verify the image is what it says it is and not created by someone else, as well as its size. This will be used later when someone goes to run that `image/container`.

If you were to do a `docker pull` from a machine that doesn't have this image, you can see it has been signed with that hash.

```
$ docker pull scottpgallagher/ubuntu


Using default tag: latest
latest: Pulling from scottpgallagher/ubuntu
Digest: sha256:98002698c8d868b03708880ad2e1d36034c79a6698044b495ac34c4c16
eacd57
Status: Downloaded newer image for scottpgallagher/ubuntu:latest
```

Again, we see the SHA value being presented when we do the `pull` command.

So, what this means is when you go to run this container, it won't run locally without first comparing the local hash to that on the registry server to ensure it hasn't changed. If they match, it will run, if they don't match, it won't run and will give you an error message about the hashes not matching.

With the Docker Hub you aren't essentially signing images with your own key, unless you manipulate the keys that are in your `~/.docker/trust/trusted-certificates/` directory. Remember that, by default, when you install Docker you are given a set of certificates that you can use.

Hardware signing

Now that we have looked at being able to sign images, which other security measure have been put in place to help make that process even more secure? Enter YubiKeys! YubiKeys is a form of two factor authentication that you can utilize. The way YubiKey works is that it has the root key on it, built into the hardware. You enable Docker Content Trust, then push your image. Upon using your image, Docker notes that you have enabled Content Trust and asks you to touch the YubiKey, yes, physically touch it. This is to ensure that you are a person and not a robot or just a script. You then need to provide a passphase to use and then, once again, touch the YubiKey. Once you have done this, you won't need the YubiKey anymore, but you will need that passphrase that you assigned earlier.

My description of this really doesn't do it justice. At DockerCon Europe 2015 (`http://europe-2015.dockercon.com`), there was a great play-by-play of this in operation between two Docker employees, Aanand Prasad and Diogo Mónica.

To view the video, please visit the following URL:

```
https://youtu.be/fLfFFtOHRZQ?t=1h21m33s
```

Docker Subscription

Docker Subscription is a service for your distributed applications that will help you support those applications as well as deploy them. The Docker Subscription package includes two critical software pieces and a support piece:

- **Docker Registry** — where you store and manage your images (locally hosted or hosted in the cloud)
- **The Docker Engine** — to run those images
- **Docker Universal Control Plane** (**UCP**)
- **Commercial support** — pick up the phone or shoot off an email for some assistance

If you are a developer, sometimes the operations side of things are either a little difficult to get set up and manage or they require some training to get going. With Docker Subscription you can off load some of those worries by utilizing the expertise that is out there with commercial level support. With this support you will get responsive turn around on your issues. You will receive any hot fixes that are available or have been made available to patch your solution. Assistance with future upgrades is also part of the added benefit of choosing the Docker Subscription plan. You will get assistance with upgrading your environments to the latest and most secure Docker environments.

Pricing is broken down based on where you want to run your environment whether it is on a server of your choosing or if it's in a cloud environment. It is also based upon how many Docker Trusted Registries and/or how many commercially supported Docker Engines you wish to have as well. All of these solutions provide you with integration into your existing **LDAP** or **Active Directory** environments. With this added benefit, you can use items such as group policies to manage access to such resources. The last thing that you will have to decide is how quick a response time you want from the support end. All of those will result in the price you pay for the subscription service. No matter what you do pay though the money spent will be well worth it, not only in respect of the peace of mind you will get but the knowledge you will gain is priceless.

You can also change your plans on a monthly or annual basis as well as upgrade, in increments of ten, your Docker Engine instances. You can also upgrade in packs of ten the number of **Docker Hub Enterprise** instances. Switching between an on premises server to the cloud, or vice-versa, is also possible.

To break some things down so as to not be confused, the Docker Engine is the core of the Docker ecosystem. It is the command line tools that you use to run, build, and manage your containers or images. The Docker Hub Enterprise is the location where you store and manage your images. These images can be public or made private. We will learn more about DTR in the next section of this chapter.

For more information about Docker Subscription, please visit the link below. You can sign up for a free 30 day trial, view subscription plans, and contact sales for additional assistance or questions. The subscription plans are flexible enough to conform to your operating environment whether it is something you want support for 24/7 or maybe just half of that:

```
https://www.docker.com/docker-subscription
```

You can also view the breakdown for commercial support here:

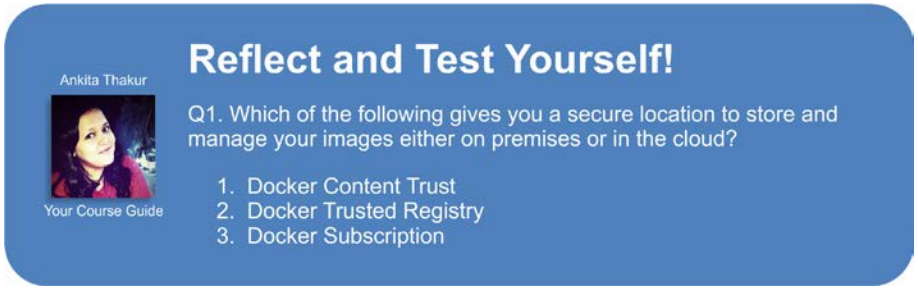
```
https://www.docker.com/support
```

Bringing this all back to the main topic of the module, Securing Docker, this is by far the most secure you can get with your Docker environment that you will be using to manage your images and containers, as well as managing the location they are all stored and run from. A little extra help never hurts and with this option, a little help will defiantly go a long way.

The latest part to be added is the Docker Universal Control Plane. The Docker UCP provides a solution for management of applications and infrastructure that is Dockerized regardless of where they might be running. This could be running on premises or in the cloud. You can find out more information about Docker UCP at the following link:

`https://www.docker.com/products/docker-universal-control-plane`

You can also get a demo of the product using the above URL. Docker UCP is scalable, easy to set up, and you can manage users and access control through integrations into your existing LDAP or Active Directory environments.



Docker Trusted Registry

The DTR is a solution that provides a secure location where you can store and manage your Docker images either on premises or in the cloud. It also provides some monitoring to let you get insight into usage so you can tell what kind of load is being passed to it. DTR, unlike Docker Registry, is not free and does come with a pricing model. As we saw earlier with Docker Subscription, the pricing plan for DTR is the same. Don't fret as we will go over Docker Registry in the next section of the module so you can understand it as well and have all the options available to you for image storage.

The reason we separate it out into its own section is that there are a lot of moving pieces involved and it's critical to understand how they all function not only as a whole to the Docker Subscription piece, but as it stands by itself, the DTR piece where all your images are being maintained and stored.

Installation

There are two ways you can install DTR, or rather there are two locations where you can install DTR. The first is that you can deploy it in house on a server you manage. The other is deploying it to a cloud provider environment like that of **Digital Ocean**, **Amazon Web Services** (**AWS**), or **Microsoft Azure**.

The first part you will need is a license for the DTR. Currently, they do offer a trial license that you can use, which I highly recommend you do. This will allow you to evaluate the software on your selected environment without having to fully commit to that environment. If there is something that you find doesn't work in a particular environment or you feel another location may suit you better, you can then switch without having to be tied to a particular location or having to move your existing environment around to a different provider or location. If you do choose to use AWS, they do have a pre-baked **Amazon Machine Image** (**AMI**) that you can utilize to get your Trusted Registry set up much quicker. This avoids having to do it all manually by hand.

Before you can install the Trusted Registry, you first need to have Docker Engine installed. If you don't already have it installed, please see the documentation located with the link below for more information on doing so.

`https://docs.docker.com/docker-trusted-registry/install/install-csengine/`

You will notice there is a difference in installing the normal Docker Engine from the **Docker CS Engine**. The Docker CS Engine stands for commercially supported Docker Engine. Be sure to check the documentation as the list of recommended or supported Linux versions are shorter than the regular list for Docker Engine.

If you are installing using the AMI, then please follow the instructions here:

`https://docs.docker.com/docker-trusted-registry/install/dtr-ami-byol-launch/`

If you are installing on Microsoft Azure, then please follow the instructions here:

`https://docs.docker.com/docker-trusted-registry/install/dtr-vhd-azure/`

Once you do have Docker Engine installed, it's time to install the DTR piece. If you are reading to this point we will be assuming that you aren't installing to AWS or Microsoft Azure. If you are using either of those two methods, please see the links from above. The installation is very straightforward:

```
$ sudo bash -c '$(sudo docker run docker/trusted-registry install)'
```

Note: You may have to remove the `sudo` options from the above command when running on Mac OS.

Once this has been run, you can navigate in your browser to the IP address of your Docker host. You will then be setting the domain name for your Trusted Registry as well applying the license. The web portal will guide you through the rest of the setup process.

In accessing the portal you can set up authentication through your existing LDAP or Active Directory environments as well, but this can be done at anytime.

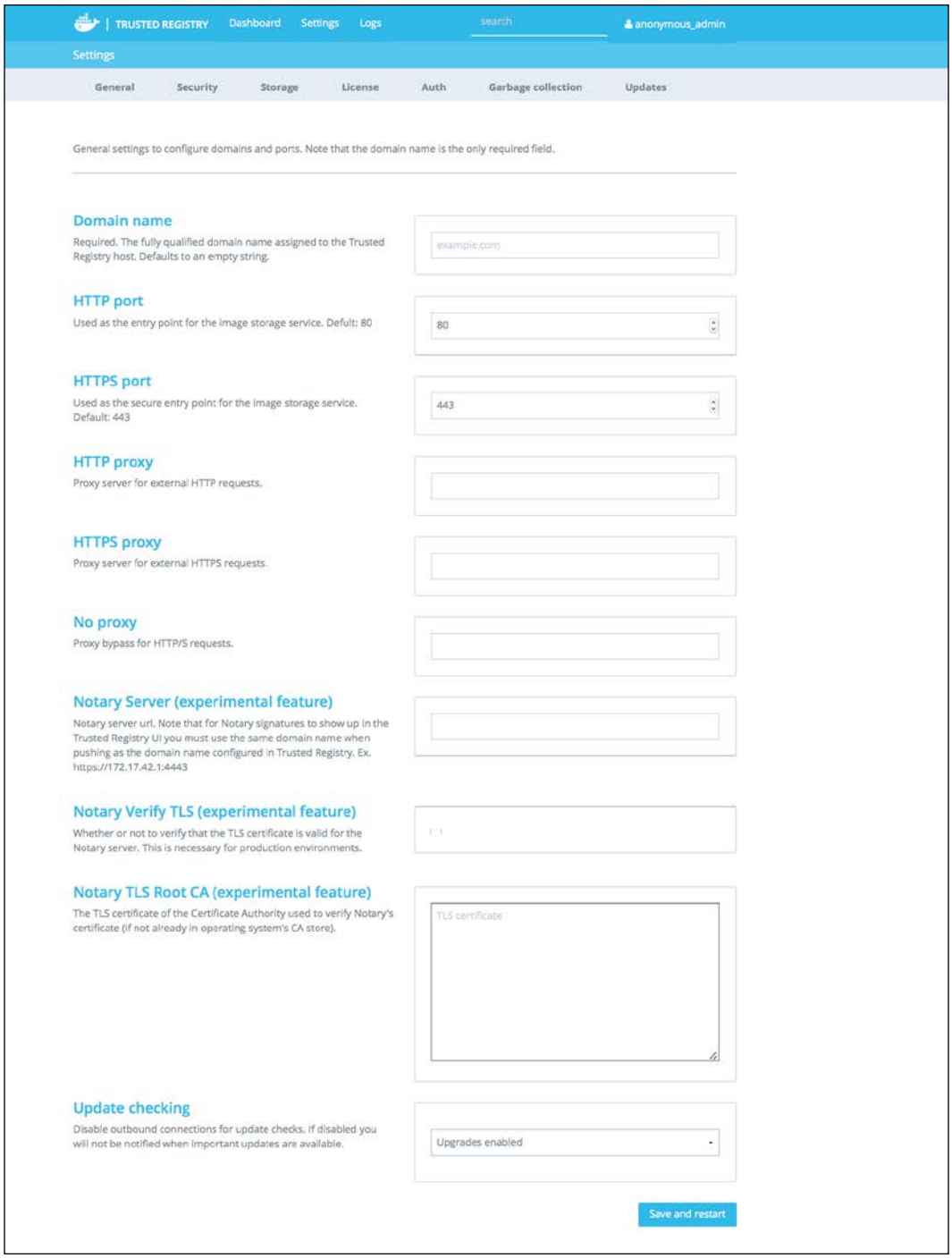
Once that is done, it is time for *Securing Docker Trusted Registry*, which we will cover in the next section.

Securing Docker Trusted Registry

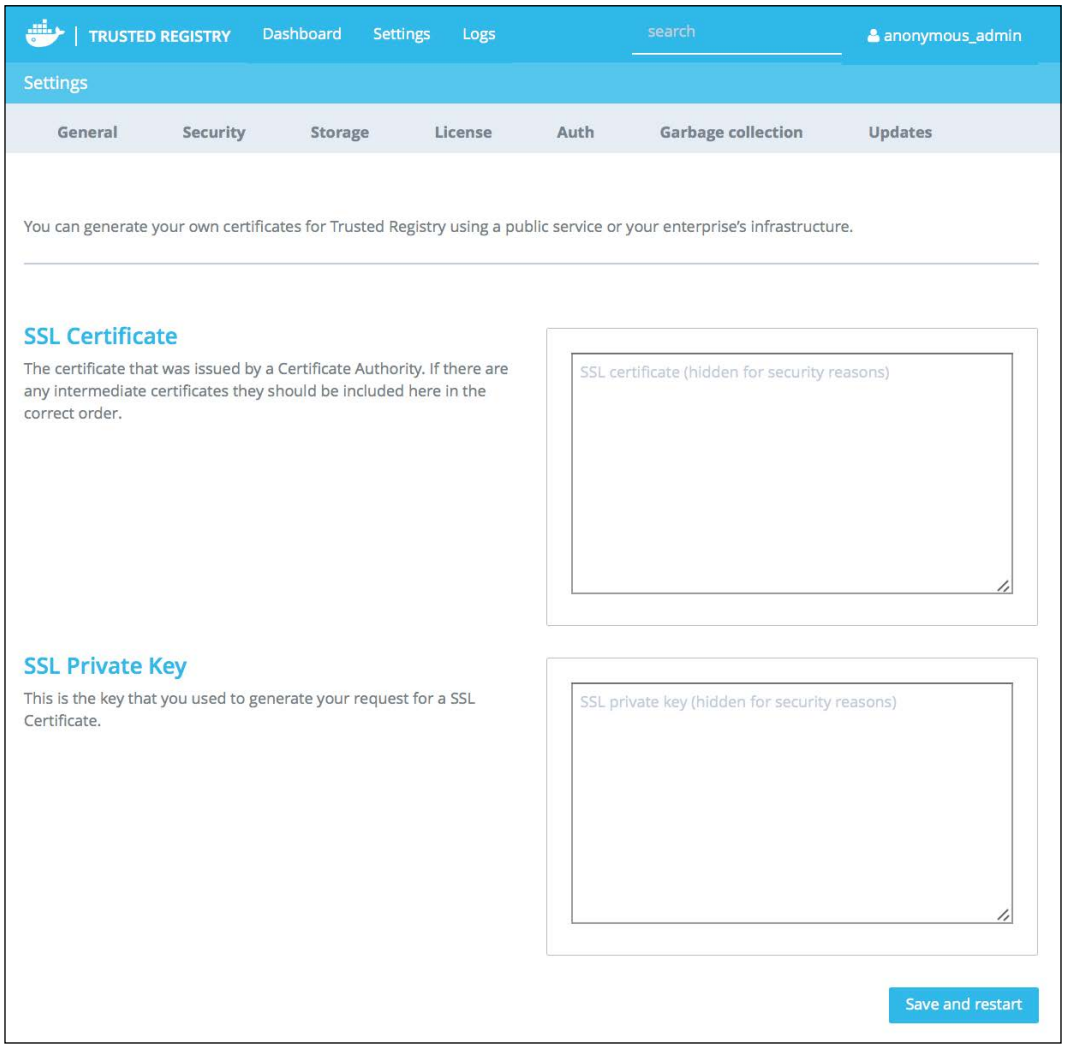
Now that we have our Trusted Registry set up, we need to make it secure. Before making it secure you will need to create an administrator account to be able to perform actions. Once you have your Trusted Registry up and running, and are logged into it, you will be able to see six areas under **Settings**. These are:

- **General** settings
- **Security** settings
- **Storage** settings
- **License**
- **Auth** settings
- **Updates**

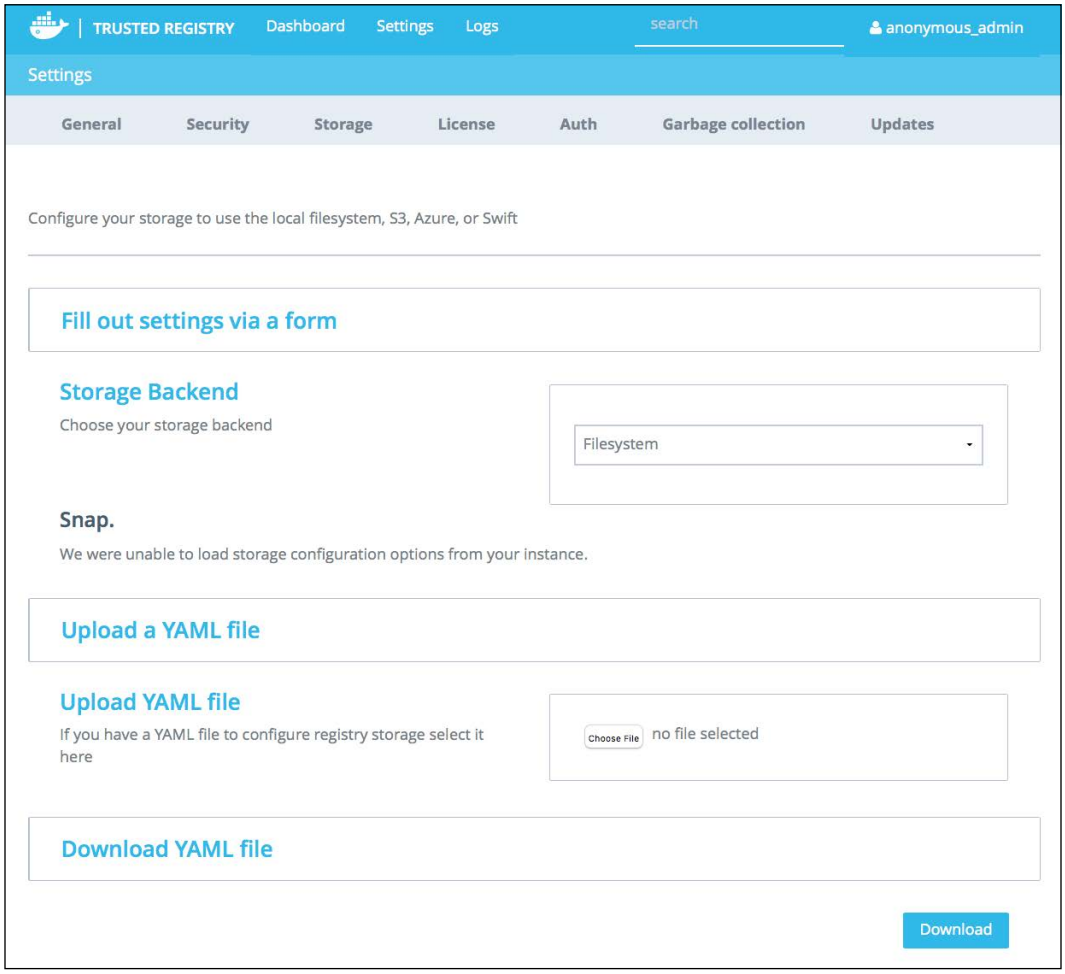
The **General** settings are mainly focused around settings such as **HTTP port** or **HTTPS port**, the **Domain name** to be used for your Trusted Registry, and proxy settings, if applicable.


TRUSTED REGISTRY
Dashboard
Settings
Logs
search
anonymous_admin
Settings
General
Security
Storage
License
Auth
Garbage collection
Updates
General settings to configure domains and ports. Note that the domain name is the only required field.
Domain name
Required. The fully qualified domain name assigned to the Trusted Registry host. Defaults to an empty string.
example.com
HTTP port
Used as the entry point for the image storage service. Defult: 80
80
HTTPS port
Used as the secure entry point for the image storage service. Default: 443
443
HTTP proxy
Proxy server for external HTTP requests.
HTTPS proxy
Proxy server for external HTTPS requests.
No proxy
Proxy bypass for HTTP/S requests.
Notary Server (experimental feature)
Notary server url. Note that for Notary signatures to show up in the Trusted Registry UI you must use the same domain name when pushing as the domain name configured in Trusted Registry. Ex. https://172.17.42.1:4443
Notary Verify TLS (experimental feature)
Whether or not to verify that the TLS certificate is valid for the Notary server. This is necessary for production environments.
Notary TLS Root CA (experimental feature)
The TLS certificate of the Certificate Authority used to verify Notary's certificate (if not already in operating system's CA store).
TLS certificate
Update checking
Disable outbound connections for update checks. If disabled you will not be notified when important updates are available.
Upgrades enabled
Save and restart

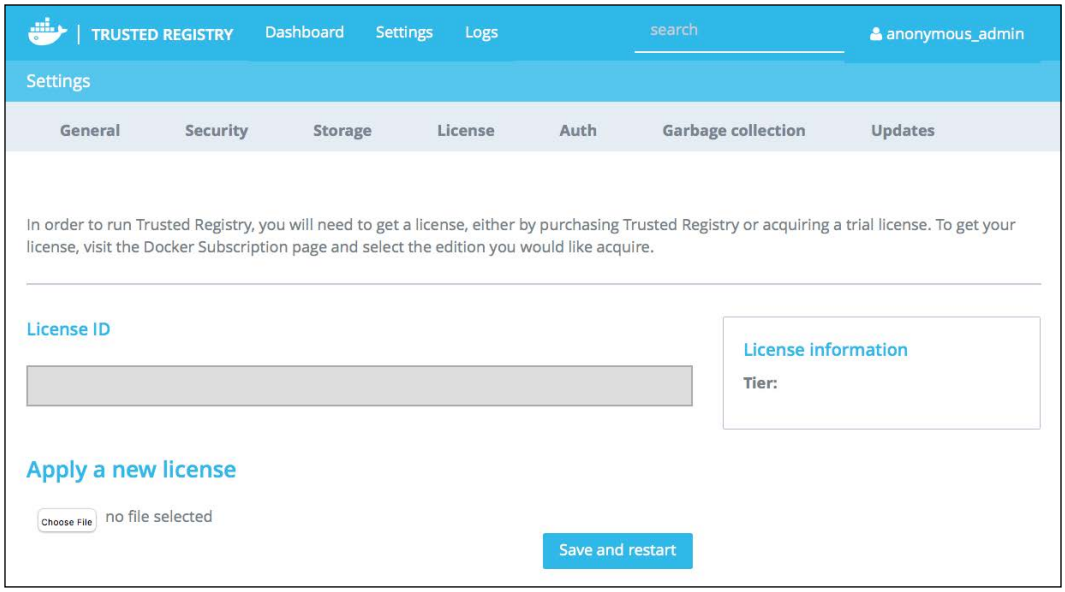
The next section, **Security** settings, is probably one of the most important ones. Within this **Dashboard** pane you are able to utilize your **SSL Certificate** and **SSL Private Key**. These are what make the communication between your Docker clients and the Trusted Registry secure. Now, there are a few options for those certificates. You can use the self signed ones that are created when installing the Trusted Registry. You can also do self signed ones of your own, using a command line tool such as **OpenSSL**. If you are in an enterprise organization, they more than likely have a location where you can request certificates such as the one that can be used with the registry. You will need to make sure that the certificates on your Trusted Registry are the same ones being used on your clients to ensure secure communications when doing `docker pull` or `docker push` commands:



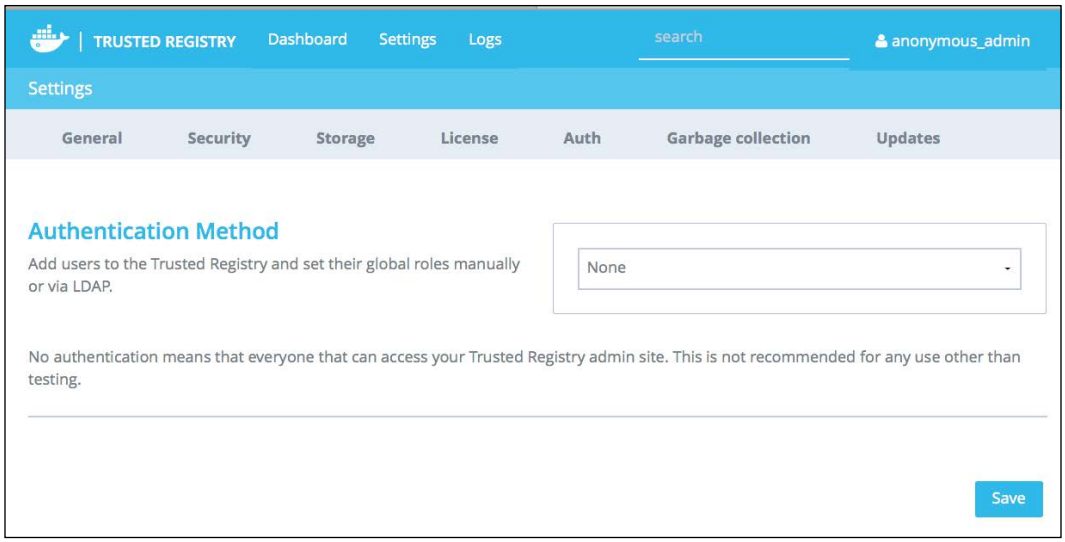
The next section deals with image storage settings. Within this **Dashboard** pane, you can set where your images are stored on the backend storage. Options for this might include an NFS share you are using, local disk storage of the Trusted Registry server, an S3 bucket from AWS, or another cloud storage solution. Once you have selected your **Storage Backend** option, you can then set the path from within that **Storage** to store your images:



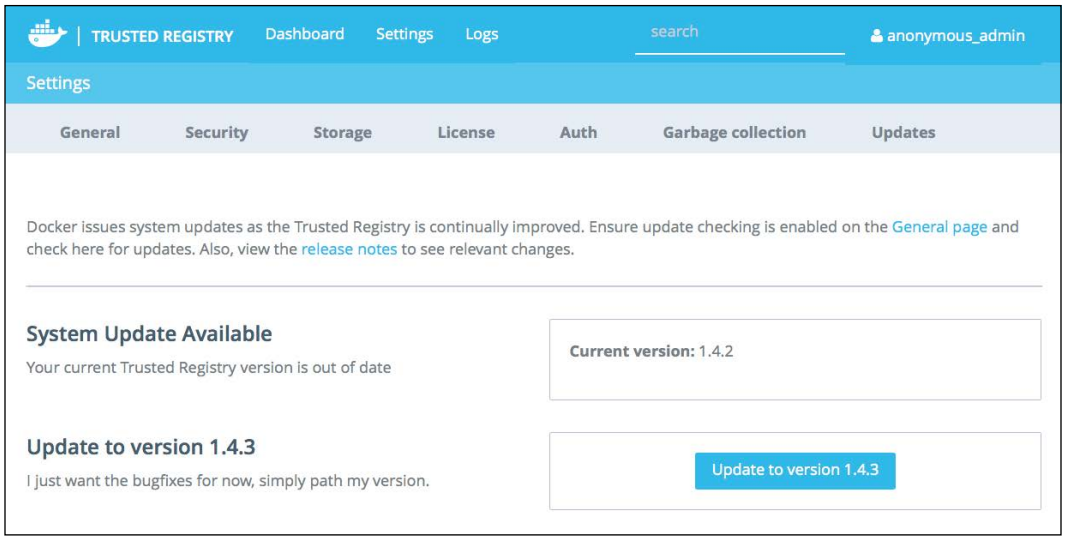
The **License** section is very straightforward as this is where you update your license when it's time to renew a new one or when you upgrade a license that might include more options:



Authentication settings allow you to tie the login to the Trusted Registry into your existing authentication environment. Your options here are: **None** or a **Managed** option. **None** is not recommended except for testing purposes. The **Managed** option is where you would set up usernames and passwords and manage them from there. The other option would be to use an **LDAP** service, one that you might already be running as well, so that users have the same login credentials for their other work appliances such as email or web logins.



The last section, **Updates**, deals with how you manage updates for the DTR. These settings would be totally up to you how you handle updates, but be sure if you are doing an automated form of updates that you are also utilizing backups for restoring purposes in the event that something goes wrong during the update process.



Administering

Now that we have covered the items that help you secure your Trusted Registry, we might as well take a few minutes to cover the other items that are within the console to help you administer it. Beyond the **Settings** tab within the registry, there are four other tabs that you can navigate and gather information about your registry. Those are:

- **Dashboard**
- **Repositories**
- **Organizations**
- **Logs**

The **Dashboard** is the main landing page you are taken to when you log in via your browser to the console. This will display information about your registry in one central location. The information you will be seeing is more hardware related information about the registry server itself as well as the Docker host that the registry server is running on. The **Repositories** section will allow you to control which repositories, either **Public** or **Private**, your users are able to pull images from. The **Organizations** section allows you to control access, that is, who on the system can push, pull, or do other Docker related commands against the repositories that you have elected to configure. The last section, the **Logs** section, will allow you to view logs based upon your containers that are being used from your registry. The logs are rotated every two weeks with a maximum size of *64 mb*. You are able to filter through the logs as well based on a container as well as being able to search for a date and/or time.

Workflow

In this section, let's pull an image, manipulate it, and then place it on our DTR for access by others within our organization.

First, we need to pull an image from the **Docker Hub**. Now, you could start from scratch with a **Dockerfile** and then do a Docker build and then push, but let's, for this demonstration, say we have the `mysql` image and we want to customize it in some way.

```
$ docker pull mysql


Using default tag: latest
```

```
latest: Pulling from library/mysql

1565e86129b8: Pull complete
a604b236bcde: Pull complete
2a1fefc8d587: Pull complete
f9519f46a2bf: Pull complete
b03fa53728a0: Pull complete
ac2f3cdeb1c6: Pull complete
b61ef27b0115: Pull complete
9ff29f750be3: Pull complete
ece4ebeae179: Pull complete
95255626f143: Pull complete
0c7947afc43f: Pull complete
b3a598670425: Pull complete
e287fa347325: Pull complete
40f595e5339f: Pull complete
0ab12a4dd3c8: Pull complete
89fa423a616b: Pull complete
Digest: sha256:72e383e001789562e943bee14728e3a93f2c3823182d14e3e01b3
fd877976265
Status: Downloaded newer image for mysql:latest

$ docker images

REPOSITORY          TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
mysql               latest              89fa423a616b        20 hours ago
359.9 MB
```

Now, let's assume we made a customization to the image. Let's say that we set up the container to ship its logs off to a log stash server that we are using to collect our logs from all our containers that we are running. We now need to save those changes.

```
$ docker commit be4ea9a7734e <dns.name>/mysql
```

When we go to do the commit, we need a few tidbits of information. The first is the container ID, which we can get from running a `docker ps` command. We also need the DNS name of our registry server that we set up earlier, and lastly a unique image name to give it. In our case, we will keep it as `mysql`.

We are now ready to push the updated image to our registry server. The only information we need is the image name that we want to push, which will be the `<dns.name>/mysql`.

```
$ docker push <dns.name>/mysql
```

The image is now ready to be used by the other users in our organization. Since the image is in our Trusted Registry, we can control access to that image from our clients. This could mean that our clients would need our certificate and keys to be able to push and pull this image, as well as permissions set up within the organization settings we previously went over in the last section.

```
$ docker pull <dns.name>/mysql
```

We can then make run the image, make changes if needed, and push the newly created image back to the Trusted Registry server as necessary.

Docker Registry

The Docker Registry is an open source option if you want to totally go at it on your own. If you totally want hands off, you can always use the Docker Hub and rely on public and private repositories, which will run you a fee on the Docker Hub though. This can be hosted locally on a server of your choosing or on a cloud service.

Installation

The installation of the Docker Registry is extremely simply as it runs in a Docker container. This allows you to run it virtually anywhere, on a virtual machine in your own server environment or in a cloud environment. The typical port that is used is port `5000`, but you can change it to suit your needs:

```
$ docker run -d -p 5000:5000 --restart=always  --name registry
registry:2.2
```

One of the other items you will notice from above is that we are specifying a version to use instead of leaving it blank and pulling the latest version. That is because as of writing this module, the latest version for that registry tag is still at version 0.9.1. Now, while this might be suitable for some, version 2 is stable enough to be considered and to run your production environment on. We are also introducing the `--restart=always` flag for that as in the event of something happening to the container, it will restart and be available to serve out or accept images.

Once you have run the command above, you will have a running container registry on the IP address of the Docker host you ran the command on along with the port selection that you used in your `docker run` command above.

Now it is time to put some images up on your new registry. The first thing we need is an image to push to the registry and we can do that in two ways. We can build images based on Docker files that we have created or we can pull down an image from another registry, in our case we will be using the Docker Hub, and then push that image to our new registry server. First, we need an image to choose and again, we will default back to the `mysql` image since it's a more popular image that most people might be using in their environments at some point along the way.

```
$ docker pull mysql
Using default tag: latest
latest: Pulling from library/mysql

1565e86129b8: Pull complete
a604b236bcde: Pull complete
2a1fefc8d587: Pull complete
f9519f46a2bf: Pull complete
b03fa53728a0: Pull complete
ac2f3cdeb1c6: Pull complete
b61ef27b0115: Pull complete
9ff29f750be3: Pull complete
ece4ebeae179: Pull complete
95255626f143: Pull complete
0c7947afc43f: Pull complete
b3a598670425: Pull complete
e287fa347325: Pull complete
40f595e5339f: Pull complete
0ab12a4dd3c8: Pull complete
89fa423a616b: Pull complete
```

```
Digest: sha256:72e383e001789562e943bee14728e3a93f2c3823182d14e3e01b3
fd877976265
Status: Downloaded newer image for mysql:latest
```

Next, you need to tag the image so it will now be pointing to your new registry so you can do push it to the new location:

```
$ docker tag mysql <IP_address>:5000/mysql
```

Let's break down that command above. What we are doing is applying the tag of `<IP_address>:5000/mysql` to the `mysql` image that we pulled from the Docker Hub. Now that `<IP_address>` piece will be replaced by the IP address from the Docker host that is running the registry container. This could also be a DNS name as well, as long as the DNS points to the correct IP that is running on the Docker host. We also need to specify the port number for our registry server, and in our case we left it with port `5000`, so we include: `5000` in the tag. Then, we are going to give it the same same of `mysql` at the end of the command. We are now ready to push this image to our new registry.

```
$ docker push <IP_address>:5000/mysql
```

After it has been pushed, you can now pull it down from another machine that is configured with Docker and has access to the registry server.

```
$ docker pull <IP_address>:5000/mysql
```

What we have looked at here are the defaults and while it could work if you want to use firewalls and such to secure the environment or even internal IP address, you still might want to take security to the next level and that is what we will look at in the next section. How can we make this even more secure?

Configuration and security

It's time to tighten up our running registry with some additional features. The first method would be to run your registry using TLS. Using TLS allows you to apply certificates to the system so that people who are pulling from it know that it is who you say it is by knowing that someone hasn't comprised the server or is doing a man in the middle attack by supplying you with compromised images.

To do that, we will need to rework the Docker `run` command we were running in the last section. This is going to assume you have gone through some of the process of obtaining a certificate and key from your enterprise environment or you have self signed one using another piece of software.

Our new command will look like this:

```
$ docker run -d -p 5000:5000 --restart=always --name registry \
  -e REGISTRY_HTTP_TLS_CERTIFICATE=server.crt \
  -e REGISTRY_HTTP_TLS_KEY=server.key \
  -v <certificate folder>/<path_on_container> \
  registry:2.2.0
```

You will need to be in the directory where the certificates are or specify the full path to them in the above command. Again, we are keeping the standard port of `5000`, along with the name of registry. You could alter that too to something that might suit you better. For the sake of this module we will keep it close to that in the official documentation in the event that you look there for more reference. Next, we add two additional lines to the `run` command:

```
  -e REGISTRY_HTTP_TLS_CERTIFICATE=server.crt \
  -e REGISTRY_HTTP_TLS_KEY=server.key \
```

This will allow you to specify the certificate and key file that you will be using. These two files will need to be in the same directory that you are running the run command from as the environmental variables will be looking for them upon run. Now you could also add a volume switch to the run command to make it a little cleaner if you like and put the certificate and key in that folder and run the registry server that way.

The other way you can help with security is by putting a username and password on the registry server. This will help when users want to push or pull an item as they will need the username and password information. The catch with this is that you have to be using TLS in conjunction with this method. This method of username and password is not a standalone option.

First, you need to create a password file that you will be using in your `run` command:

```
$ docker run --entrypoint htpasswd registry:2.2.0 -bn <username>
<password> > htpasswd
```

Now, it can be a little confusing to understand what is happening here, so let's clear that up before we jump to the `run` command. First, we are issuing a `run` command. This command is going to run the `registry:2.2.0` container and its entry point specified means to run the `htpasswd` command along with the `-bn` switches, which will inject the `username` and `password` in an encrypted fashion into a file called `htpasswd` that you will be using for authentication purposes on the registry server. The `-b` means to run in batch mode while the `-n` means to display the results, and the `>` means to put those items into a file instead of to the actual output screen.

Now, onto our newly enhanced and totally secure Docker `run` command for our registry:

```
$ docker run -d -p 5000:5000 --restart=always --name registry \
  -e "REGISTRY_AUTH=htpasswd" \
  -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Name" \
  -e REGISTRY_AUTH_HTPASSWD_PATH=htpasswd \
  -e REGISTRY_HTTP_TLS_CERTIFICATE=server.crt \
  -e REGISTRY_HTTP_TLS_KEY=server.key \
  registry:2.20
```

Again, it's a lot to digest but let's walk through it. We have seen some of these lines before in:

```
  -e REGISTRY_HTTP_TLS_CERTIFICATE=server.crt \
  -e REGISTRY_HTTP_TLS_KEY=server.key \
```

The new ones are:

```
  -e "REGISTRY_AUTH=htpasswd" \
  -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Name" \
  -e REGISTRY_AUTH_HTPASSWD_PATH=htpasswd \
```

The first one tells the registry server to use `htpasswd` as its authentication method to verify clients. The second gives your registry a name and can be changed at your own discretion. The last one tells the registry server the location of the file that is to be used for the `htpasswd` authentication. Again, you will need to use volumes and put the `htpasswd` file in its own volume inside the container so it allows for easier updating down the road. You also need to remember the `htpasswd` file needs to be placed in the same directory as the certificate and key file when you execute the Docker `run` command.

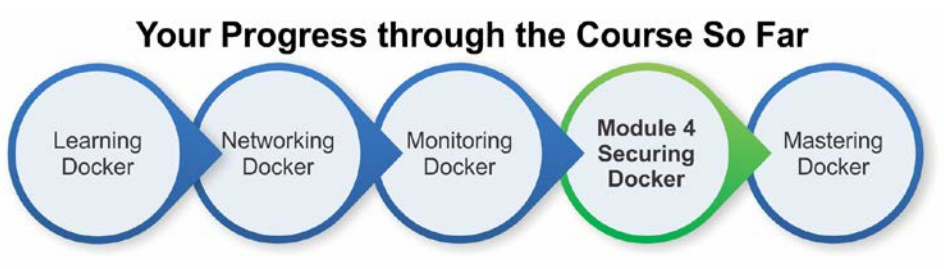
Summary of Module 4 Chapter 2

In this chapter, we have looked at being able to sign your images using the components of Docker Content Trust as well as hardware signing using Docker Content Trust along with the third party utilities in the form of YubiKeys. We also took a look at Docker Subscription that you can utilize to your advantage to help set up not only secure Docker environments but also ones that are supported by those at Docker itself. We then looked at DTR as a solution that you can use to store your Docker images. Lastly, we looked at the Docker Registry, which is a self hosted registry that you can use to store and manage your images. This chapter should help give you enough configuration items to chew on to help you make the right decision as to where to store your images.

In the next chapter we will be looking at securing/hardening Linux kernels. As the kernel is what is used to run all your containers, it is important that it is secured in the proper way to help alleviate any security related issues. We will be covering some hardening guides that you can use to accomplish this goal.

Ankita Thakur

Your Course Guide



3
Securing and Hardening Linux Kernels

In this chapter, we will turn our attention to securing and hardening the one key piece that every container running on your Docker host relies on: the Linux kernel. We will focus on two topics: guides that you can follow to harden the Linux kernel and tools that you can add to your arsenal to help harden the Linux kernel. Let's take a brief look at what we will be covering in this chapter before diving in:

- Linux kernel hardening guides
- Linux kernel hardening tools
    - **Grsecurity**
    - **Lynis**

Linux kernel hardening guides

In this section, we will be looking at the SANS Institute hardening guide for the Linux kernel. While a lot of this information is outdated, I believe that it is important for you to understand how the Linux kernel has evolved and become a secure entity. If you were to step into a time machine and go back to the year 2003 and attempt to do the things you want to do today, this is everything you would have to do.

First, some background information about the SANS Institute. It is a private US-based company that specializes in cybersecurity and information technology-related training and education. These trainings prepare professionals to defend their environments against attackers. SANS also offers a variety of free security-related content via their SANS Technology Institute Leadership Lab. More information about this can be found at `http://www.sans.edu/research/leadership-laboratory`.

To help alleviate against this widespread attack base, there needs to be security focus on every aspect of your IT infrastructure and software. Based upon this, the first place to start would be at the Linux kernel.

SANS hardening guide deep dive

As we have already covered the background of the SANS Institute, let's go ahead and jump into the guide that we will be following to secure our Linux kernel(s).

For reference, we will be using the following URL and highlighting the sticking points that you should be focusing on and implementing in your environments to secure the Linux kernel:

```
https://www.sans.org/reading-room/whitepapers/linux/linux-kernel-hardening-1294
```

The Linux kernel is an always-developing and maturing piece of the Linux ecosystem and for this reason, it's important to get a firm grasp on the Linux kernel as it stands currently, which will help when looking to lockdown the new feature sets that might come in future releases.

The Linux kernel allows loading modules without having to recompile or reboot, which is great when you are looking to eliminate downtime. Some various operating systems require reboots when trying to apply updates to a certain operating system/application criteria. This can also be a bad thing with regards to the Linux kernel as the attackers can inject harmful material into the kernel and wouldn't need to reboot the machine, which might be caught by someone noticing the reboot of the system. For this reason, it is suggested that a statically compiled kernel with the load option be disabled to help prevent against attack vectors.

Buffer overflows are another way attackers can compromise a kernel and gain entry. Applications have a limit, or buffer, on how much a user can store in memory. An attacker overflows this buffer with specially crafted code, which could let the attacker gain control of the system that, in turn, will empower them to do whatever they want at that point. They could add backdoors to the system, send logs off to a nefarious place, add additional users to the system, or even lock you out of the system. To prevent these type of attacks, there are three areas of focus that the guide hones in on.

The first is the **Openwall** Linux kernel patch that was a patch created to address this issue. This patch also includes some other security enhancements that might be attributed to your running environments. Some of these items included restricted links and file reads/writes in the `/tmp` folder location and restricted access to the `/proc` locations on the filesystem. It also includes enhanced enforcement for a number of user processes that you could control as well as the ability to destroy shared memory segments, which were not in use, and lastly, some other enhancements for those of you that are running kernel versions older than version 2.4.

If you are running an older version of the Linux kernel, you will want to check out the Openwall hardened Linux at `http://www.openwall.com/Owl/` and Openwall Linux at `http://www.openwall.com/linux/`.

The next piece of software is called **Exec Shield** and it takes a similar approach to the Openwall Linux kernel patch, which implements a non-executable stack, but Exec Shield extends this by attempting to protect any and all segments of virtual memory. This patch is limited to the prevention of attacks against the Linux kernel address space. These address spaces include stack, buffer, or function pointer overflow spaces.

More information about this patch can be found at `https://en.wikipedia.org/wiki/Exec_Shield`.

The last one is **PaX**, which is a team that creates a patch for the Linux kernel to prevent against a variety of software vulnerabilities. As this is something we will be talking about in-depth in the next section, we will just discuss some of its features. This patch focuses on the following three address spaces:

- **PAGEEXEC**: These are paging-based, non-executable pages
- **SEGMEXEC**: These are segmentation–based, non-executable pages
- **MPROTECT**: These are `mmap()` and `mprotect()` restrictions

To learn more about PaX, visit `https://pax.grsecurity.net`.

Now that you have seen how much efforts you had to put in, you should be glad that security is now at the forefront for everyone, especially, the Linux kernel. In some of the later chapters, we will be looking at some of the following new technologies that are being used to help secure environments:

- Namespaces
- cgroups
- sVirt
- Summon

There are also a lot of capabilities that can be accomplished through the `--cap-ad and --cap-drop` switches on your `docker run` command.

Even like the days before, you still need to be aware of the fact that the kernel is shared throughout all your containers on a host, therefore, you need to protect this kernel and watch out for vulnerabilities when necessary. The following link allows you to view **Common Vulnerabilities and Exposures** (**CVE**) in the Linux kernel:

`https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/cvssscoremin-7/cvssscoremax-7.99/Linux-Linux-Kernel.html`

Access controls

There are various levels of access controls that you can layer on top of Linux as well as recommendations that you should follow with reference to certain users, and these would be the superusers on your system. Just to give some definition to superusers, they are the accounts on the system that have unfettered access to do anything and everything. You should include the root user when you are layering on these access controls.

These access control recommendations will be the following:

- Restricting usage of the root user
- Restricting its ability to SSH

  By default, on some systems, root has the ability to SSH to machine if SSH is enabled, which we can see from a portion of the `/etc/ssh/sshd_config` file on some Linux systems, as follows:

  ```
  # Authentication:

  #LoginGraceTime 2m
  #PermitRootLogin no
  #StrictModes yes
  #MaxAuthTries 6
  #MaxSessions 10
  ```

  From what you can see here, the section for `PermitRootLogin no` is commented out with the `#` symbol so that means this line won't be interpreted. To change this, simply remove the `#` symbol and save the file and restart the service. The section of this file should now be similar to the following code:

  ```
  # Authentication:

  #LoginGraceTime 2m
  ```

```
PermitRootLogin no
#StrictModes yes
#MaxAuthTries 6
#MaxSessions 10
```

Now, you may want to restart the SSH service for these changes to take affect, as follows:

```
$ sudo service sshd restart
```

- Restrict its ability to log in beyond the console. On most Linux systems, there is a file in `/etc/default/login` and in that file, there is a line that is similar to the following:

  ```
  #CONSOLE=/dev/console
  ```

  Similar to the preceding example, we need to uncomment this line by removing `#` for this to take affect. This will only allow the root user to log in at `console` and not via SSH or other methods.

- Restrict `su` command

  The `su` commands allow you to login as the root user and be able to issue root-level commands, which gives you full access to the entire system. To restrict access to who can use this command, there is a file located at `/etc/pam.d/su`, and in this file, you will see a line similar to the following:

  ```
  auth required /lib/security/pam_wheel.so use_uid
  ```

  You can also choose the following line of code here, depending upon your Linux flavor:

  ```
  auth required pam_wheel.so use_uid
  ```

  The check for wheel membership will be done against the current user ID for the ability to use the `su` command.

- Requiring `sudo` to run commands
- Some other access controls that are remanded are the use of the following controls:
  - **Mandatory Access Controls** (**MAC**): Restricting what users can do on systems
  - **Role-Based Access Controls**: Using groups to assign the roles that these groups can perform
  - **Rule Set Based Access Controls** (**RSBAC**): Rule sets that are grouped in the request type and performs actions based on set rule(s)

- **Domain and Type Enforcement** (**DTE**): Allow or restrict certain domains from performing set actions or preventing domains from interacting with each other

You can also utilize the following:

- SELinux (RPM-based systems (such as Red Hat, CentOS, and Fedora)
- AppArmor (apt-get-based systems (such as Ubuntu and Debian)

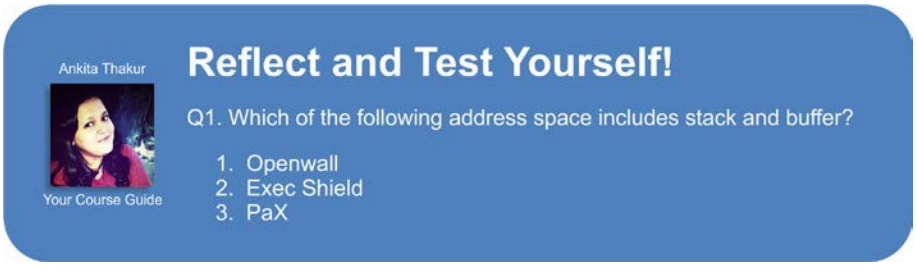
These RSBAC, as we discussed earlier, allow you to choose methods of control that are appropriate for what your system is running. You can also create your own access control modules that can help enforce. By default, on most Linux systems, these type of environments are enabled or in enforcing mode. Majority of people will turn these off when they create a new system, but it comes with security drawbacks, therefore, it's important to learn how these systems work and use them in the enabled or enforcement mode to help mitigate further risks.

More information about each can be found at the following:

- **SELinux**: `https://en.wikipedia.org/wiki/Security-Enhanced_Linux`
- **AppArmor**: `https://en.wikipedia.org/wiki/AppArmor`

Distribution focused

There are many Linux distributions, or flavors as they call them, in the Linux community that have been *pre-baked* to be already hardened. We referenced one earlier, the **Owlwall** flavor of Linux, but there are others out there as well. Out of the other two, one that is no longer around is **Adamantix** and the other is **Gentoo Linux**. These Linux flavors feature some baked-in Linux kernel hardening as standards of their operating system builds.



Linux kernel hardening tools

There are some Linux kernel hardening tools out there, but we will focus on only two of them in this section. The first being Grsecurity and the second being Lynis. These are tools that you can add to your arsenal to help in increasing the security of the environments that you will be running your Docker containers on.

Grsecurity

So, what exactly is Grsecurity? According to their website, Grsecurity is an extensive security enhancement for the Linux kernel. This enhancement contains a wide range of items that help in defending against various threats. These threats might include the following components:

- **Zero day exploits**: This mitigates and keeps your environment protected until a long-term solution can be made available through the vendor.
- **Shared host or container weaknesses**: This protects you against kernel compromises that various technologies, and very much so containers, use for each container on the host.
- **It goes beyond basic access controls**: Grsecurity works with the PaX team to introduce complexity and unpredictability to the attacker, while responding and denying the attacker any more chances.
- **Integrates with you existing Linux distribution:** As Grsecurity is kernel-based, it can be used with any Linux flavors such as Red Hat, Ubuntu, Debian, and Gentoo. Whatever your Linux flavor is, it doesn't matter, as the focus is on the underlying Linux kernel.

More information can be found at `https://grsecurity.net/`.

To directly get to the good stuff and see the feature set that is offered by utilizing a tool like Grsecurity, you will want to go to the following link:

```
http://grsecurity.net/features.php
```

On this page, items will be grouped into the following five categories:

- Memory Corruption Defenses
- Filesystem Hardening
- Miscellaneous Protections
- RBAC
- GCC Plugins

Lynis

Lynis is an open source tool that is used to audit your systems for security. It runs directly on the host so that it has access to the Linux kernel itself, as well as various other items. Lynis runs on almost every Unix operating system including the following:

- AIS
- FreeBSD
- Mac OS
- Linux
- Solaris

Lynis was written as a shell script, therefore, it's just as easy as copying and pasting on your system and running a simple command:

```
./lynis audit system
```

While it is running, the following actions are being taken:

- Determining the OS
- Performing a search for available tools and utilities
- Checking for any Lynis update
- Running tests from enabled plugins
- Running security tests per category
- Reporting status of security scan

More information can be found at `https://rootkit.nl/projects/lynis.html` and `https://cisofy.com/lynis/`.

Your Coding Challenge

Ankita Thakur

Your Course Guide

Let's now test what we've learned so far:

- Which are the Linux kernel hardening tools that we explored in this chapter?
- What does the following access controls do?
  - Mandatory Access Controls
  - Role-Based Access Controls
  - Rule Set Based Access Controls
  - Domain and Type Enforcement

Summary of Module 4 Chapter 3

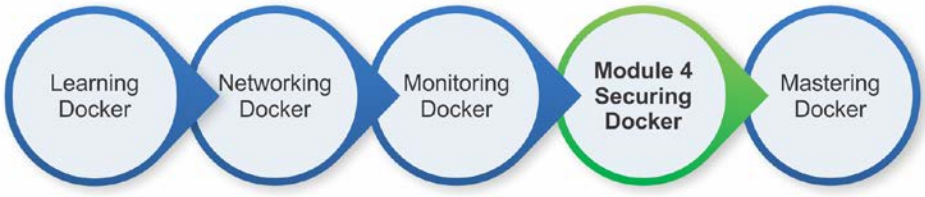
Ankita Thakur

Your Course Guide

In this chapter, we took a look at hardening and securing Linux kernels. We first looked at some hardening guides followed by a deep dive of an overview of the SANS Institute Hardening Guide. We also took a look at how to prevent buffer overflows in our kernels and applications through various patches. We also looked at various access controls, SELinux, and AppArmor. Lastly, we took a look at two hardening tools that we can add to our toolbox of software in the form of Grsecurity and Lynis.

In the next chapter, we will take a look at the Docker Bench application for security. This is an application that can look at the various Docker items, such as host configuration, Docker daemon configuration, daemon configuration files, container images and build files, container runtime, and lastly, Docker security operations. It will contain hands-on examples with a lot of code outputs.

Your Progress through the Course So Far



4
Docker Bench for Security

In this chapter, we will be looking at the **Docker Bench for Security**. This is a tool that can be utilized to scan your Docker environments, start the host level and inspect all the aspects of this host, inspect the Docker daemon and its configuration, inspect the containers running on the Docker host, and review the Docker security operations and give you recommendations across the board of a threat or concern that you might want to look at in order to address it. In this chapter, we will be looking at the following items:

- **Docker security** – best practices
- **Docker** – best practices
- **Center for Internet Security** (**CIS**) guide
    - Host configuration
    - Docker daemon configuration
    - Docker daemon configuration files
    - Container images/runtime
    - Docker security operations
- The Docker Bench Security application
    - Running the tool
    - Understanding the output

Docker security – best practices

In this section, we will take a look at the best practices when it comes to Docker as well as the CIS 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 of what needs to or should be fixed. The guide is broken down into the following sections:

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

Docker – best practices

Before we dive into the CIS guide, let's go over some of the following best practices when using Docker:

- **One application per container**: Spread your applications to one per container. Docker was built for this and it makes everything easy at the end of the day. The isolation that we talked about earlier is where this is the key.
- **Review who has access to your Docker hosts**: Remember that whoever has the access to your Docker hosts has the 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 the 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 **Internet Relay Chat** (**IRC**) chat rooms to your advantage.

CIS guide

The CIS guide is a document (`https://benchmarks.cisecurity.org/tools2/docker/cis_docker_1.6_benchmark_v1.0.0.pdf`) that goes over the aspects of the Docker pieces to help you securely configure the various pieces of your Docker environment. We will cover these in the following sections.

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 the attackers.

Docker daemon configuration

This part of the guide recommends securing 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 that we saw previously and items you will see in the following section when we run through the tool.

Docker daemon configuration files

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

Container images/runtime

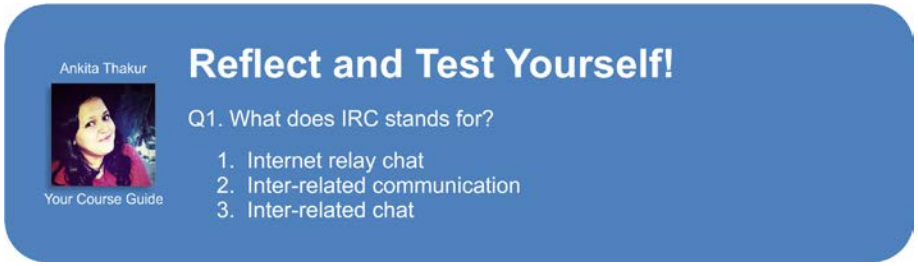
This part of the guide contains both the information for securing the container images as well as the container runtime.

The first part contains images, cover base images, and build files that were used. You need to be sure about the images you are using not only for your base images, but also 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 of the runtime variables that 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 also the Docker host and other containers running on this host.

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 recommendations of items that are 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 following components:

- The host configuration
- The Docker daemon configuration
- The Docker daemon configuration files
- Container images and build files
- Container runtime
- 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 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 that 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 \
docker/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, as shown in the following command:

```
# ------------------------------------------------------------------------
-------
# 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 Sun Jan 17 19:18:56 UTC 2016
```

Running the tool – host configuration

Let's take a look at the output of the host configuration runtime:

```
[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]       * Using 1.9.1 which is current as of 2015-11-09
[INFO]       * Check with your operating system vendor for support and
security maintenance for docker
[INFO] 1.7  - Only allow trusted users to control Docker daemon
[INFO]      * docker:x:100:docker
[WARN] 1.8  - Failed to inspect: auditctl command not found.
[INFO] 1.9  - Audit Docker files and directories - /var/lib/docker
[INFO]      * Directory 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
[INFO] 1.18 - Audit Docker files and directories - /etc/default/docker
[INFO]       * File not found
```

Running the tool – Docker daemon configuration

Let's take a look at the output for the Docker daemon configuration runtime, as shown in the following command:

```
[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
```

Running the tool – Docker daemon configuration files

Let's take a look at the output for the Docker daemon configuration files runtime, as follows:

```
[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
[PASS] 3.25 - Verify that Docker socket file ownership is set to
root:docker
[PASS] 3.26 - Verify that Docker socket file permissions are set to 660
```

Running the tool – container images and build files

Let's take a look at the output for the container images and build files runtime, as shown in the following command:

```
[INFO] 4 - Container Images and Build Files
[INFO] 4.1  - Create a user for the container
[INFO]      * No containers running
```

Running the tool – container runtime

Let's take a look at the output for the container runtime, as follows:

```
[INFO] 5  - Container Runtime
[INFO]      * No containers running, skipping Section 5
```

Running the tool – Docker security operations

Let's take a look at the output for the Docker security operations runtime, as shown in the following command:

```
[INFO] 6  - Docker Security Operations
[INFO] 6.5 - Use a centralized and remote log collection service
[INFO]      * No containers running
[INFO] 6.6 - Avoid image sprawl
[INFO]      * There are currently: 23 images
[WARN] 6.7 - Avoid container sprawl
[WARN]      * There are currently a total of 51 containers, with only 1
of them currently running
```

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

Understanding the output

There are three types of output that we will see, as follows:

- `[PASS]`: These items are solid and good to go. They don't need any attention, but they 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 that 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, as shown in the following:

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

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 again focus on the `[WARN]` items in each section. Other warnings may come up when you run yours, but these will be the ones that come up the most, if not for everyone at first.

Understanding the output – host configuration

Let's take a look at the following output for the host configuration runtime output:

```
[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.
```

Understanding the output – the Docker daemon configuration

Let's take a look at the following output for the Docker daemon configuration output:

```
[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 daemon start up process that will prevent Docker from using the `aufs` storage driver. 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 that you are using.

Understanding the output – 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 a few warnings here. This is one area where you should hope to never see any warnings.

Understanding the output – container images and build files

Let's take a look at the following output for the container images and build files runtime output:

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

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

Understanding the output – container runtime

Let's take a look at the output for the container runtime output, as follows:

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

This states that the `suspicious_mccarthy` container 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 that you are only running one process per container. If you are running more than one process, you will want to spread them out across multiple containers and use container linking, as shown in the following command:

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

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

```
[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
```

This 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. You can utilize the `docker exec` command to execute the items against your containers (see more information here: `https://docs.docker.com/engine/reference/commandline/exec/`), as shown in the following command:

```
[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 started, the `--net=host` switch was passed along. It is not recommended to use this as it allows the container to modify the network configuration and open low port numbers as well as access networking services on the Docker host, as follows:

```
[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), as shown in the following command:

```
[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 `--cpu-shares` switch while issuing your `docker run` command. The CPU share is based off of the number 1,024. Therefore, half would be 512 and 25% would be 256. Use 1,024 as the base number to determine the CPU share, as follows:

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

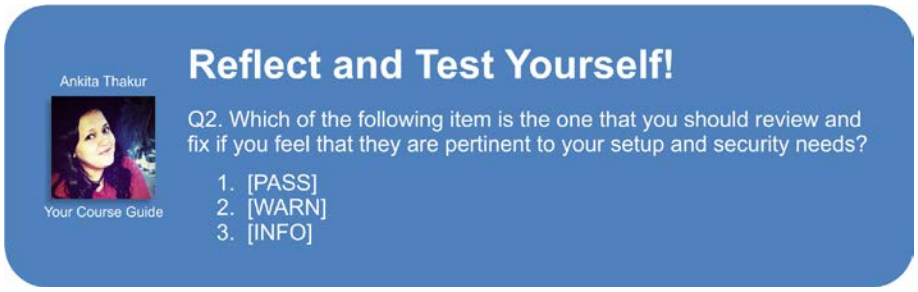
You really want to be using your containers as immutable environments, meaning that they don't write any data inside them. Data should be written out to volumes. Again, you can use the `--read-only` switch, as follows:

```
[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.

Understanding the output – Docker security operations

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



Your Coding Challenge

Ankita Thakur

Your Course Guide

Here are some questions for you to know whether you have understood the concepts:

- What are the best practices when using Docker?
- What is a CIS guide?
- Which components are inspected by the tool named Docker Bench Security application?

Summary of Module 4 Chapter 4

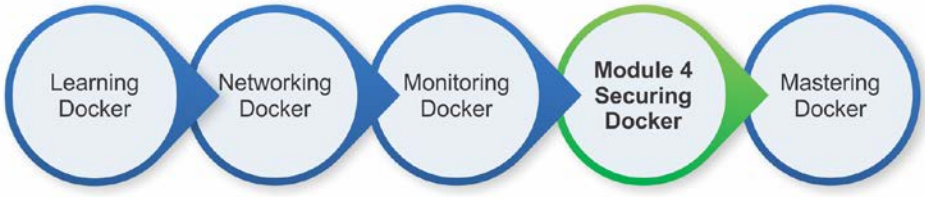
Ankita Thakur

Your Course Guide

In this chapter, we took a look at the CIS guidelines for Docker. This guide will assist you in setting up multiple aspects of your Docker environment. Lastly, we looked at the Docker Bench for Security. We looked at how to get it up and running and went through an example of what the output would look like once it has been run. We then took a look at the output to see what all it meant. Remember the six items that the application covered: 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 how to monitor as well as report any Docker security issues that you come across. This will help you know where to look for anything related to the security that may pertain to your existing environment. If you are to come across security-related issues that you find yourself, there are best practices for reporting these issues to give time to Docker to fix them before allowing the public community time to know about the issue, which will allow the hackers to use these vulnerabilities to their advantage.

Your Progress through the Course So Far



5
Monitoring and Reporting Docker Security Incidents

In this chapter, we will take a look at how to stay on top of the items that Docker has released, regarding the security findings in order to be aware of your environments. Also, we will take a look at how to safely report any security findings that you come across in order to ensure that Docker has a chance to alleviate the concern before it becomes public and widespread. In this chapter, we will be covering the following topics:

- Docker security monitoring
- Docker **Common Vulnerabilities and Exposures** (**CVE**)
- Mailing lists
- Docker security reporting
    - Responsible disclosure
    - Security reporting
- Additional Docker resources
    - Docker Notary
    - Hardware signing
    - Reading materials

Docker security monitoring

In this section, we will take a look at some ways of monitoring security issues that relate to any Docker products you may be using. While you are using the various products, you need to be able to be aware of, if any, security issues that arise so that you can mitigate these risks to keep your environments and data safe.

Docker CVE

To understand what a Docker CVE is, you need to first know what is CVE. CVEs are actually a system that is maintained by the MITRE Corporation. These are used as a public way of providing information based on a CVE number that is dedicated to each vulnerability for easy reference. This allows a national database of all the vulnerabilities that are given a CVE number from the MITRE Corporation. To learn more about CVEs, you can find it on the Wikipedia article here:

`https://en.wikipedia.org/wiki/Common_Vulnerabilities_and_Exposures`

The Wikipedia article explains things such as how they go about giving CVE numbers and the format that they all follow.

Now that you know what CVEs are, you probably have already pieced together what Docker CVEs are. They are CVEs that are directly related to Docker security incidents or issues. To learn more about Docker CVEs or see a list of current Docker CVEs, visit `https://www.docker.com/docker-cve-database`.

This listing will be updated anytime a CVE is created for a Docker product. As you can see, the list is very small, therefore, this is probably a list that will not grow on a day-to-day, or even a month-to-month, basis frequency.

Mailing lists

Another method for following or discussing security-related issues of any Docker products in the ecosystem is to join their mailing lists. Currently, they have two mailing lists that you can either join or follow along with.

The first is a developer list that you can join or follow along with. This is a list for those who are either helping in assisting with contributing the code to the Docker products or developing products using the Docker code base provided in the following:

`https://groups.google.com/forum/#!forum/docker-dev`

The second list is a user list. This list is for those who, you guessed it, are the users of the various Docker products that might have security-related questions. You can search from the already submitted discussions, join existing conversations, or ask new questions that will be answered by those who are also on the mailing lists at the following forum:

`https://groups.google.com/forum/#!forum/docker-user`

Before asking some security-related questions, you will want to read the following section to ensure that you are not exposing any existing security issues that might tempt an attacker out there.

Docker security reporting

Reporting Docker security issues is just as important as monitoring security issues with regards to Docker. While it is important to report these issues, there are certain standards that you should follow when you find security issues and are going to, hopefully, report them.

Responsible disclosure

When disclosing security-related issues, not only for Docker, but for any product out there, there is a term called **responsible disclosure** that everyone should follow. Responsible disclosure is an agreement that allows the developer or maintainer of the product ample time to provide a fix for the security issue before disclosing the issue to the general public.

To learn more about responsible disclosure, you can visit `https://en.wikipedia.org/wiki/Responsible_disclosure`.

Remember to put yourself in the shoes of the group that is responsible for the code. If it were your code, wouldn't you want someone to give you a notice of a vulnerability so that you had ample time to fix the issue before it was disclosed, causing widespread panic and flooding the inbox with e-mails from the masses.

Security reporting

Currently, the method for reporting security issues is to e-mail the Docker security team and give them as much information as you can provide about the security issue. While these are not the exact items that Docker might recommend, there are general guidelines that most other security professionals like to see when reporting security issues, such as the following:

- Product and version, where the security issue was discovered
- Method to reproduce the issue
- Operating system that was being used at the time, plus the version
- Any additional information you can provide

Remember, the more information you provide from the beginning, the quicker the team has to react from their end by being on top of the issue and attack it more aggressively from the start.

To report a security issue for any Docker-related product, make sure to e-mail any information to `security@docker.com`

Additional Docker security resources

If you are looking for some other items to look into, there are some additional items that we have covered in *Chapter 1*, *Securing Docker Hosts* that are worthwhile to conduct a quick review. Make sure to look back at *Chapter 1*, *Securing Docker Hosts* to get more details on the next couple of items or links that will be provided in each section.

Docker Notary

Let's take a quick look at **Docker Notary**, but for more information about Docker Notary, you can look back at the information in *Chapter 2*, *Securing Docker Components* or the following URL:

```
https://github.com/docker/notary
```

Docker Notary allows you to publish your content by signing it with private keys that you are recommended to keep offline. Using these keys to sign your content helps in ensuring others to know that the content they are using is, in fact, from who it says it is—you—and that the content can be trusted, assuming the users trust you.

Docker Notary has a few key goals that I believe are important to point out in the following:

- Survivable key compromise
- Freshness guarantee
- Configurable trust thresholds
- Signing delegation
- Use of existing distribution
- Untrusted mirrors and transport

It is important to know that Docker Notary has a server and client component as well. To use Notary, you will have to be familiar with the command-line environment. The preceding link will break it down for you and give you walkthroughs on setting up and using each component.

Hardware signing

Similar to the previous *Docker Notary* section, let's take a quick look at the hardware signing as it's a very important feature that must be understood fully.

Docker also allows hardware signing. What does this mean? From the previous section, we saw that you can use highly secure keys to sign your content, allowing others to verify that the information is from who it says it is, which ultimately provides everyone great peace of mind.

Hardware signing takes this to a whole new level by allowing you to add yet another layer of code signing. By introducing a hardware device, Yubikey—a USB piece of hardware—you can use your private keys (remember to keep them secure and offline somewhere) as well as a piece of hardware that requires you to tap it when you sign your code. This proves that you are a human by the fact of having to physically touch the YubiKey when you are signing your code.

For more information about the hardware signing part of Notary, it is worthwhile to read their announcement when they released this feature from the following URL:

`https://blog.docker.com/2015/11/docker-content-trust-yubikey/`

For a video demonstration of using **YubiKeys** and Docker Notary, please visit the following YouTube URL:

```
https://youtu.be/fLfFFtOHRZQ?t=1h21m23s
```

To find out more information about YubiKeys, visit their website at the following URL:

```
https://www.yubico.com
```

Reading materials

There are also some additional reading materials that can assist with ensuring your focus is on monitoring the security aspect of the entire Docker ecosystem.

Looking back at *Chapter 4*, *Docker Bench for Security*, we covered the Docker Bench, which is a scanning application for your entire Docker environment. This is highly useful to help in pointing out any security risks that you might have.

There is also a great free Docker security eBook that I found. This module will cover potential security issues along with tools and techniques that you can utilize to secure your container environments. Not bad for free, right?! You can find this module at the following URL:

```
https://www.openshift.com/promotions/docker-security.html
```

You can refer to the following *Introduction to Container Security* whitepaper for more information:

```
https://d3oypxn00j2a10.cloudfront.net/assets/img/Docker%20Security/
WP_Intro_to_container_security_03.20.2015.pdf
```

You can also refer to *The Definitive Guide To Docker Containers* whitepaper here:

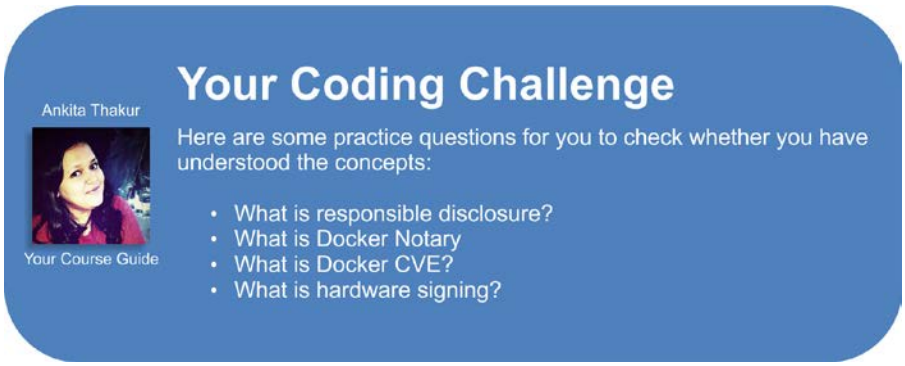
```
https://www.docker.com/sites/default/files/WP-%20Definitive%20
Guide%20To%20Containers.pdf
```

The last two items—*Introduction to Container Security* whitepaper and *The Definitive Guide To Docker Containers*—are directly created from Docker, therefore, they contain information that is directly related to understanding how containers are structured and they breakdown a lot of the Docker information into a central location, which you can download or print out and have at hand at any point of time. They also help you to understand the various layers of containers and how they help keep your environment and applications secure from each other.

Awesome Docker

While this is not a security-related tool, it is a Docker tool that is very useful and is updated quite frequently. Awesome Docker is a curated list of any and all Docker projects. It allows others to contribute with pull requests to the curated list. The list includes topics for those who are looking to get started with Docker; useful articles; deep-dive articles; networking articles; and articles on using multi-server Docker environments, cloud infrastructure, tips, and newsletters, the list just keeps going on. To view the project as well as the *awesomeness* of everything that it includes, visit the following URL:

```
https://github.com/veggiemonk/awesome-docker
```



Summary of Module 4 Chapter 5

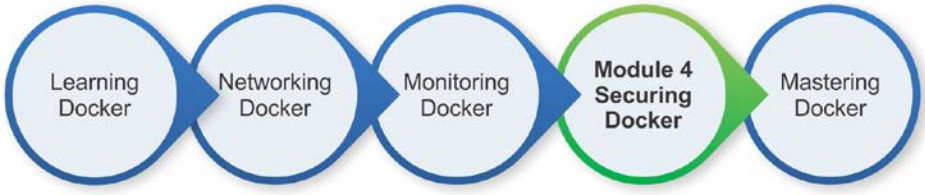
In this chapter, we looked at a number of ways to monitor and report Docker security issues. We looked at some mailing lists that you can join monitoring the Docker CVE list. We also reviewed using both Docker Notary to sign your images as well as hardware signing to utilize hardware items such as YubiKeys. We also looked at using responsible disclosure, which is giving Docker a chance to fix any security-related issue prior to releasing them to the public.

In the next chapter, we will be looking at working with some Docker tools. These tools can be used to secure the Docker environment. We will look at both command-line tools as well as GUI tools that you can use to your advantage. We will be looking at utilizing TLS in your environments using read-only containers, utilizing kernel namespaces and control groups, and mitigating against the risk, while being aware of the Docker daemon attack surface.

Ankita Thakur

Your Course Guide

Your Progress through the Course So Far



6
Using Docker's Built-in Security Features

In this chapter, we will take a look at working with Docker tools that can be used to secure your environment. We will be taking a look at both command-line tools as well as GUI tools that you can utilize to your advantage. We will cover the following items in this chapter:

- Docker tools
    - Using TLS in your environments to help ensure that pieces are communicating securely
    - Using read-only containers to help protect the data in a container from being manipulated in some form
- Docker security fundamentals
    - Kernel namespaces
    - Control groups
    - Linux kernel capabilities

Docker tools

In this section, we will cover the tools that can help you secure your Docker environment. These are options that are built into the Docker software, which you are already using. It's time to learn how to enable or utilize these such features to help give you the peace of mind in order to be sure that the communication is secure; this is where we will cover enabling TLS, which is a protocol that ensures privacy between applications. It ensures that nobody is listening in on the communication. Think of it as when you are watching a movie and people on the phone say, *is this line secure?* It's the same kind of idea when it comes to network communication. Then, we will look at how you can utilize the read-only containers to ensure that the data you are serving up can't be manipulated by anyone.

Using TLS

It is highly recommended to use the Docker Machine to create and manage your Docker hosts. It will automatically set up the communication to use TLS. Here's how you can verify that the *default* host that was created by `docker-machine` indeed uses TLS.

One of the important factors is knowing if you are using TLS or not and then adjusting to use TLS if you are, in fact, not using TLS. The important thing to remember is that, nowadays, almost all the Docker tools ship with the TLS enabled, or if they don't, they appear to be working towards this goal. One of the commands that you can use to check in order to see if your Docker host is utilizing the TLS is with the Docker Machine `inspect` command. In the following, we will take a look at a host and see if it is running with the TLS enabled:

```
docker-machine inspect default


{
    "ConfigVersion": 3,
    "Driver": {
      "IPAddress": "192.168.99.100",
      "MachineName": "default",
      "SSHUser": "docker",
      "SSHPort": 50858,
      "SSHKeyPath": "/Users/scottgallagher/.docker/
        machine/machines/default/id_rsa",
      "StorePath": "/Users/scottgallagher/.docker/machine",
      "SwarmMaster": false,
```

```
    "SwarmHost": "tcp://0.0.0.0:3376",
    "SwarmDiscovery": "",
    "VBoxManager": {},
    "CPU": 1,
    "Memory": 2048,
    "DiskSize": 204800,
    "Boot2DockerURL": "",
    "Boot2DockerImportVM": "",
    "HostDNSResolver": false,
    "HostOnlyCIDR": "192.168.99.1/24",
    "HostOnlyNicType": "82540EM",
    "HostOnlyPromiscMode": "deny",
    "NoShare": false,
    "DNSProxy": false,
    "NoVTXCheck": false
  },
  "DriverName": "virtualbox",
  "HostOptions": {
    "Driver": "",
    "Memory": 0,
    "Disk": 0,
    "EngineOptions": {
      "ArbitraryFlags": [],
      "Dns": null,
      "GraphDir": "",
      "Env": [],
      "Ipv6": false,
      "InsecureRegistry": [],
      "Labels": [],
      "LogLevel": "",
      "StorageDriver": "",
      "SelinuxEnabled": false,
      "TlsVerify": true,
      "RegistryMirror": [],
      "InstallURL": "https://get.docker.com"
    },
```

```
      "SwarmOptions": {
        "IsSwarm": false,
        "Address": "",
        "Discovery": "",
        "Master": false,
        "Host": "tcp://0.0.0.0:3376",
        "Image": "swarm:latest",
        "Strategy": "spread",
        "Heartbeat": 0,
        "Overcommit": 0,
        "ArbitraryFlags": [],
        "Env": null
      },
      "AuthOptions": {
        "CertDir": "/Users/scottgallagher/.docker/machine/certs",
        "CaCertPath": "/Users/scottgallagher/.docker/
          machine/certs/ca.pem",
        "CaPrivateKeyPath": "/Users/scottgallagher/.docker/
           machine/certs/ca-key.pem",
        "CaCertRemotePath": "",
        "ServerCertPath": "/Users/scottgallagher/.docker/
          machine/machines/default/server.pem",
        "ServerKeyPath": "/Users/scottgallagher/.docker/
          machine/machines/default/server-key.pem",
        "ClientKeyPath": "/Users/scottgallagher/.docker/
          machine/certs/key.pem",
        "ServerCertRemotePath": "",
        "ServerKeyRemotePath": "",
        "ClientCertPath": "/Users/scottgallagher/.docker/
          machine/certs/cert.pem",
        "ServerCertSANs": [],
        "StorePath": "/Users/scottgallagher/.docker/
          machine/machines/default"
      }
    },
    "Name": "default"
}
```

From the preceding output, we can focus on the following line:

```
    "SwarmHost": "tcp://0.0.0.0:3376",
```

This shows us that if we were running **Swarm**, this host would be utilizing the secure `3376` port. Now, if you aren't using Docker Swarm, then you can disregard this line. However, if you are using Docker Swarm, then this line is important.

Just to take a step back, let's identify what Docker Swarm is. Docker Swarm is native clustering within Docker. It helps in turning multiple Docker hosts into an easy-to-manage single virtual host:

```
 "AuthOptions": {
            "CertDir": "/Users/scottgallagher/.docker/machine/certs",
            "CaCertPath": "/Users/scottgallagher/.docker/machine/certs/
ca.pem",
            "CaPrivateKeyPath": "/Users/scottgallagher/.docker/machine/
certs/ca-key.pem",
            "CaCertRemotePath": "",
            "ServerCertPath": "/Users/scottgallagher/.docker/machine/
machines/default/server.pem",
            "ServerKeyPath": "/Users/scottgallagher/.docker/machine/
machines/default/server-key.pem",
            "ClientKeyPath": "/Users/scottgallagher/.docker/machine/
certs/key.pem",
            "ServerCertRemotePath": "",
            "ServerKeyRemotePath": "",
            "ClientCertPath": "/Users/scottgallagher/.docker/machine/
certs/cert.pem",
            "ServerCertSANs": [],
            "StorePath": "/Users/scottgallagher/.docker/machine/machines/
default"
        }
```

This shows us that this host is, in fact, using the certificates so we know that it is using TLS, but how do we know from just that? In the following section, we will take a look at how to tell that it is, in fact, using TLS for sure.

Docker Machine also has the option to run everything over TLS. This is the most secure way of using Docker Machine in order 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 see the location on your machine where the certificates are stored at from the preceding output.

Let's take a look at how we can achieve the goal of viewing if our Docker host is utilize TLS:

```
docker-machine ls
NAME      ACTIVE   URL          STATE     URL SWARM   DOCKER    ERRORS
default   *        virtualbox   Running   tcp://192.168.99.100:2376
v1.9.1
```

This is where we can tell that it is using TLS. The insecure port of Docker Machine hosts is the `2375` port and this host is using `2376`, which is the secure TLS port for Docker Machine. Therefore, this host is, in fact, using TLS to communicate, which gives you the peace of mind in knowing that the communication is secure.

Read-only containers

With respect to the `docker run` command, we will mainly focus on the option that allows us to set everything inside the container as read-only. 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 --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. Make sure to use this wisely. Test thoroughly, as it can 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 a single volume (or more, if you use more `-v` switches) to read-only. The thing to remember about volumes is that when you use a volume and mount it in a container, it will mount as an empty volume over the top of 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:/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 in order to keep the data or configuration files intact.

The last option that 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. For doing so, there are some security risks that we need to be aware of. By default, when you do this, the container will get full the 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.

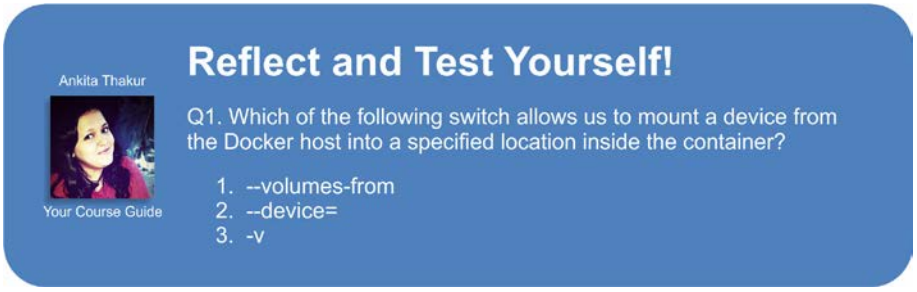
Let's take a look at some of these and see how they work:

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

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

```
$ docker run --device=/dev/sdb:/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 at the `/dev/sdc2` location. However, this one has the `:r` tag at the end of it that specifies that it's read-only and can't be written.



Docker security fundamentals

In the previous sections, we looked into some Docker tools that you can use, such as TLS for communication, and using read-only containers to help ensure data isn't changed or manipulated. In this section, we will focus on some more options that are available from within the Docker ecosystem that can be used to help secure up your environments to another level. We will take a look at the kernel namespaces that provide another layer of abstraction by providing the running process to its own resources that appear only to the process itself and not to other processes that might be running. We will cover more about kernel namespaces in this section. We will then take a look at the control groups. Control groups, more commonly known as cgroups, give you the ability to limit the resources that a particular process has. We will then cover the Linux kernel capabilities. By that, we will look at the restrictions that are placed on containers, by default, when they are run using Docker. Lastly, we will take a look at the Docker daemon attack surface, risks that exist with the Docker daemon that you need to be aware of, and mitigation of these risks.

Kernel namespaces

Kernel namespaces provide a form of isolation for containers. Think of them as a container wrapped inside another container. Processes that are running in one container can't disrupt the processes running inside another container or let alone run on the Docker host that the container is operating on. The way this works is that each container gets its own network stacks to operate with. However, there are ways to link these containers together in order to be able to interact with each other; however, by default, they are isolated from each other. Kernel namespaces have been around for quite a while too, so they are a tried and true method of isolation protection. They were introduced in 2008 and at the time of writing this module, it's 2016. You can see that they will be eight years old, come this July. Therefore, when you issue the `docker run` command, you are benefiting from a lot of heavy lifting that is going on behind the scenes. This heavy lifting is creating its own network stack to operate on. This is also shielding off the container from other containers being able to manipulate the container's running processes or data.

Control groups

Control groups, or more commonly referred to as cgroups, are a Linux kernel feature that allows you to limit the resources that a container can use. While they limit the resources, they also ensure that each container gets the resources it needs as well as that no single container can take down the entire Docker host.

With control groups, you can limit the amount of CPU, memory, or disk I/O that a particular container gets. If we look at the `docker run` command's help, let's highlight the items that we can control. We will just be highlighting a few items that are particularly useful for the majority of users, but please review them to see if any others fit your environment, as follows:

```
$ docker run --help

Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Run a command in a new container

  -a, --attach=[]                 Attach to STDIN, STDOUT or STDERR
  --add-host=[]                   Add a custom host-to-IP mapping
(host:ip)
  --blkio-weight=0                Block IO (relative weight), between 10
and 1000
  --cpu-shares=0                  CPU shares (relative weight)
  --cap-add=[]                    Add Linux capabilities
  --cap-drop=[]                   Drop Linux capabilities
  --cgroup-parent=                Optional parent cgroup for the
container
  --cidfile=                      Write the container ID to the file
  --cpu-period=0                  Limit CPU CFS (Completely Fair
Scheduler) period
  --cpu-quota=0                   Limit 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)
  -d, --detach=false              Run container in background and print
container ID
  --device=[]                     Add a host device to the container
  --disable-content-trust=true    Skip image verification
  --dns=[]                        Set custom DNS servers
  --dns-opt=[]                    Set DNS options
  --dns-search=[]                 Set custom DNS search domains
  -e, --env=[]                    Set environment variables
```

```
  --entrypoint=                    Overwrite the default ENTRYPOINT of the
image
  --env-file=[]                    Read in a file of environment variables
  --expose=[]                      Expose a port or a range of ports
  --group-add=[]                   Add additional groups to join
  -h, --hostname=                  Container host name
  --help=false                     Print usage
  -i, --interactive=false          Keep STDIN open even if not attached
  --ipc=                           IPC namespace to use
  --kernel-memory=                 Kernel memory limit
  -l, --label=[]                   Set meta data on a container
  --label-file=[]                  Read in a line delimited file of labels
  --link=[]                        Add link to another container
  --log-driver=                    Logging driver for container
  --log-opt=[]                     Log driver options
  --lxc-conf=[]                    Add custom lxc options
  -m, --memory=                    Memory limit
  --mac-address=                   Container MAC address (e.g.
92:d0:c6:0a:29:33)
  --memory-reservation=            Memory soft limit
  --memory-swap=                   Total memory (memory + swap), '-1' to
disable swap
  --memory-swappiness=-1           Tuning container memory swappiness (0
to 100)
  --name=                          Assign a name to the container
  --net=default                    Set the Network for the container
  --oom-kill-disable=false         Disable OOM Killer
  -P, --publish-all=false          Publish all exposed ports to random
ports
  -p, --publish=[]                 Publish a container's port(s) to the
host
  --pid=                           PID namespace to use
  --privileged=false               Give extended privileges to this
container
  --read-only=false                Mount the container's root filesystem
as read only
  --restart=no                     Restart policy to apply when a
container exits
```

```
  --rm=false                        Automatically remove the container when
it exits
  --security-opt=[]                 Security Options
  --sig-proxy=true                  Proxy received signals to the process
  --stop-signal=SIGTERM             Signal to stop a container, SIGTERM by
default
  -t, --tty=false                   Allocate a pseudo-TTY
  -u, --user=                       Username or UID (format:
<name|uid>[:<group|gid>])
  --ulimit=[]                       Ulimit options
  --uts=                            UTS namespace to use
  -v, --volume=[]                   Bind mount a volume
  --volume-driver=                  Optional volume driver for the
container
  --volumes-from=[]                 Mount volumes from the specified
container(s)
  -w, --workdir=                    Working directory inside the container
```

As you can see from the preceding highlighted portions, these are just a few items that you can control on a per-container basis.

Linux kernel capabilities

Docker uses the kernel capabilities to place the restrictions that Docker places on the containers when they are launched or started. Limiting the root access is the ultimate agenda with these kernel capabilities. There are a few services that typically run as root, but can now be run without these permissions. Some of these include `SSH`, `cron`, and `syslogd`.

Overall, what this means is that you don't need root in the server sense you typically think of. You can run with a reduced capacity set. This means that your root user doesn't need the privilege it typically needs.

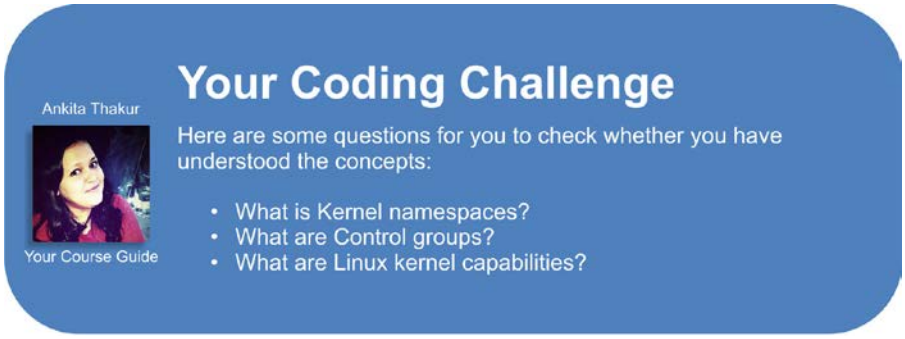
Some of the things that you might not need to enable anymore are shown in the following:

- Performing mount operations
- Using raw sockets, which will help to prevent spoofing of packets
- Creating new devices
- Changing the owner of files
- Altering attributes

This helps due to the fact that if someone does compromise a container, then they can't escalate any more than what you are providing them. It will be much harder, if not impossible, to escalate their privileges from a running container to running Docker host. Due to such complexity, the attackers will probably look elsewhere than your Docker environments to try to attack. Docker also supports the addition and removal of capabilities, therefore, it's recommend to remove all the capabilities, except the ones that you intend to use. An example would be to use the `-cap-add net_bind_service` switch on your `docker run` command.

Containers versus virtual machines

Hopefully, you trust your organization and all those who have access to these systems. 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. Therefore, you will be aware of what is inside the virtual machine and what isn't. This being said, with the Docker containers, you will not be aware of what is there inside the image that you may be using for your container(s).



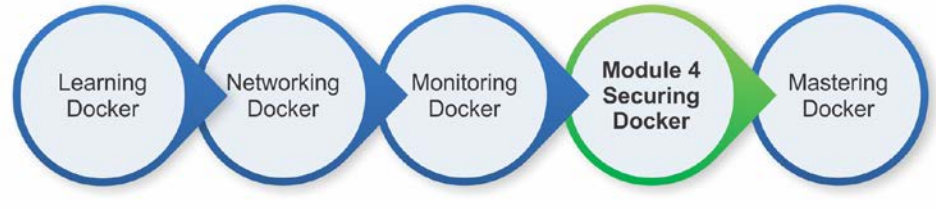
Summary of Module 4 Chapter 6

In this chapter, we looked at deploying TLS to all the pieces of our Docker environment so that we can ensure that everything is communicating securely and the traffic can't be intercepted and then interpreted. We also understood how to utilize the read-only containers to our advantage in order to ensure the data that is being served up can't be manipulated. We then took a look at how to provide processes with their own abstraction of items, such as networks, mounts, users, and more. We then dove into control groups, or cgroups as their more commonly referred to as, as a way to limit the resources that a process or container has. We also took a look at the Linux kernel capabilities, that is, the restrictions that are placed on a container when it is started or launched. Lastly, we dove into mitigating risks against the Docker daemon attack surface.

In the next chapter, we will look at securing Docker with third-party tools and learn which third-party tools, beyond those offered by Docker, are out there to help secure your environments to help keep your application(s) secure when running on Docker.

Ankita Thakur

Your Course Guide



7
Securing Docker with Third-Party Tools

In this chapter, let's take a look at securing Docker using third-party tools. These would be tools that are not part of the Docker ecosystem, which you can use to help secure your systems. We will be taking a look at the following three items:

- **Traffic Authorization**: This allows inbound and outbound traffic to be verified by the token broker in order to ensure that traffic between services is secure.
- **Summon**: Summon is a command-line tool that reads a file in the `secrets.yml` format and injects secrets as environment variables into any process. Once the process exits, the secrets are gone.
- **sVirt and SELinux**: sVirt is a community project that integrates **Mandatory Access Control** (**MAC**) security and Linux-based virtualization (**Kernel-base Virtual Machine** (**KVM**), lguest, and so on).

We will then add bonus material with regards to some extra third-party tools that are quite useful and powerful and deserve to get some recognition as useful third-party tools. These tools include **dockersh**, **DockerUI**, **Shipyard**, and **Logspout**. Without further ado, let's jump in and get started on our path to the most secure environments that we can obtain.

Third-party tools

So, what third-party tools will we focus on? Well from the preceding introduction, you learned that we will be looking at three tools in particular. These would be Traffic Authorization, Summon, and sVirt with SELinux. All the three tools help in different aspects and can be used to perform different things. We will learn the differences between them and help you to determine which ones to implement. You can decide whether you want to implement them all, only one or two of them, or maybe you feel that none of these would pertain to your current environment. However, it is good to know what is out there, in case, your needs change and the overall architecture of your Docker environments change over time.

Traffic Authorization

Traffic Authorization can be used to regulate HTTP/HTTPS traffic between services. This involves a forwarder, gatekeeper, and token broker. This allows inbound and outbound traffic to be verified by the token broker in order to ensure that traffic between services is secure. Each container runs a gatekeeper that is used to intercept all the HTTP/HTTPS inbound traffic and verifies its authenticity from a token that is found in the authorization header. The forwarder also runs on each container, and like the gatekeeper, this also intercepts traffic; however, instead of intercepting inbound traffic, it intercepts outbound traffic and places the token on the authorization header. These tokens are issues from the token broker. These tokens can also be cached to save time and minimize the impact of latency. Let's break it down into a series of steps, as shown in the following:

1. Service A initiates a request to Service B.
2. The forwarder on Service A will authenticate itself with the token broker.
3. The token broker will issue a token that Service A will apply to the authorization header and forward the request to Service B.
4. Service B's gatekeeper will intercept the request and verify the authorization header against the token broker.
5. Once the authorization header has been verified, it is then forwarded to Service B.

As you can see, this applies extra authorizations on both inbound and outbound requests. As we will see in the next section, you can also use Summon along with Traffic Authorization to use shared secrets that are available once they are used, but go away once the application has completed its actions.

For more information about Traffic Authorization and Docker, visit `https://blog.conjur.net/securing-docker-with-secrets-and-dynamic-traffic-authorization`.

Summon

Summon is a command-line tool and is used to help pass along secrets or things you don't want exposed, such as passwords or environmental variables and then these secrets are disposed upon exiting the process. This is great as once the secret is used and the process exits, the secret no longer exists. This means the secret isn't lingering around until it is either removed manually or discovered by an attacker for malicious use. Let's take a look at how to utilize Summon.

Summon typically uses three files: a `secrets.yml` file, script used to perform the action or task, and Dockerfile. As you have learned previously, or based on your current Docker experience, the Dockerfile is the basis of what helps in building your containers and has instructions on how to set up the container, what to install, what to configure, and so on.

One great example have for the usage of Summon is to be able to deploy your AWS credentials to a container. For utilizing AWS CLI, you need a few key pieces of information that should be kept secret. These two pieces of information are your **AWS Access Key ID** and **AWS Secret Access Key**. With these two pieces of information, you can manipulate someone's AWS account and perform actions within this account. Let's take a look at the contents of one of these files, the `secrets.yml` file:

```
secrets.yml
AWS_ACCESS_KEY_ID: !var $env/aws_access_key_id
AWS_SECRET_ACCESS_KEY: !var $env/aws_secret_access_key
```

The `-D` option is used to substitute values while `$env` is an example of a substitution variable, therefore, the options can be interchanged.

In the preceding content, we can see that we want to pass along these two values into our application. With this file, the script file you want to deploy, and the Dockerfile, you are now ready to build your application.

We simply utilize the `docker build` command inside the folder that has our three files in it:

```
$ docker build -t scottpgallagher/aws-deploy .
```

Next, we need to install Summon, which can be done with a simple `curl` command, as follows:

```
$ curl -sSL https://raw.githubusercontent.com/conjurinc/summon/master/
install.sh | bash
```

Now that we have Summon installed, we need to run the container with Summon and pass along our secret values (note that this will only work on OS X):

```
$ security add-generic-password -s "summon" -a "aws_access_key_id" -w
"ACESS_KEY_ID"
$ security add-generic-password -s "summon" -a "aws_secret_access_key" -w
"SECRET_ACCESS_KEY"
```

Now we are ready to run Docker with Summon in order to pass along these credentials to the container:

```
$ summon -p ring.py docker run —env-file @ENVFILE aws-deploy
```

You can also view the values that you have passed along by using the following `cat` command:

```
$ summon -p ring.py cat @SUMMONENVFILE
aws_access_key_id=ACESS_KEY_ID
aws_secret_access_key=SECRET_ACCESS_KEY
```

The `@SUMMONENVFILE` is a memory-mapped file that contains the values from the `secrets.yml` file.

For more information and to see other options to utilize Summon, visit `https://conjurinc.github.io/summon/#examples`.

sVirt and SELinux

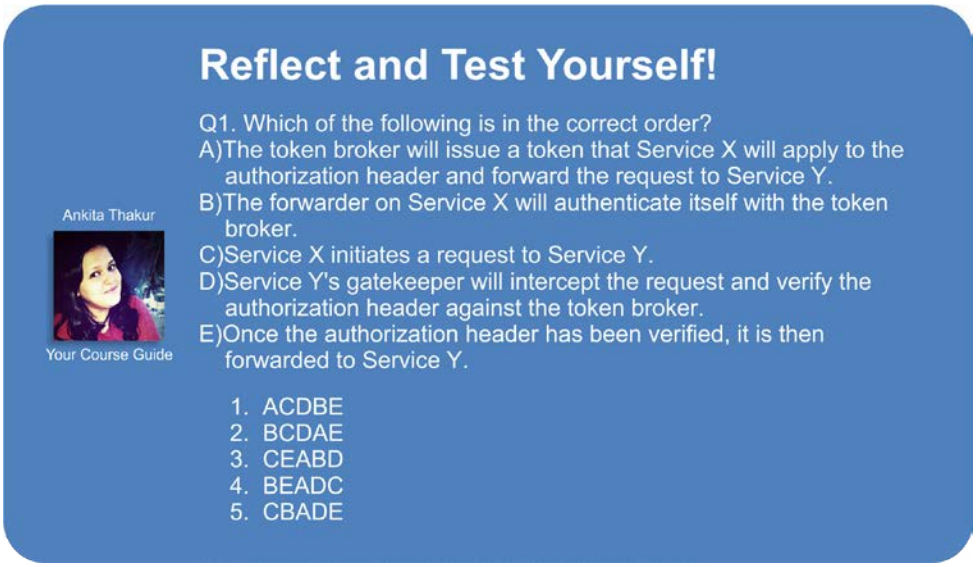
sVirt is part of the SELinux implementation, but it is typically turned off as most view it as a roadblock. The only roadblock should be learning sVirt and SELinux.

sVirt is an open source community project that implements MAC security for Linux-based virtualization. A reason you would want to implement sVirt is to improve the security as well as harden the system against any bugs that might exist in the hypervisor. This will help in eliminating any attack vectors that might be aimed towards the virtual machine or host.

Remember that all containers on a Docker host share the usage of the Linux kernel that is running on the Docker host. If there is an exploit to this Linux kernel on the host, then all containers running on this Docker host have the potential to be easily compromised. If you implement sVirt and a container is compromised, there is no way for the compromise to reach your Docker host and then out to other Docker containers.

sVirt utilizes labels in the same way as SELinux. The following table is a list of these labels and their descriptions:

Type	SELinux Context	Description
Virtual machine processes	`system_u:system_r:svirt_t:MCS1`	`MCS1` is a randomly selected MCS field. Currently, approximately 500,000 labels are supported.
Virtual machine image	`system_u:object_r:svirt_image_t:MCS1`	Only processes labeled `svirt_t` with the same MCS fields are able to read/write these image files and devices.
Virtual machine shared read/write content	`system_u:object_r:svirt_image_t:s0`	All processes labeled `svirt_t` are allowed to write to the `svirt_image_t:s0` files and devices.
Virtual machine image	`system_u:object_r:virt_content_t:s0`	This is the system default label used when an image exits. No `svirt_t` virtual processes are allowed to read files/devices with this label.



Other third-party tools

There are some other third-party tools that do deserve a mention in this chapter and are worth exploring to see the value that they can add for you. It seems that these days, a lot of focus is on GUI applications to help with securing applications and infrastructures. The following utilities will give you a few options that could be pertinent to the environment you are running with the Docker tools.

Note that you should use caution when implementing some of the following items as there could be unwanted repercussions. Make sure to use testing environments prior to production implementation.

dockersh

The dockersh was designed to be used as a login shell replacement on machines that support multiple interactive users. Why is this important? If you remember some of the general security warnings that you have when dealing with Docker containers on a Docker host, you will know that whoever has access to the Docker host has access to all the running containers on this Docker host. With dockersh, you can isolate the use on a per-container basis and only allow users access the containers that you want them to, while maintaining administrative control over the Docker host and keeping the security threshold minimum.

This is an ideal way to help isolate users on a per-container basis, while containers help eliminate the need for SSH by utilizing dockersh, you can remove some of these fears about providing everyone that needs container to access, the access to the Docker host(s) as well. There is a lot of information required to set up and invoke dockersh, therefore, if you are interested, it's recommended to visit the following URL to find more about dockersh, including how to set it up and use it:

```
https://github.com/Yelp/dockersh
```

DockerUI

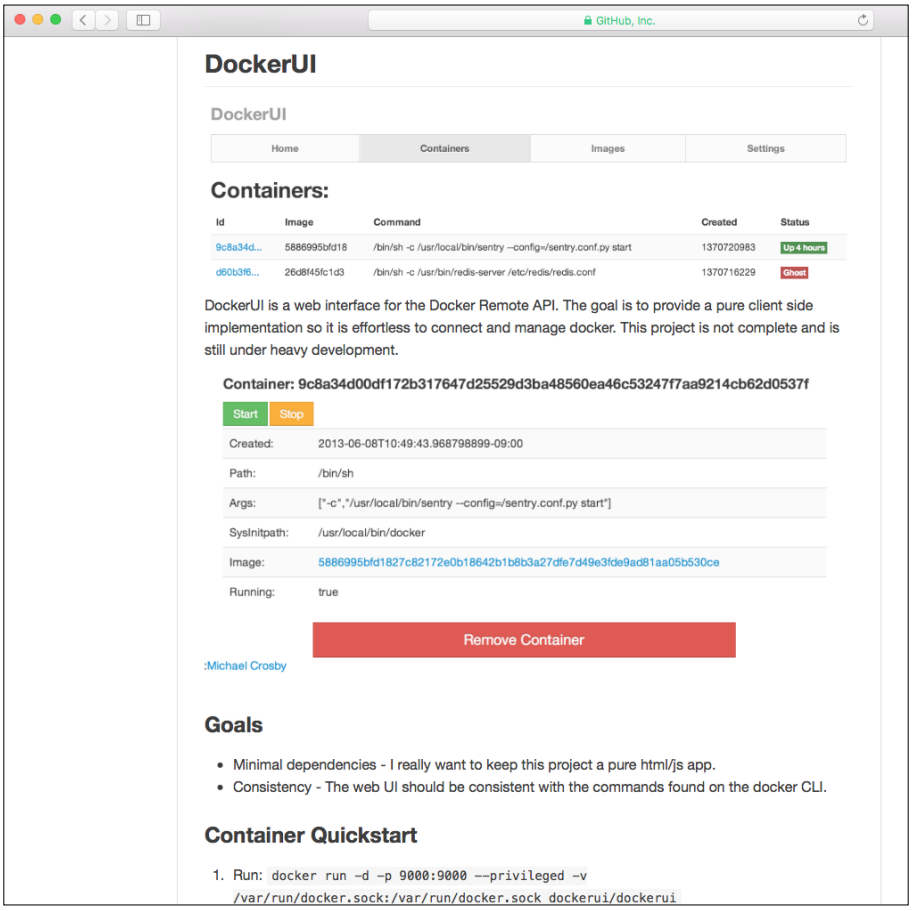
DockerUI is a simple way to view what is going on inside your Docker host. The installation of DockerUI is very straightforward and is done by running a simple `docker run` command in order to get started:

```
$ docker run -d -p 9000:9000 --privileged -v /var/run/docker.sock:/var/
run/docker.sock dockerui/dockerui
```

To access the DockerUI, you simply open a browser and navigate to the following link:

```
http://<docker_host_ip>:9000
```

This opens your DockerUI to the world on port `9000`, as shown in the following screenshot:



You can get the general high-level view of your Docker host and its ecosystem and can do things such as manipulate the containers on the Docker host by restarting, stopping, or starting them from a stopped state. DockerUI takes some of the steep learning curve of running command-line items and places them into actions that you perform in a web browser using point and click.

For more information about DockerUI, visit `https://github.com/crosbymichael/dockerui`.

Shipyard

Shipyard, like DockerUI, allows you to use a GUI web interface to manage various aspects—mainly in your containers—and manipulate them. Shipyard is build on top of Docker Swarm so that you get to utilize the feature set of Docker Swarm, where you can manage multiple hosts and containers instead of having to just focus on one host and its containers at a time.

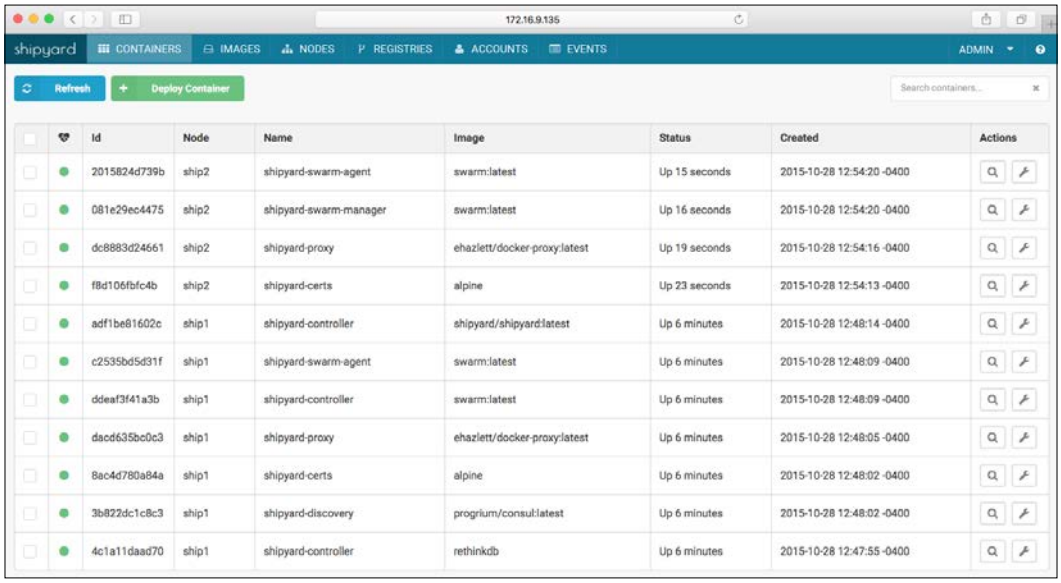
Using Shipyard is simple and the following `curl` command re-enters the picture:

```
$ curl -sSL https://shipyard-project.com/deploy | bash -s
```

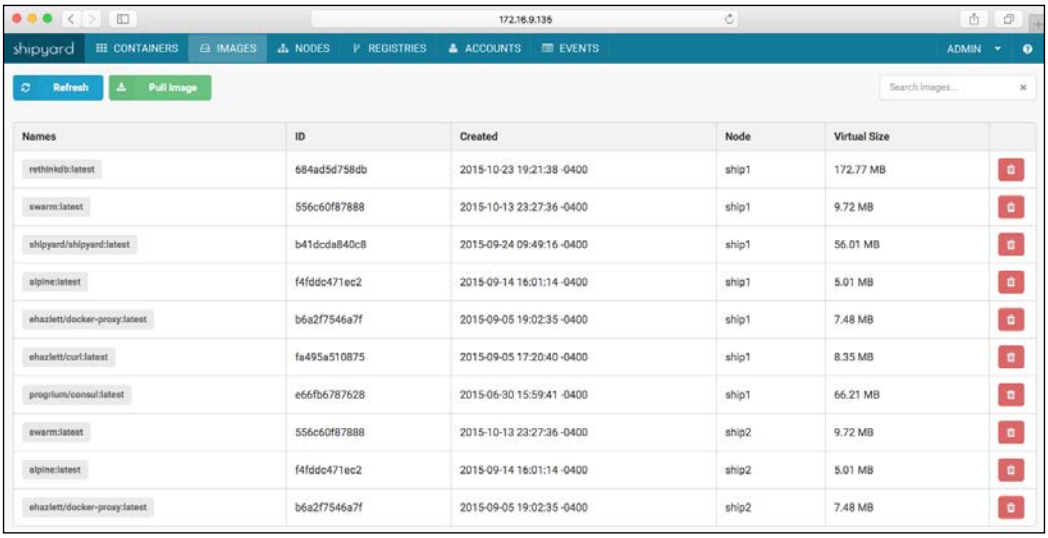
To access the Shipyard once the set up is completed, you can simply open a browser and navigate to the following link:

```
http://<docker_host_ip>:8080
```

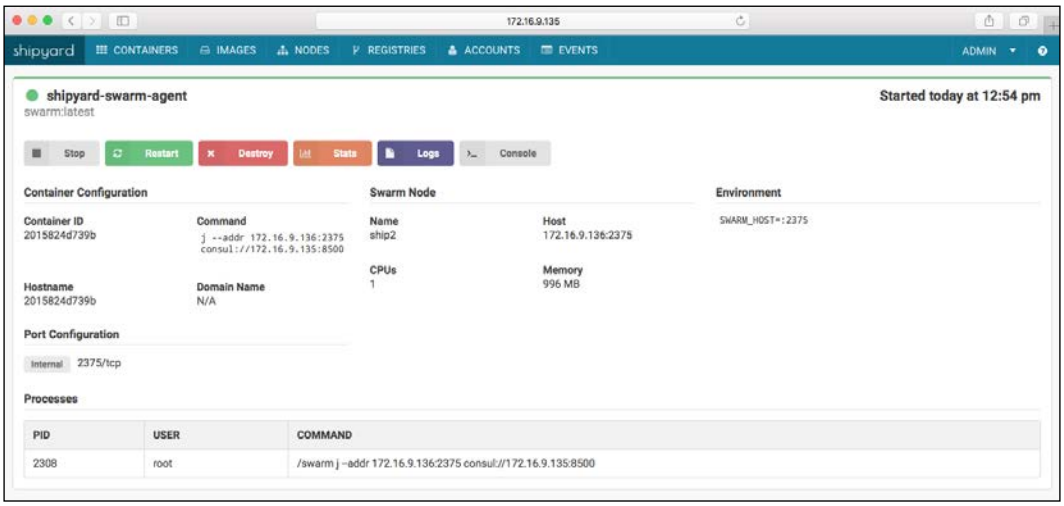
As we can see in the following screenshot, we can view all the containers on our Docker host:



We can also view all the images that are on our Docker host, as shown in the following screenshot:



We can also control our containers, as seen in the following screenshot:



Shipyard, like DockerUI, allows you to manipulate your Docker hosts and containers, by restarting them, stopping them, starting them from a failed state, or deploying new containers and having them join the Swarm cluster. Shipyard also allows you to view information such as port mapping information that is what port from the host maps to the container. This allows you to get a hold of important information like that when you need it quickly to address any security related issues. Shipyard also has user management where DockerUI lacks such capability.

For more information about Shipyard simply visit the following URLs:

- `https://github.com/shipyard/shipyard`
- `http://shipyard-project.com`

Logspout

Where do you go when there is an issue that needs to be addressed? Most people will first look at the logs of that application to see if it is outputting any errors. With Logspout, this becomes a much more manageable task with many multiple running containers. With Logspout, you can route all the logs for each and every container to a location of your choice. Then, you could parse these logs in one place. Instead of having to pull the logs from each container and review them individually you can instead have Logspout do that work for you.

Logspout is just as easy to set up as we have seen for other third-party solutions. Simply run the following command on each Docker host to start collecting the logs:

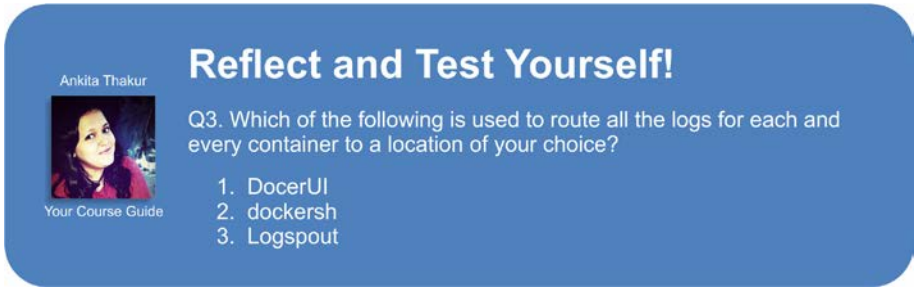
```
$ docker run --name="logspout" \
    --volume=/var/run/docker.sock:/tmp/docker.sock \
    --publish=127.0.0.1:8000:8080 \
    gliderlabs/logspout
```

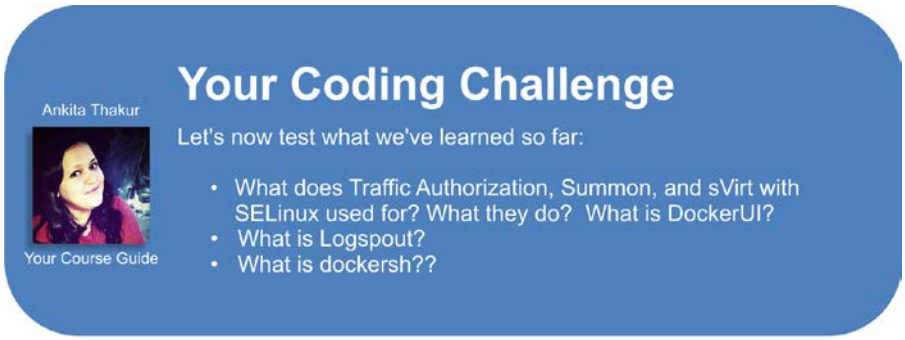
Now that we have all the container logs collected in one area, we need to parse through these logs, but how do we do it?

```
$ curl http://127.0.0.1:8000/logs
```

Here's the `curl` command to the rescue again! Logs get prefixed with the container names and colorized in a manner in order to distinguish the logs. You can replace the loopback (`127.0.0.1`) address in the `docker run` invocations with the IP address of the Docker host so that it's easier to connect to in order to be able to get the logs as well as change the port from `8000` to something of your choice. There are also different modules that you can utilize to obtain and collect logs.

For more information about Logspout, visit `https://github.com/gliderlabs/logspout`.





Summary of Module 4 Chapter 7

In this chapter, we looked at some third-party tools in order to be able to help secure Docker environments. Mainly, we looked at three tools: Traffic Authorization, Summon, and sVirt with SELinux. All the three can be utilized in different ways to help secure your Docker environments to give you the peace of mind at end of the day to run your applications in the Docker containers. We learned what thirdparty tools, beyond those offered by Docker, are out there to help secure your environments to keep your application(s) secure when running on Docker.

Ankita Thakur

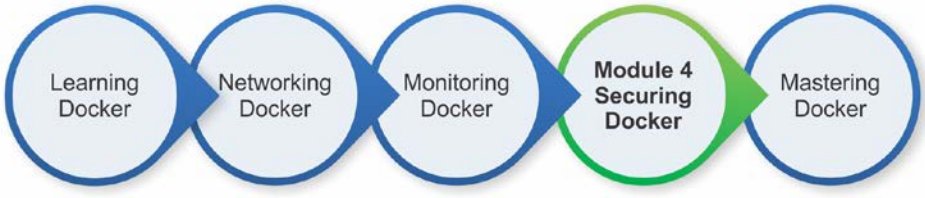
Your Course Guide

We then took a look at some other third-party tools. These are extra tools that are worthwhile to some, given what your Docker environment setup looks like. Some of these tools include dockersh, DockerUI, Shipyard, and Logsprout. These tools, when carefully applied, layer on extra enhancements to help in the overall security of your Docker configurations.

In the next chapter, we will be looking at keeping up on security. With so much going on these days that surrounds the security, it's sometimes tough to know where to look for updated information and be able to apply quick fixes.

You will be learning to help enforce the idea of keeping security in the forefront of your mind and subscribing to things such as e-mail lists that not only include Docker, but also include items that are related to the environments you are running with Linux. Other items are keeping up on following what is going on with regards to items such as GitHub issues that relate to Docker security, following along in the IRC rooms, and watching websites such as the CVE.

Your Progress through the Course So Far



8
Keeping up Security

In this chapter, we will be taking a look at keeping up with security as it relates to Docker. By what means you can use to help keep up to date on Docker-related security issues that are out there for the version of the Docker tools you might be running now? How do you stay ahead of any security issues and keep your environments secure even with threats? In this chapter, we will look at multiple ways in which you can keep up on any security issues that arise and the best way to obtain information as quickly as possible. You will cover learning to help enforce the idea of keeping security in the forefront of your mind and subscribing to things such as e-mail lists that not only include Docker, but also include items that are related to the environments you are running with Linux. Other items are keeping up on following what is going on with regards to items such as GitHub issues that relate to Docker security, following along with the **Internet Relay Chat** (**IRC**) rooms, and watching websites such as the CVE.

In this chapter, we will be covering the following topics:

- Keeping up with security
    - E-mail list options
    - GitHub issues
    - IRC rooms
    - CVE websites
- Other areas of interest

Keeping up with security

In this section, we will take a look at the multiple ways that you can obtain or keep up to date about the information related to the security issues that may occur in Docker products. While this isn't a complete list of tools that you can use to keep up on issues, this is a great start and consists of the most commonly used items that are used. These items include e-mail distribution lists, following the GitHub issues for Docker, IRC chat rooms for the multiple Docker products that exist, CVE website(s), and other areas of interest to follow on items that relate to Docker products, such as the Linux kernel vulnerabilities and other items you can use to mitigate the risks.

E-mail list options

Docker operates two mailing lists that users can sign up to be a part of. These mailing lists provide means to both gather information about the issues or projects others are working on and spark your thoughts into doing the same for your environment. You can also use them to help blanket the Docker community with questions or issues that you are running into when using various Docker products or even other products in relation to Docker products.

The two e-mail lists are as follows:

- Docker-dev
- Docker-user

What is the Docker-dev mailing list mostly geared towards? You guessed it, it is geared towards the developers! These are the people who are either interested in developer type roles and what others are developing or are themselves developing code for something that might integrate into various Docker products. This could be something such as creating a web interface around Docker Swarm. This list would be the one you want to post your questions at. The list consists of other developers and possibly even those that work at Docker itself that might be able to help you with any questions or issues that you have.

The other list, the Docker-user list, is geared towards the users of the various Docker products or services and have questions on either how to use the products/services or how they might be able to integrate third-party products with Docker. This might include how to integrate **Heroku** with Docker or use Docker in the cloud. If you are a user of Docker, then this list is the right one for you. You can also contribute to the list as well if you have advanced experience, or something comes across the list that you have experience in, or have dealt with previously.

There is no rule that says you can't be on both. If you want to get the best of both worlds, you can sign up for both and gauge the amount of traffic that comes across each one and then make the decision to only be on one, based on where your interests lie. You also have the option of not joining the lists and just following them on the Google Groups pages for each list.

The Google groups page for the Docker-dev list is `https://groups.google.com/forum/#!forum/docker-dev` and the Google groups page for the Docker-user list is `https://groups.google.com/forum/#!forum/docker-user`.

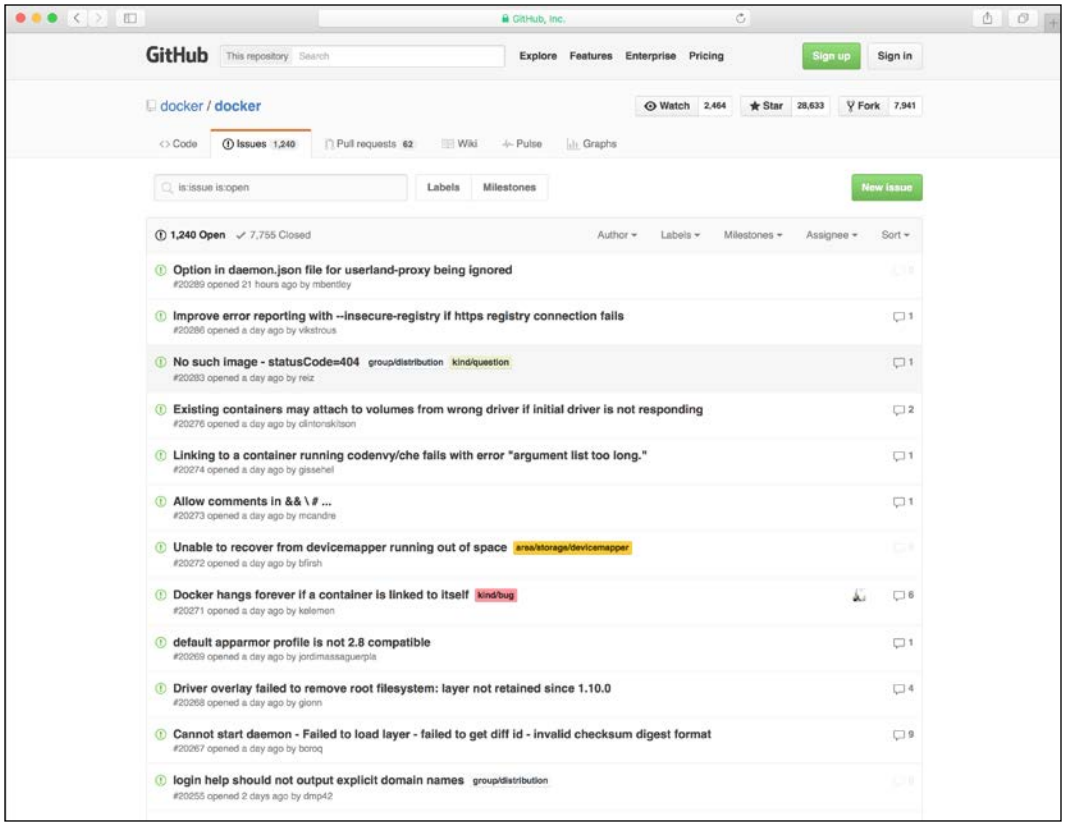
Don't forget that you can also search through these lists to see if your issue or questions might have already been answered. As this module is about security, don't forget that you can use these two mailing lists to discuss items that are security related—whether they be development or user related.

GitHub issues

Another method of keeping up with security-related issues is to follow the GitHub issues. As all the code for the Docker core and other various piece of Docker such as **Machine, Swarm**, **Compose**, and all others are stored on GitHub, it provides an area. What exactly are GitHub issues and why should I care about them is what you are probably asking yourself right now. GitHub Issues is a bug tracker system that GitHub uses. By tracking these issues, you can view the issues that others are experiencing and get ahead of them in your own environment, or it could solve the problem in your environment, knowing that others are having the same issue and it's not just on your end. You can stop pulling what is left of your hair.

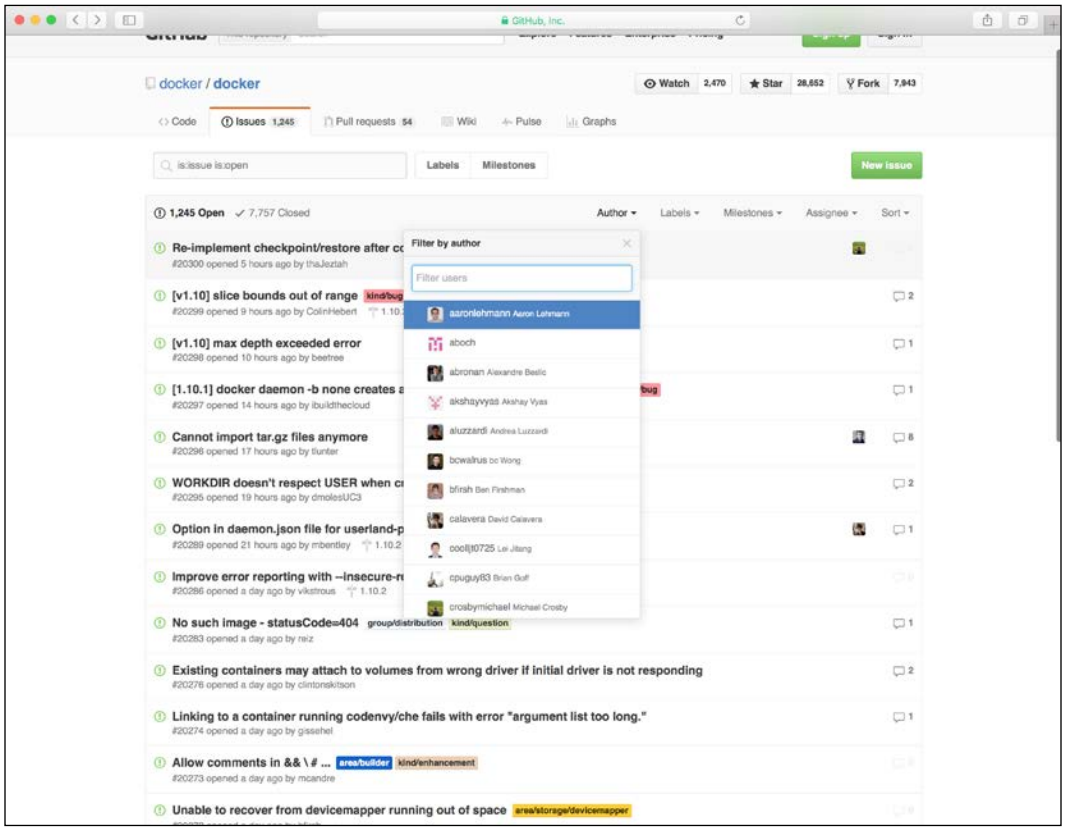
As each GitHub repository has its own issues section, we don't need to look at each and every issues section, but I believe it is worthwhile to view one of the repositories issues section so that you know what exactly you are looking at in order to help decipher it all.

The following screenshot (which can be found at `https://github.com/docker/docker/issues`) shows all the current issues that exist with the Docker core software code:

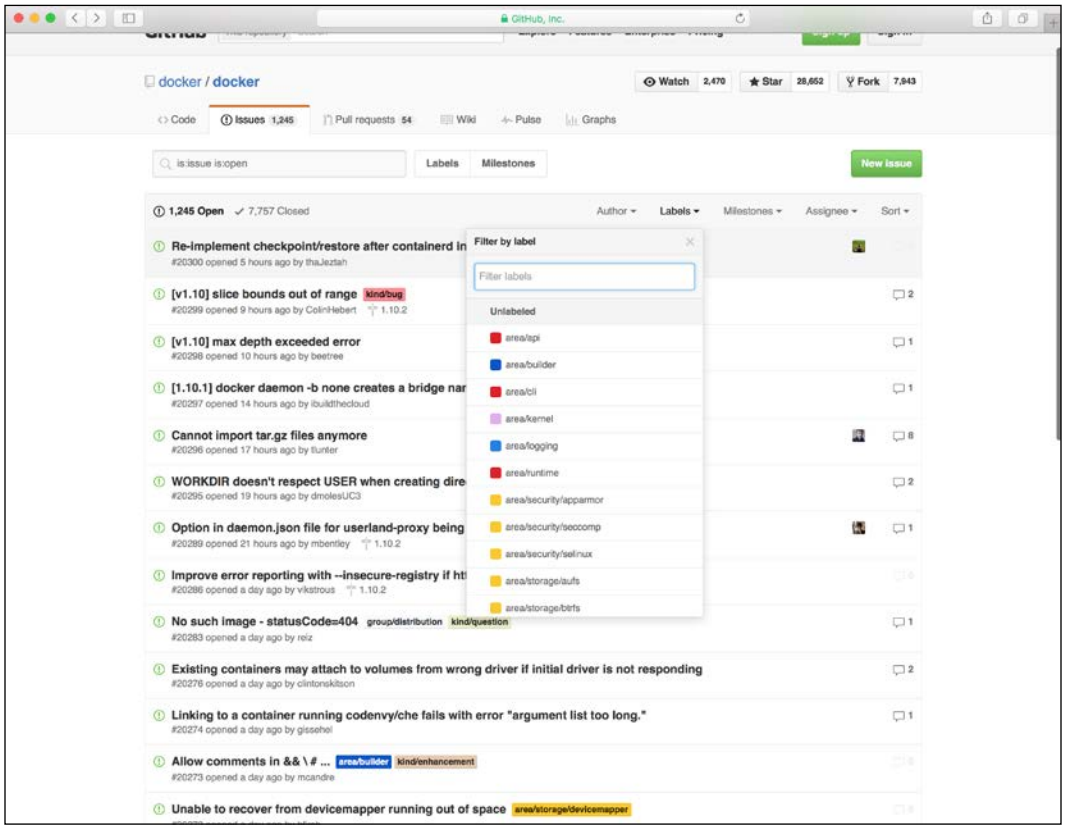


From this screen, we can not only see how many issues are open, but also know how many have been closed. These are issues that were once an issue and solutions were derived for them and now they have been closed. The closed ones are here for historic purposes in order to be able to go back in time and see what solution might have been provided to solve an issue.

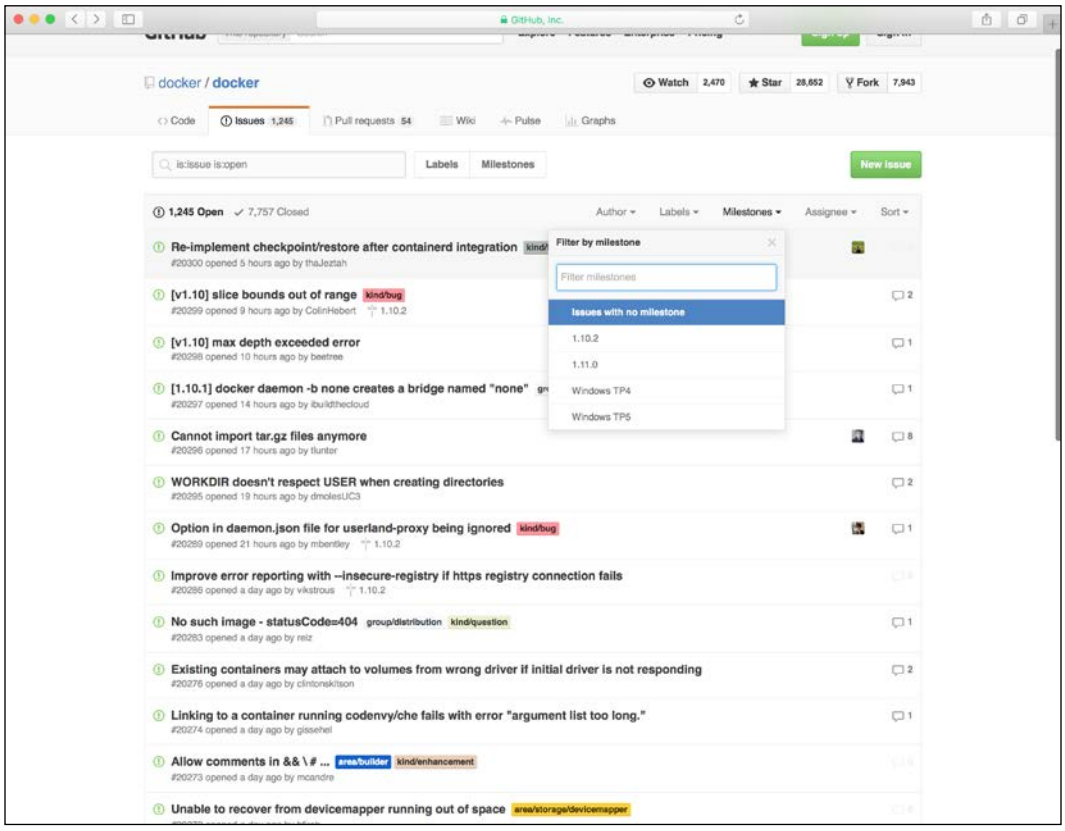
In the following screenshot, we can filter the issue based on the author, that is, the person who submitted the issue:



In the following screenshot, we can also filter the issue based on labels and these might include **api**, **kernel**, **apparmor**, **selinux**, **aufs**, and many more:

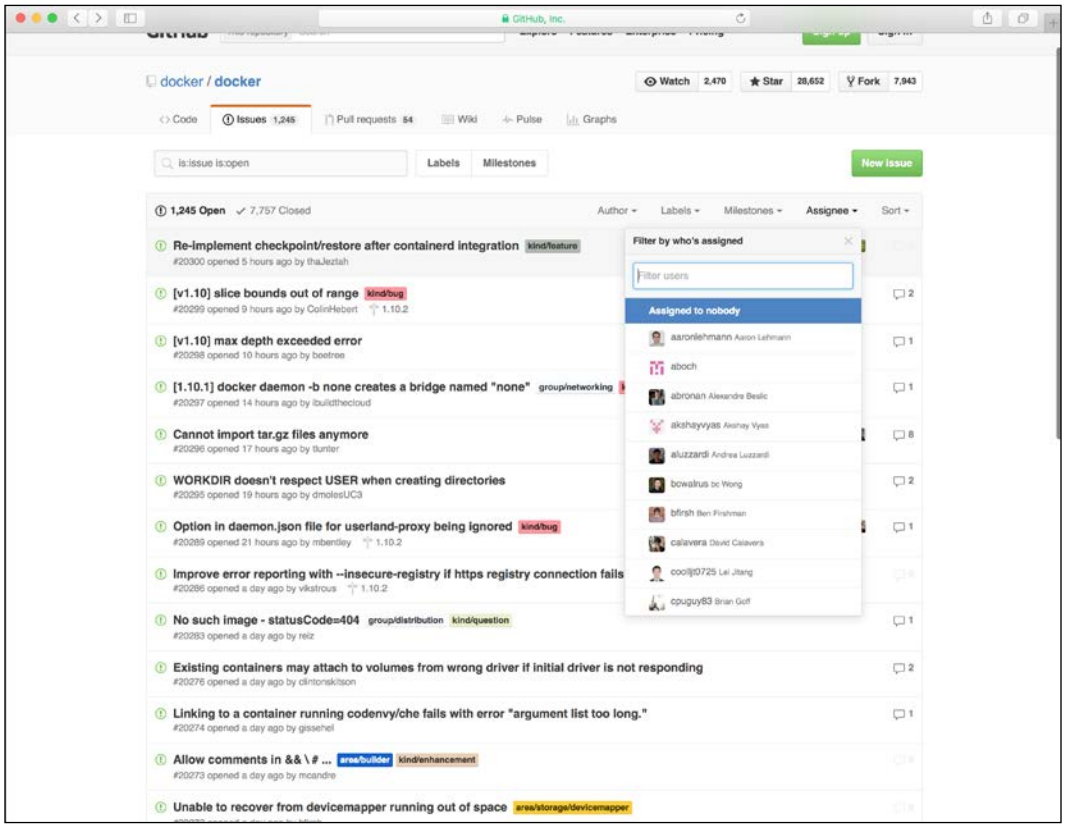


In the following screenshot, we see that we can also filter by milestone:



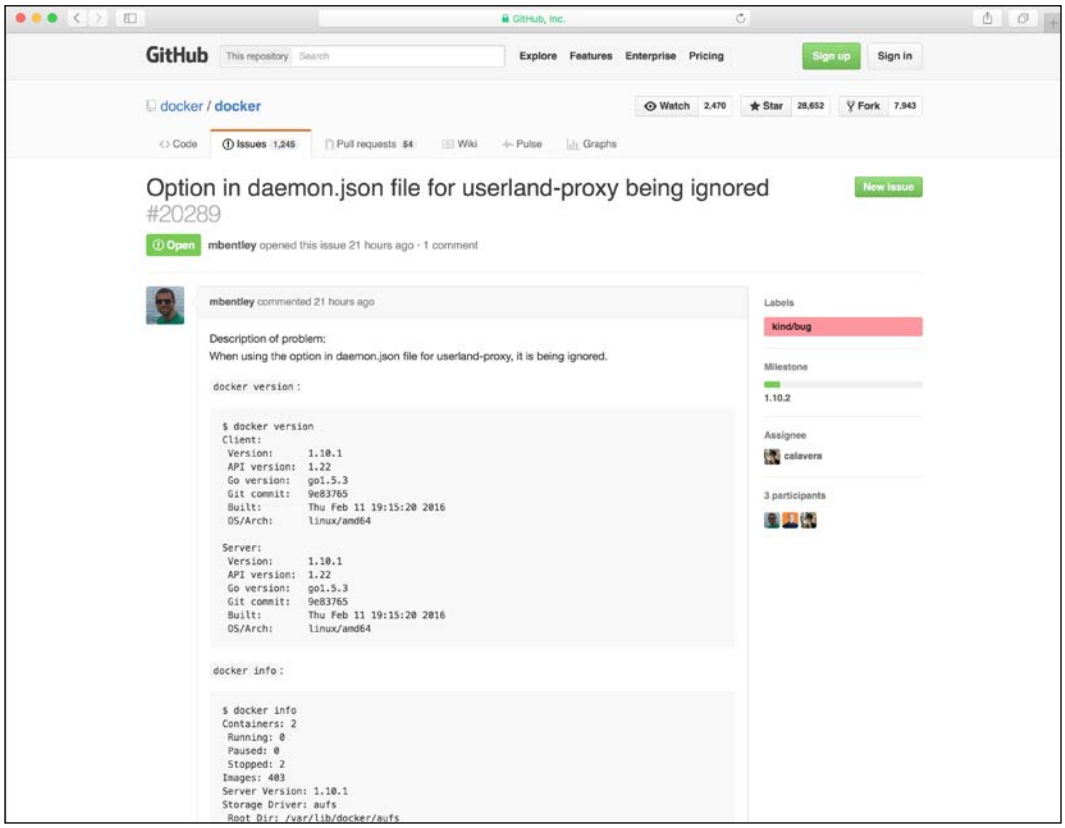
Milestones are essentially tags to help sort issues based on fixing an issue for a particular purpose. They can also be used to plan upcoming releases. As we can see here, some of these include **Windows TP4** and **Windows TP5**.

Lastly, we can filter issues based on assignee, that is, the person to whom it is assigned to fix or address the issue, as shown in the following screenshot:



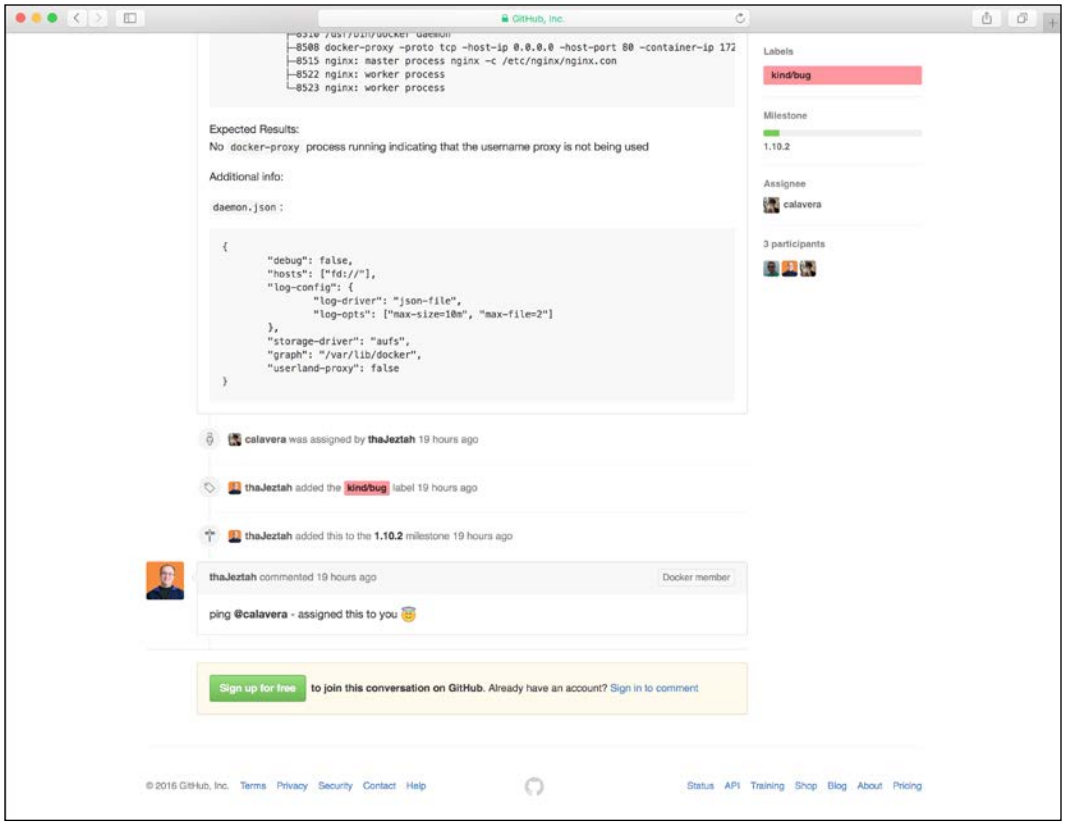
As we can see, there are lot of ways in which we can filter the issues, but what does an issue actually look like and what does it contain? Let's take a look at that in the following section.

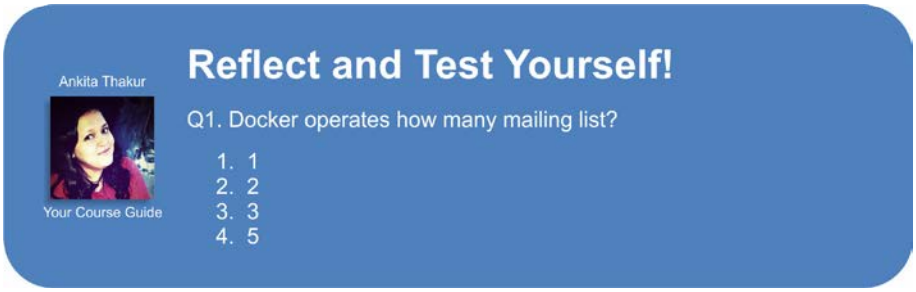
In the following screenshot, we can see what an actual issue looks like:



Some of the information that we can see is the title of the issue and the unique issue number. We can then see that this particular issue is open, the person who reported the issue, and for how long it's opened. We can then see how many comments are there on the issue and then a large explanation of the issue itself. On the right-hand side, we can see what labels the issue has, what its milestone is, who it is assigned to, and how many participants are involved in the issue. Those involved are people who have commented on the issue in some way.

In the last image, which is at the bottom of the issue from the preceding image, we can see the timeline of the issue, such as who it was assigned to and when, as well as when it was assigned a label and any additional comments.





IRC rooms

The first thing to understand is what exactly IRC is. If you think back to the older days, we probably all had some form of IRC rooms when we had AOL and had chat rooms that you could join based on your location or topic. IRC operates in the same way where there is a server that clients, such as yourself, connect to. These rooms are typically based on a topic, product, or service that people have in common that can come together to discuss. You can chat as a group but also utilize private chats with others in the same room or channel as you.

Docker utilizes IRC for discussion about its products. This allows not only end users of the products to engage in discussion, but also in the case of Docker, most of those who actually work for Docker and on these products tend to be in these rooms on a daily basis and will engage with you about issues you might be seeing or questions you have.

With IRC, there are multiple servers that you can use to connect to the hosted channels. Docker uses the `http://freenode.net` server (it is the server you would use if you were to use a third-party client to connect to IRC; however, you can also use `http://webchat.freenode.net`) and then all their channels for their products are things such as **#docker**, **#docker-dev**, **#docker-swarm**, **#docker-compose**, and **#docker-machine**. All channels start with the pound sign (#), followed by the channel name. Within these channels, there are discussion for each product. Beyond these channels, there are other channels where you can discuss Docker-related topics. In the previous chapter, we discussed the Shipyard project, which allows you to have a GUI interface that overlays on top of your Docker Swarm environment. If you had questions about this particular product, you could join the channel for that product, which is **#shipyard**. There are other channels you can join as well and more created daily. To get a list of channels, you will need to connect to your IRC client and issue a command. Follow the given link to find out how to do this:

```
http://irc.netsplit.de/channels/?net=freenode
```

Chat archives are also kept for each channel, therefore, you can search through them as well to find out whether discussions are happening around a question or issue that you may be experiencing. For example, if you wanted to see the logs of the **#docker** channel, you could find them here:

```
https://botbot.me/freenode/docker/
```

You can search for other channel archives on the following website:

```
https://botbot.me
```

CVE websites

In *Chapter 5*, *Monitoring and Reporting Docker Security Incidents*, we covered CVEs and Docker CVEs. A few things to remember about them are listed in the following:

- CVEs can be found at `https://cve.mitre.org/index.html`
- Docker-related ones can be found at `https://www.docker.com/docker-cve-database`
- To search for CVE's use the following URL: `https://cve.mitre.org/index.html`
- If you were to open this CVE from the preceding link, you will see that it gathers some information as shown in the following:
    - CVE ID
    - Description
    - References
    - Date entry created
    - Phase
    - Votes
    - Comments
    - Proposed

Other areas of interest

There are some areas of interest that you should keep in mind with regards to security. The Linux kernel, as we have talked about a lot during this module, is the key part of the Docker ecosystem. For this reason, it's very important to keep the kernel as up to date as possible. With regards to updates, it is also important to keep the Docker products you are using up to date too. Most updates include security updates, and for this reason, they should be updated when new product updates are released.

Twitter has become the social hotspot when you are looking to promote your products and Docker does the same. There are a few accounts that Docker operates for different purposes and they are listed in the following. Depending on what piece of Docker you are using, it would be wise to follow one or all of them, as shown in the following list:

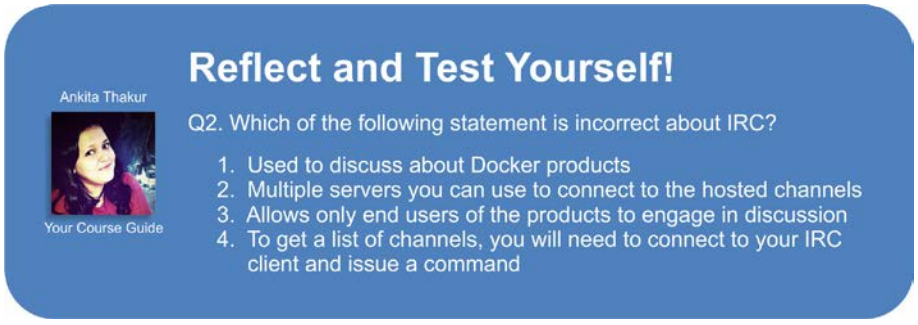
- **@docker**
- **@dockerstatus**
- **@dockerswarm**
- **@dockermachine**

Twitter also utilizes hashtags that group the tweets together, based on their hashtags. For Docker, it's the same and they use the #docker hashtag, which you can search for on Twitter to gather tweets that all talk about Docker.

The last thing we want to cover is Stack Overflow. Stack Overflow is a question and answer website and uses votes to promote the answers that are provided to help you get the best answer in the quickest manner. Stack Overflow utilizes a method similar to Twitter with tagging questions so that you can search for all the questions on a particular topic. The following is the link that you can use to gather all the Docker questions into one search:

```
http://stackoverflow.com/questions/tagged/docker
```

When you visit the URL, you will see a list of questions as well as how many votes each question has, number of answers, number of views, and a green check mark on some of them. The checked answers are the answers that the person who asked them mark as accepted, meaning that it's the best answer. Some of the people who monitor Docker questions are those that work for Docker, doing the work behind the scenes and providing the best answers, therefore, it's a great place to pose any questions that you might have.



Your Coding Challenge

Ankita Thakur

Your Course Guide

Here are some practice questions for you to check whether you have understood the concepts:

- What are the multiple ways that you can obtain or keep up to date about the information related to the security issues that may occur in Docker products?
- Which are the two mailing list that Docker operates on?

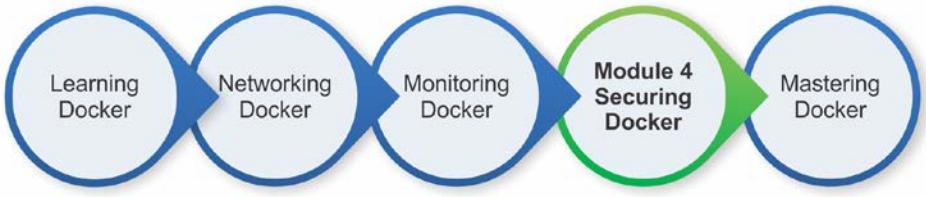
Summary of Module 4 Chapter 8

Ankita Thakur

Your Course Guide

In this chapter, we looked at how to keep up with security-related issues that not only pertain to Docker products that you may be running now or in the near future, but they also pertain to security issues such as kernel issues. As Docker relies on the kernel for all Docker containers on a Docker host, the kernel is very important. We looked at multiple mailing lists that you can sign up for, getting notifications in this manner. Joining IRC chat rooms and following GitHub issues for anything securityrelated or anything that isn't currently working might affect your environments. It is very important to always keep security in the front of your mind when deploying anything and while the Docker is inherently secure, there are always people out there that will take advantage of any given vulnerability, therefore, keep all of your environments safe and as up to date as possible.

Your Progress through the Course So Far



Course Module 5
Mastering Docker

Course Module 5
Mastering Docker

Ankita Thakur

Your Course Guide

Explore Docker more with Course Module 5, Mastering Docker

Course Module 5

Firstly, well done for doing well so far! I'm sure now you recognize Docker's importance for innovation in everything from system administration to web development. As the name suggest, we'll see some advanced concepts of Docker in this last module.

Now that you have all the tools in your arsenal, it's time to deploy Docker in your production environment. But how do we go about doing this? We'll 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.

Ankita Thakur

Your Course Guide

Then, we'll focus on three GUI applications (Shipyard, Panamax, and Tutum) 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 Shipyard, Panamax, Tutum UI. You will be able to see the benefits of using such a GUI to help manage your environment. This will leave you with the ability to evaluate which GUI is right for your needs.

Finally, you'll learn how to scale your environments using discovery services. We'll also look at the common issues that are faced as well as the solutions to fix them. We'll look at the Docker APIs that are out there and how to tie into them and use them to our advantage. We'll look at how we can contribute to Docker. If we can't contribute to the code, how we can help otherwise.

Sound interesting? Let's dive in!

1
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 module 1, 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

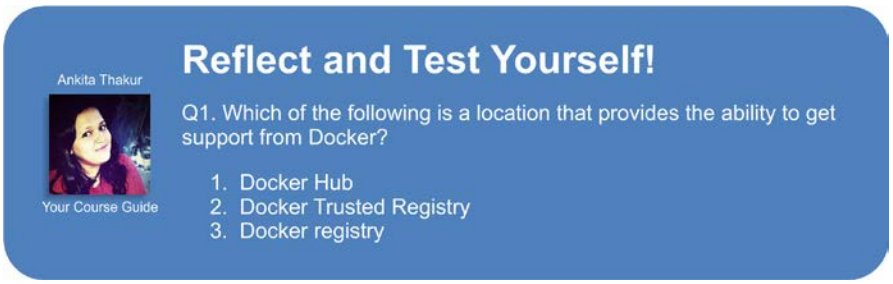
Docker Compose also allows you to scale containers that are located in the `docker-compose.yml` file. For example, let's say our Compose file looks as follows:

```
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:

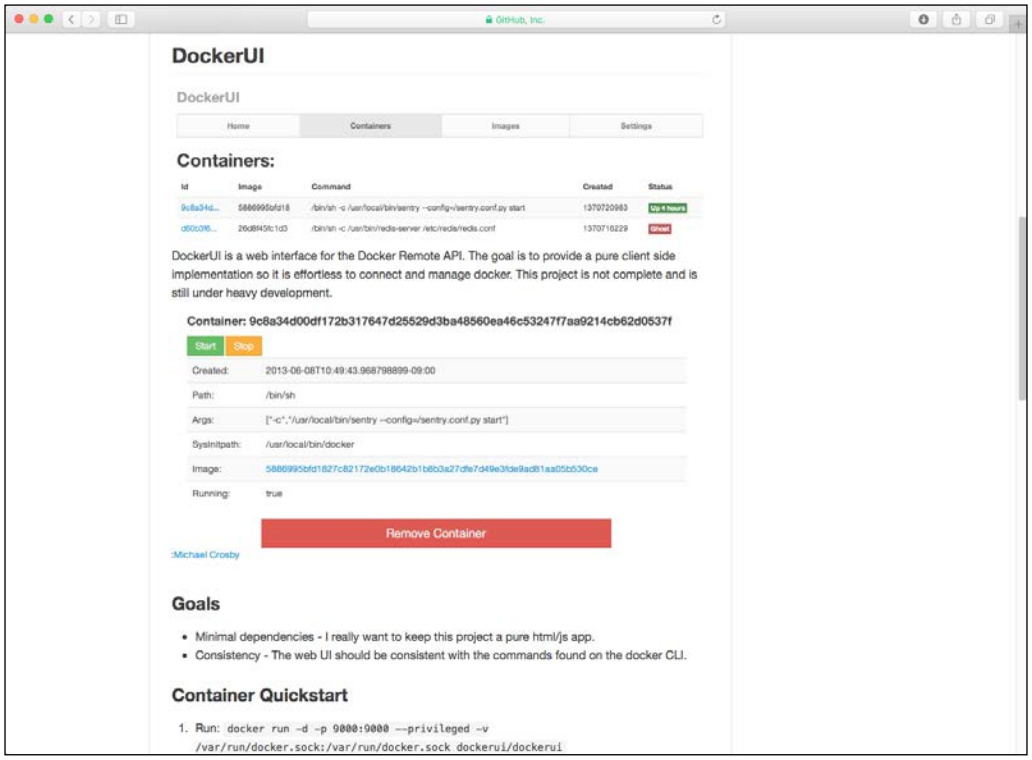
  ```
  $ docker run -d -v /opt/uploads:ro nginx
  $ docker run -d --volumes-from data:ro nginx
  ```

- Remember to utilize the Docker security benchmark application to help tune your environments.
- 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.

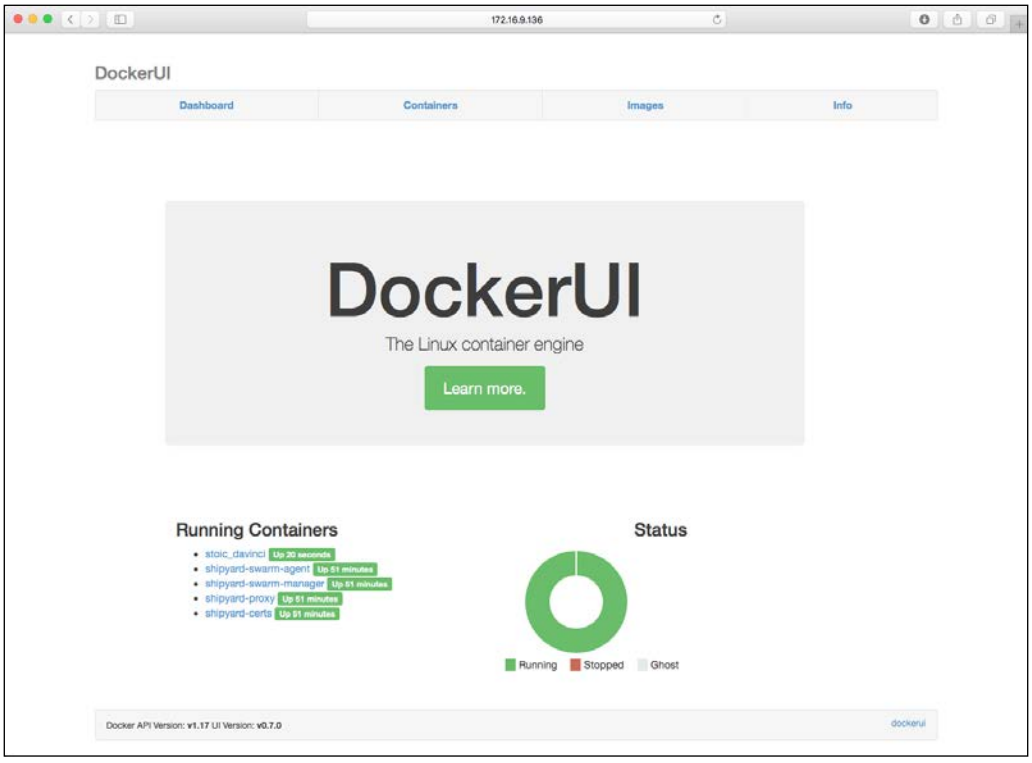


This is a screenshot of the GitHub repository, where the code for DockerUI is kept. You can view the content yourself by visiting `https://github.com/crosbymichael/dockerui`.

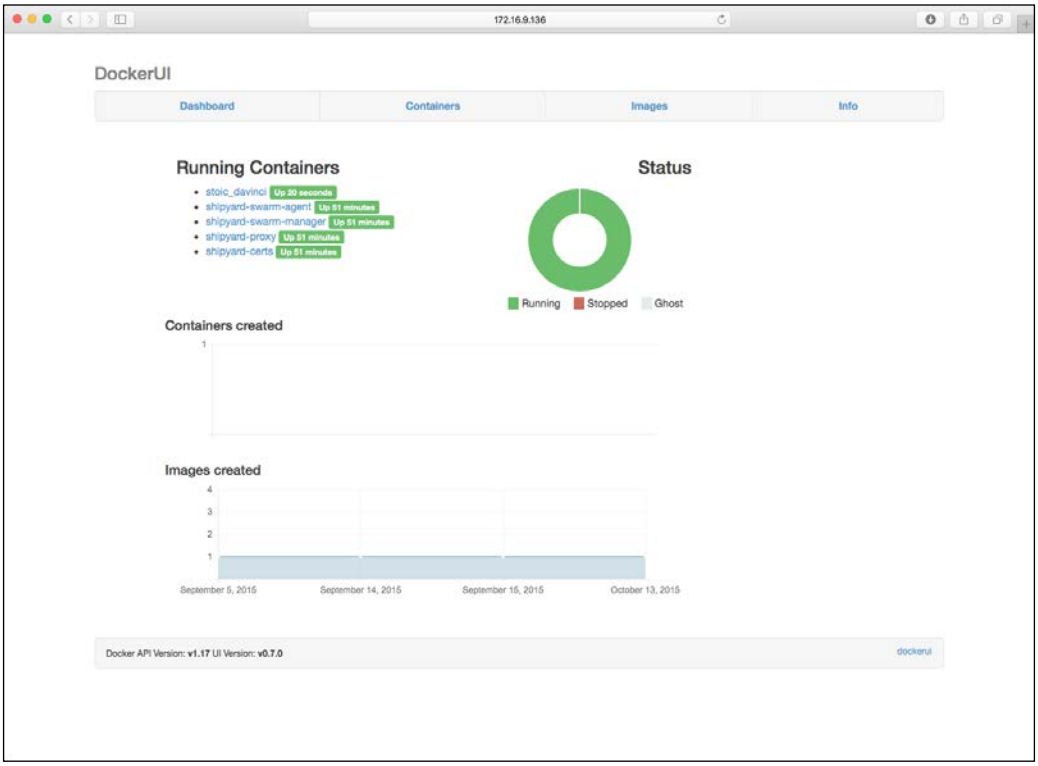
This page will include screenshots of DockerUI in action as well as the current features of DockerUI that are available. You can create pull requests against the code if you have ideas you would like to see in DockerUI and would like to help contribute to the code. You can also submit issues that you might find with DockerUI.

The installation of DockerUI is very straightforward with you just running a simple Docker run command to get started:

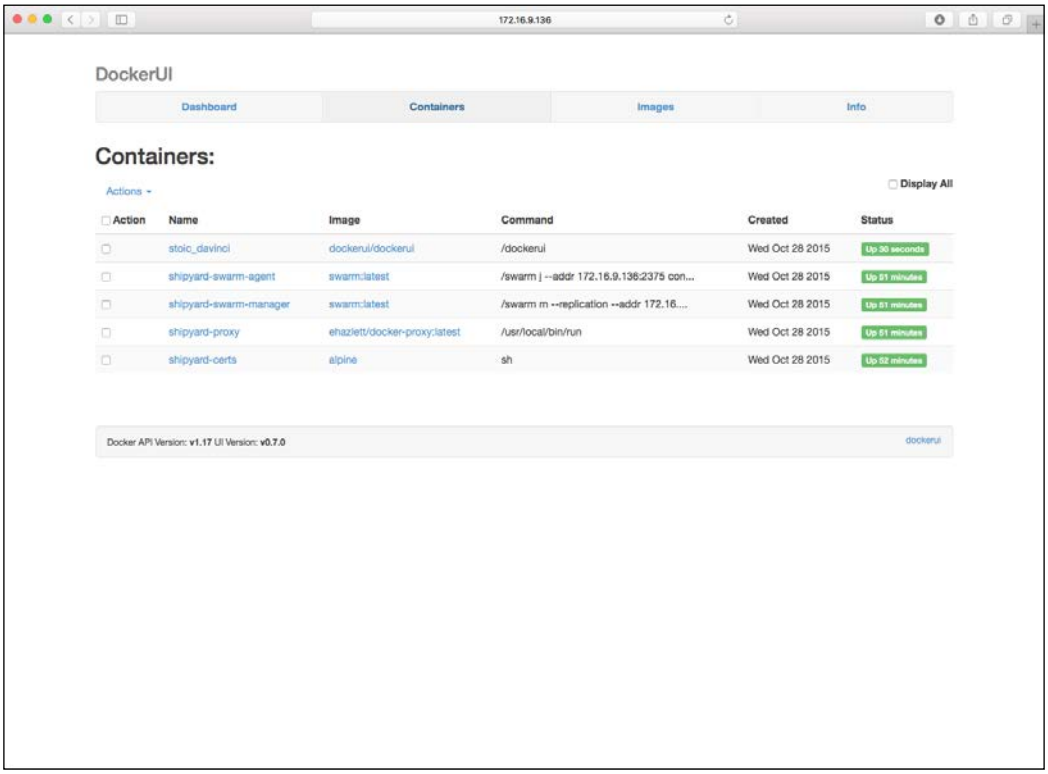
```
$ docker run -d -p 9000:9000 --privileged -v /var/run/docker.sock:/var/
run/docker.sock dockerui/dockerui
```



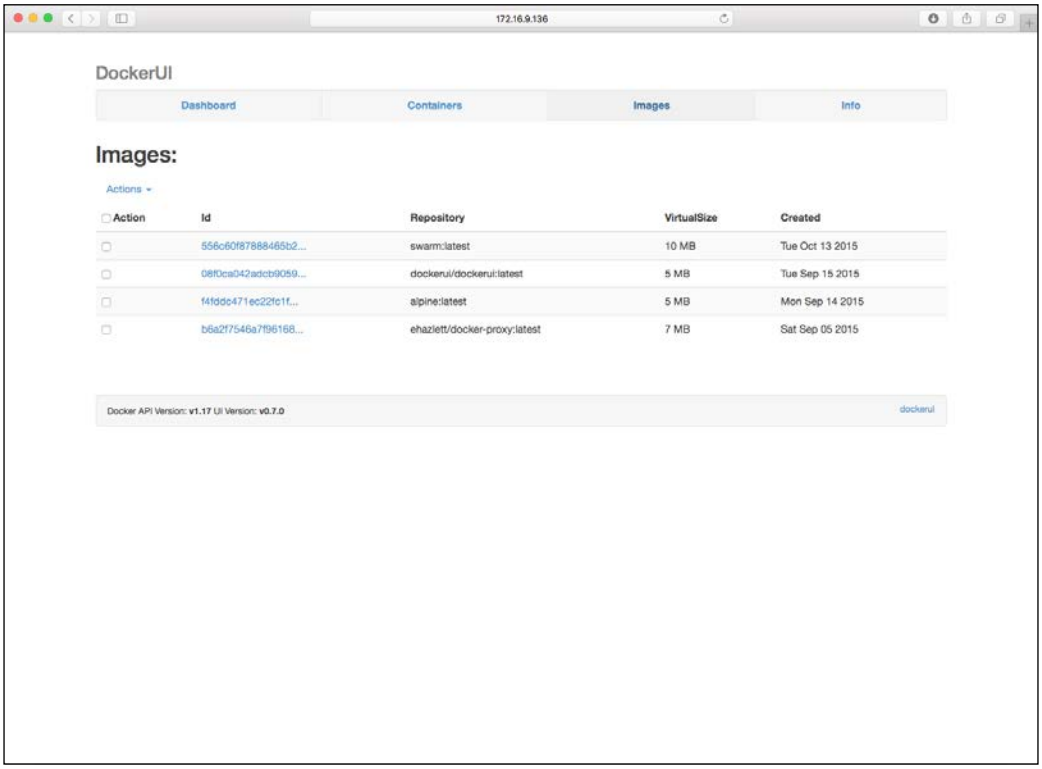
After you have run the previous command, you will be able to navigate to the DockerUI web interface. You should be able to easily break down the run command and see what it is doing and where you need to go to get to the dashboard. However, in case you are stumped, here is what the command is doing: 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.



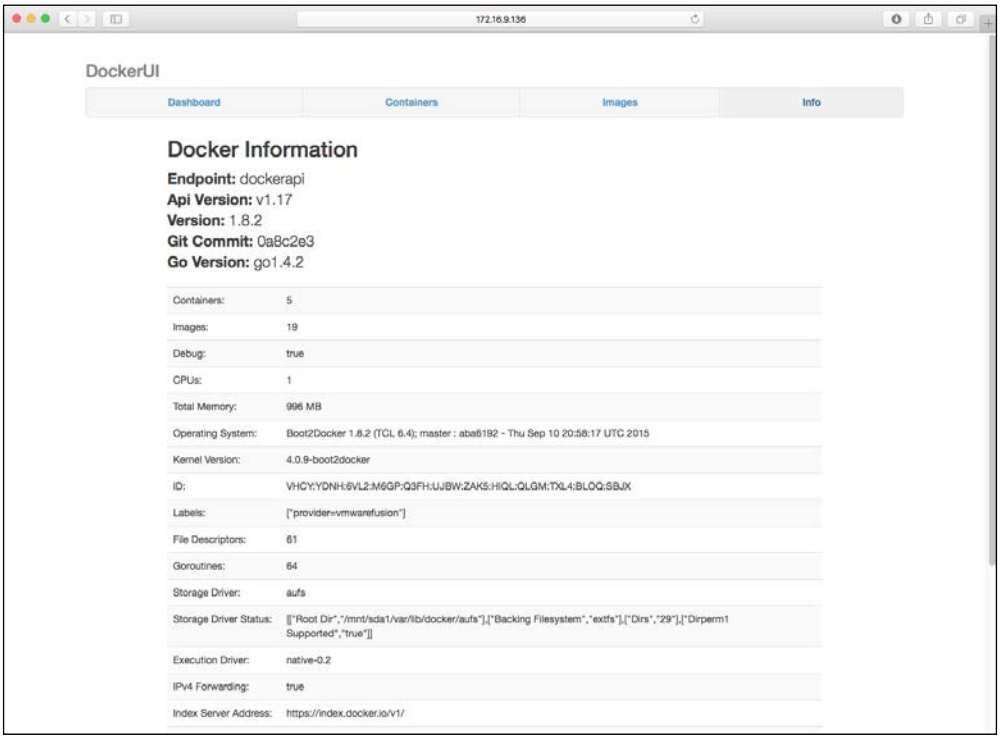
This is another view of the dashboard shortly after you have launched the container and visited the web interface. You can see information such as what containers are currently running on your Docker host and what their statuses are; some could be stopped as well. It will also show you the containers 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.



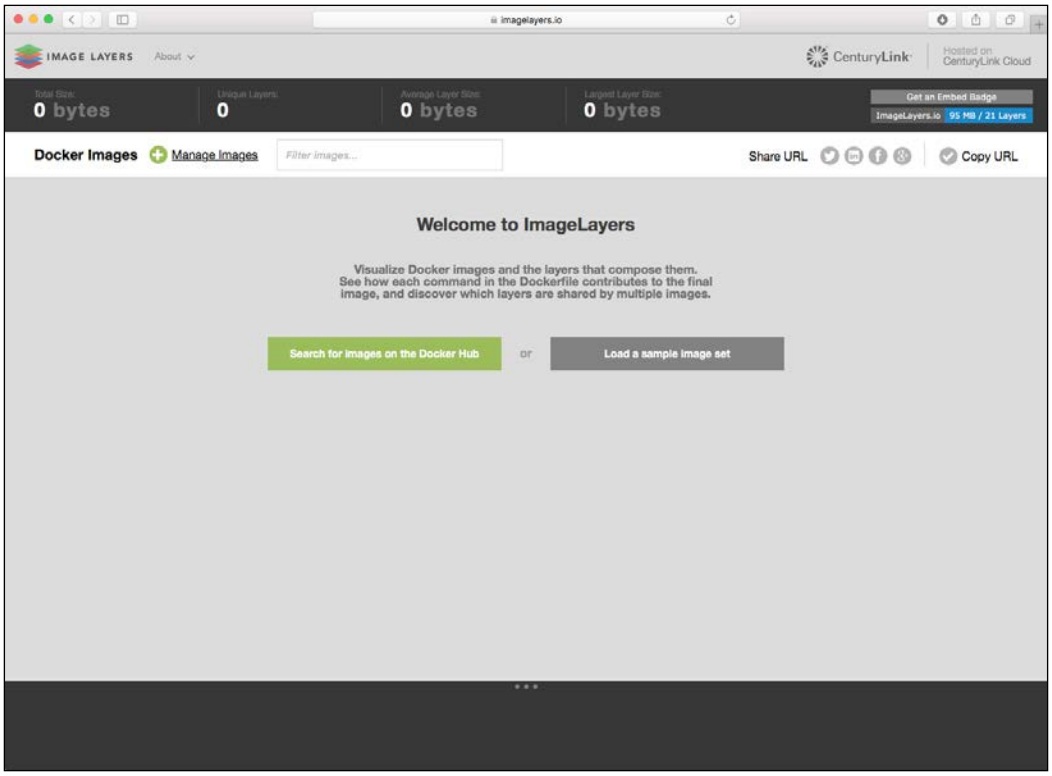
Next up in the navigation bar is **Images**. Again, like **Containers**, you can get all the information on all the images being used on your Docker host here. Information such as their IDs, what repositories they are from, their virtual sizes, and when they were created will be displayed here. Again, 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.

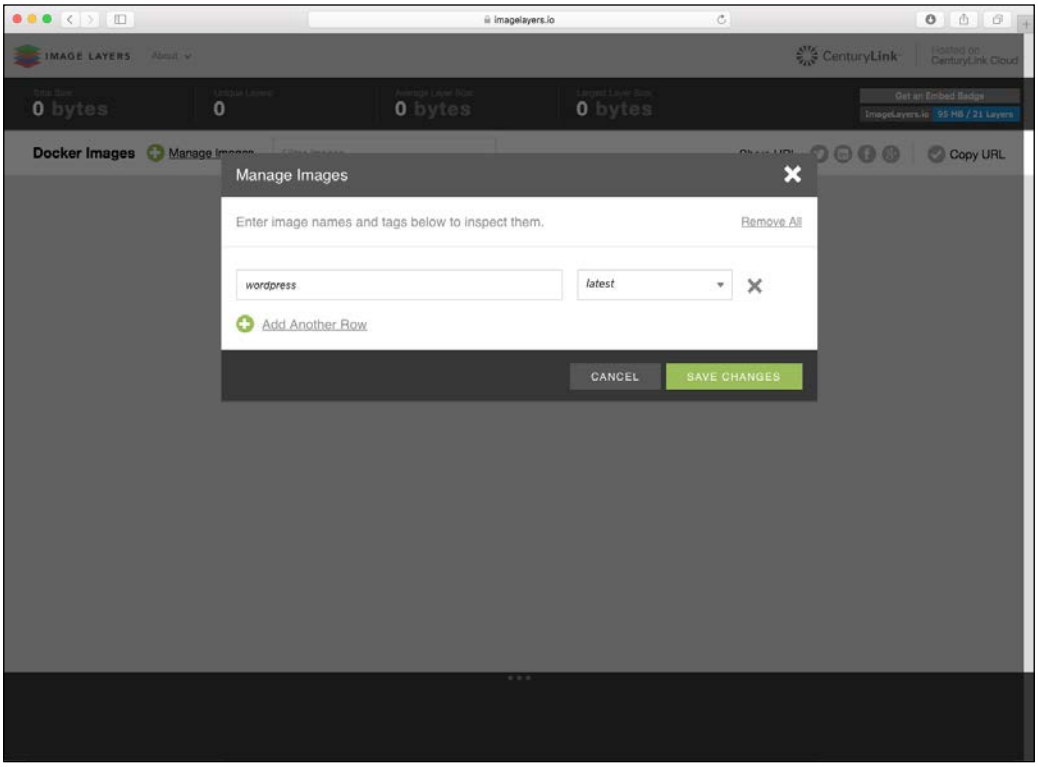
ImageLayers

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.

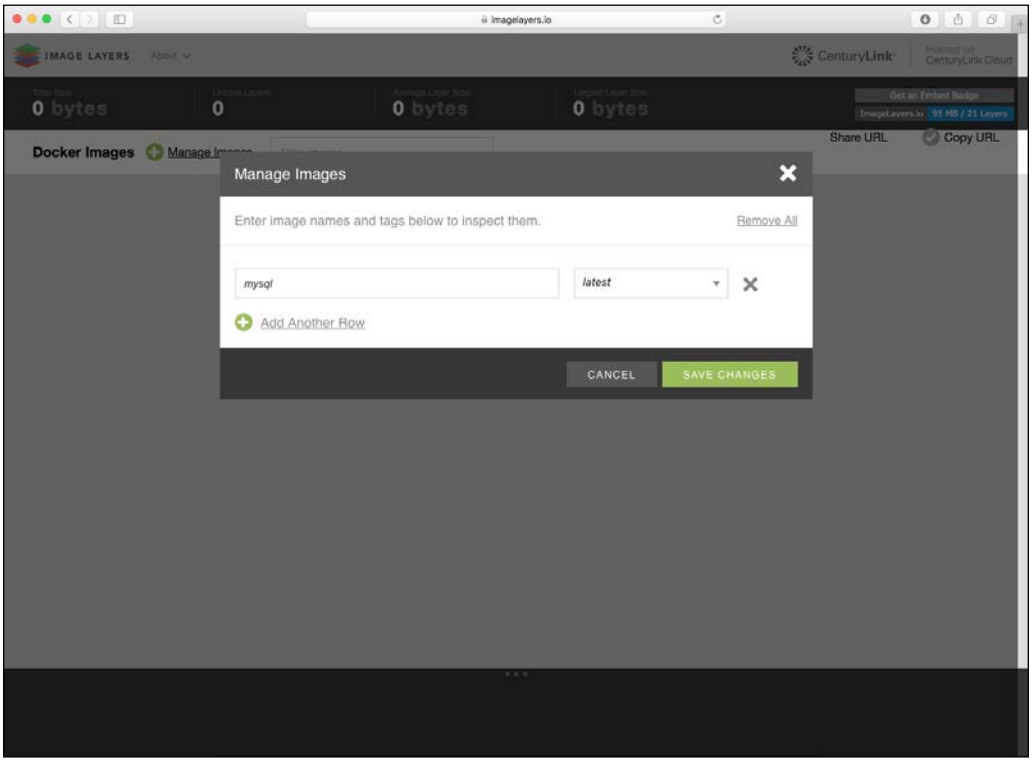


This screenshot is what you will be presented with while navigating to the ImageLayers website: `https://imagelayers.io`.

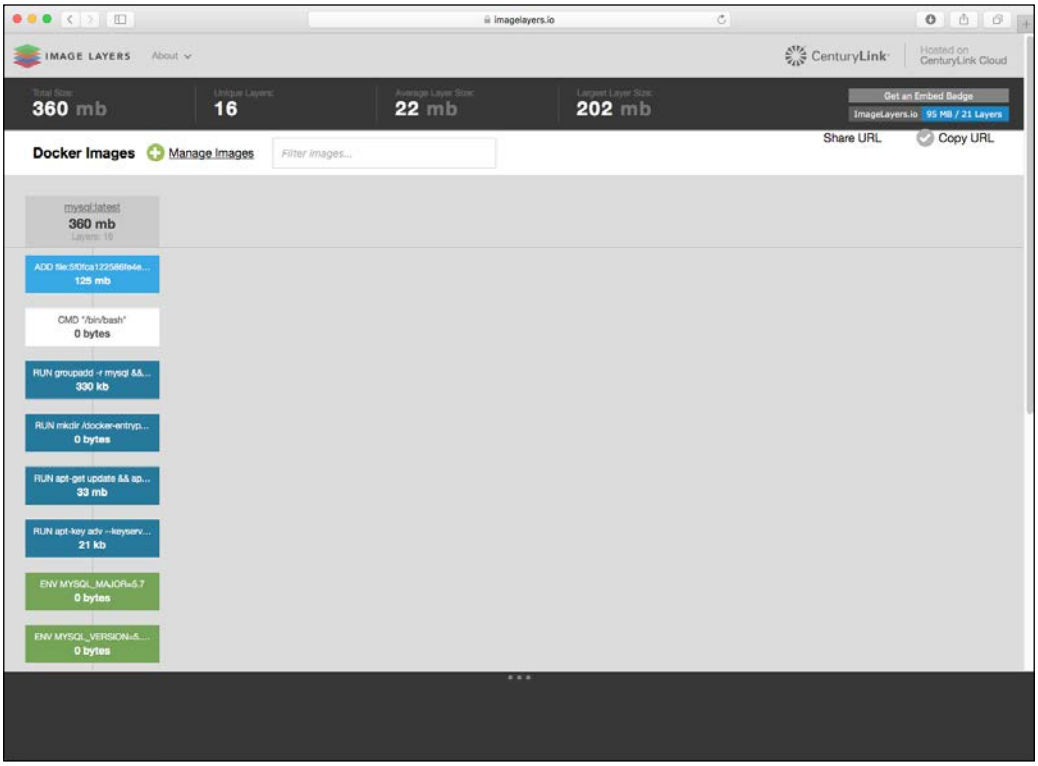
You can search for images that are on Docker Hub to have ImageLayers provide information on the image back to you. Or, you can load up a sample image set if you are looking at providing some sample sets or seeing some more complex setups.



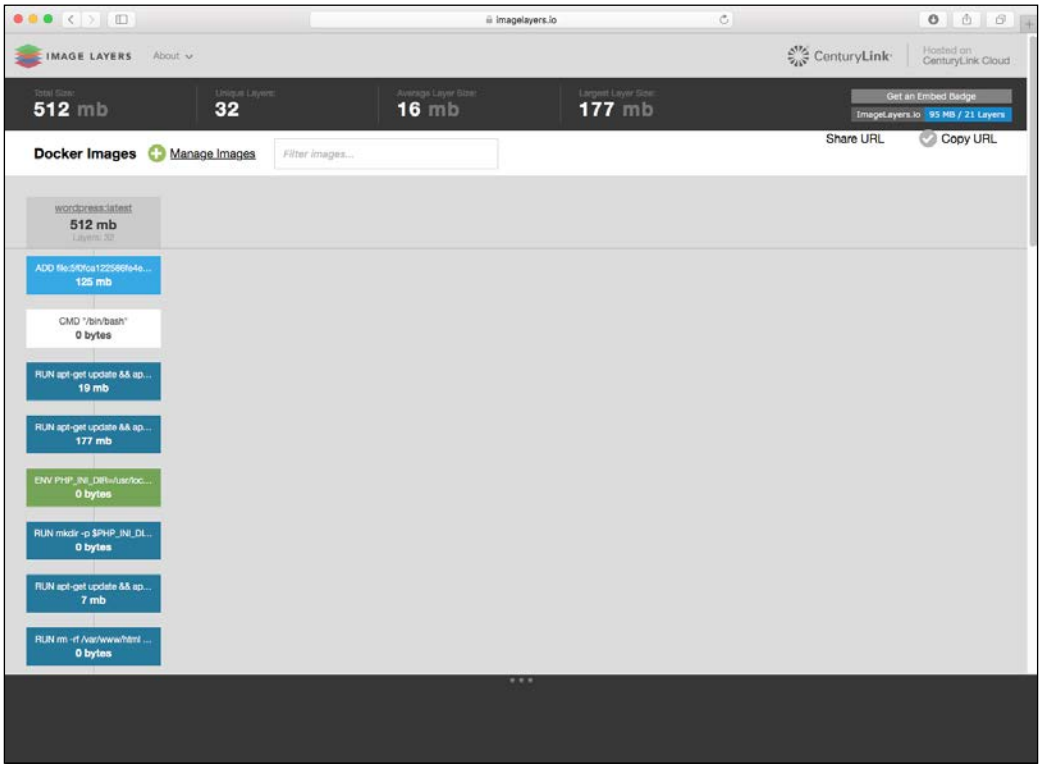
In this example, we are going to search for the `wordpress` image and select the **latest** tag. Now, you can search for any image and it will do auto-complete. Then, you can select the appropriate tag you wish to use. This could be useful if you have, say, a staging tag and are thinking of pushing a new image to your **latest** tag, but you want to see what impact it has on the size of the image.



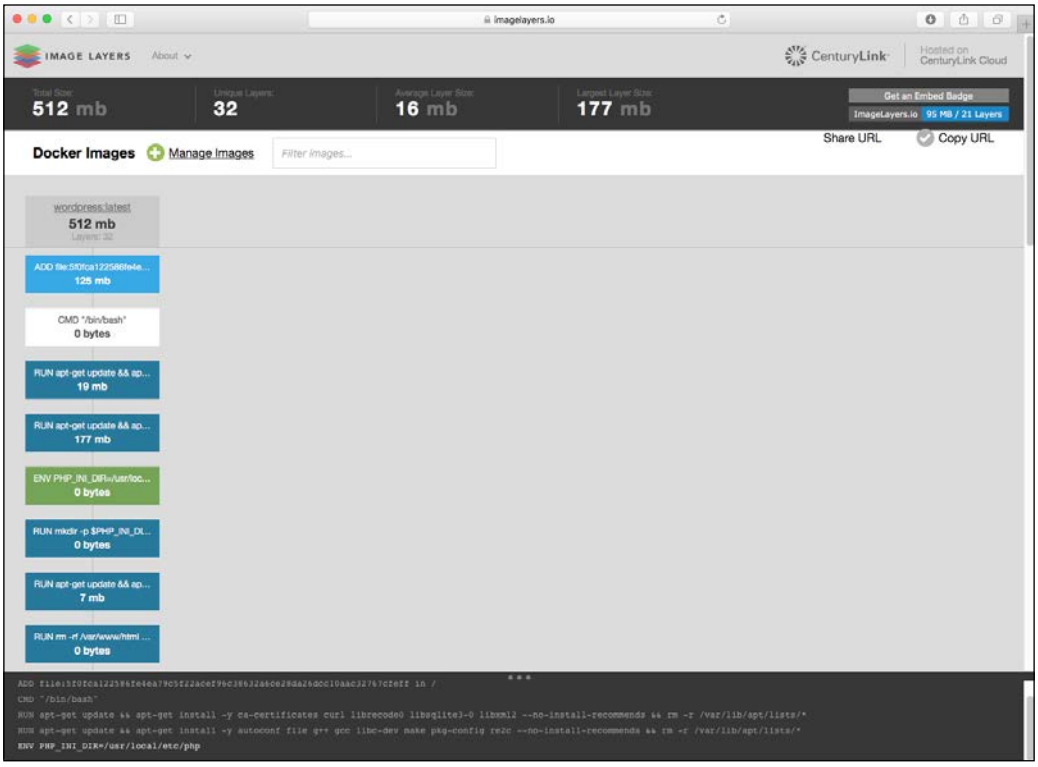
So, let's walk through an example. In this example, we are going to select a `mysql` image and the **latest** tag. We will use this since it is a common image that most people will use at some point in their Docker experience.



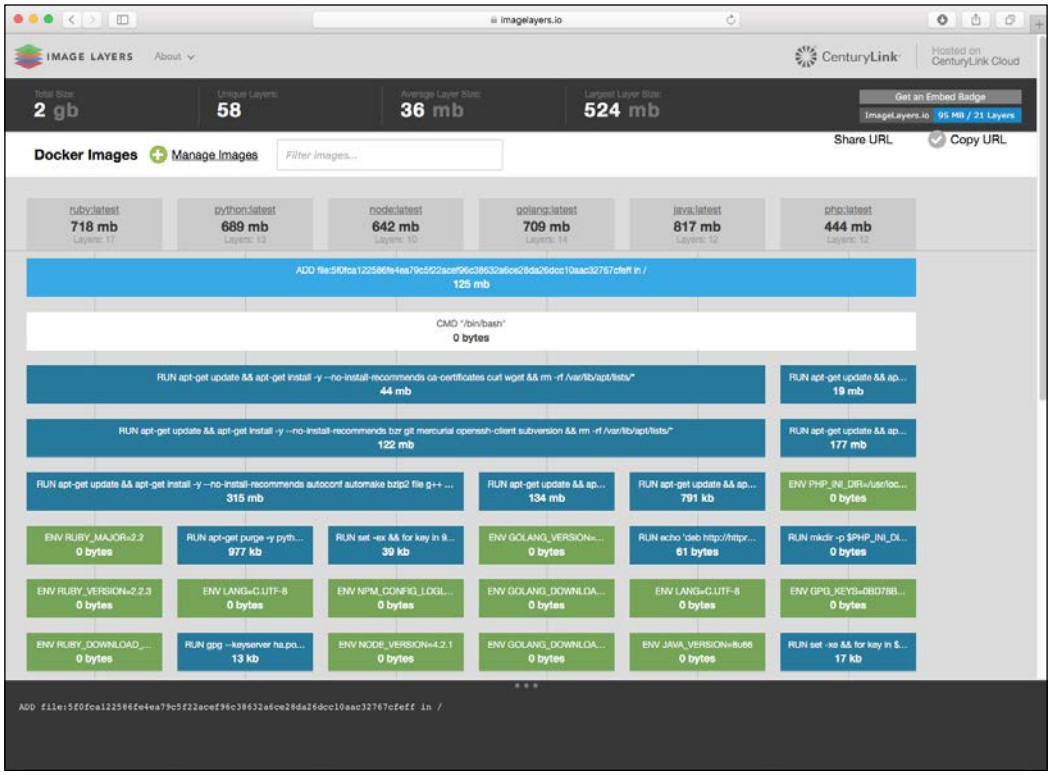
Once we click on **Save Changes** from the previous item, we will be shown something similar to the preceding screenshot (now, this will vary depending upon the image you have selected in your search). This displays some information at the top, such as the total image size, unique layers, the average 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 of Module 5 Chapter 1

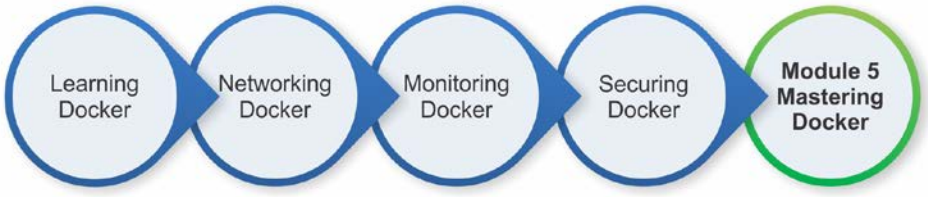
Ankita Thakur

Your Course Guide

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!

Your Progress through the Course So Far



2
Shipyard

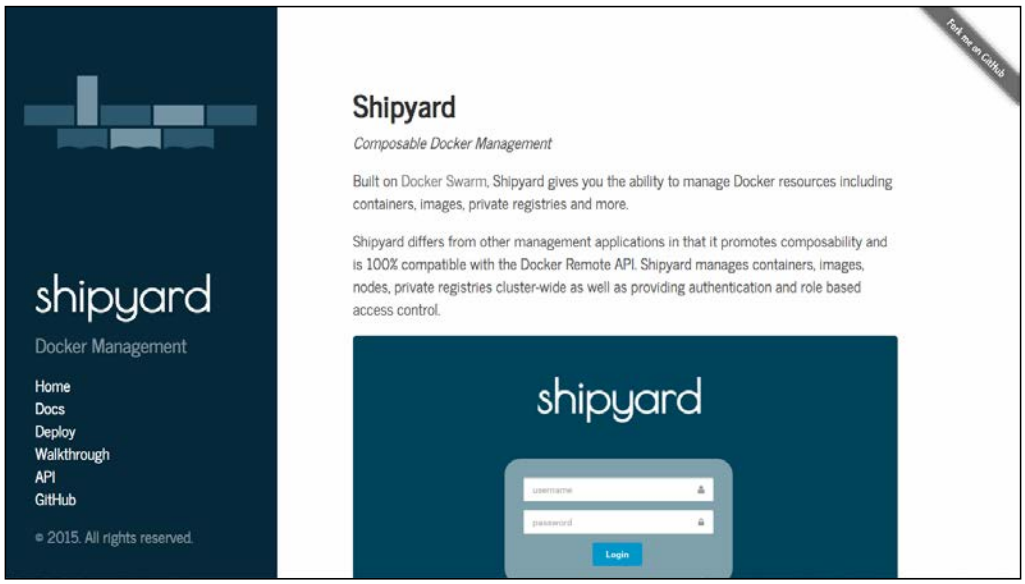
In this chapter, we will take a look at Shipyard. Shipyard is a tool that allows you to manage Docker resources from a web UI or a GUI interface.

The topics that will be covered are:

- Starting Shipyard
- The components of Shipyard

Up and running

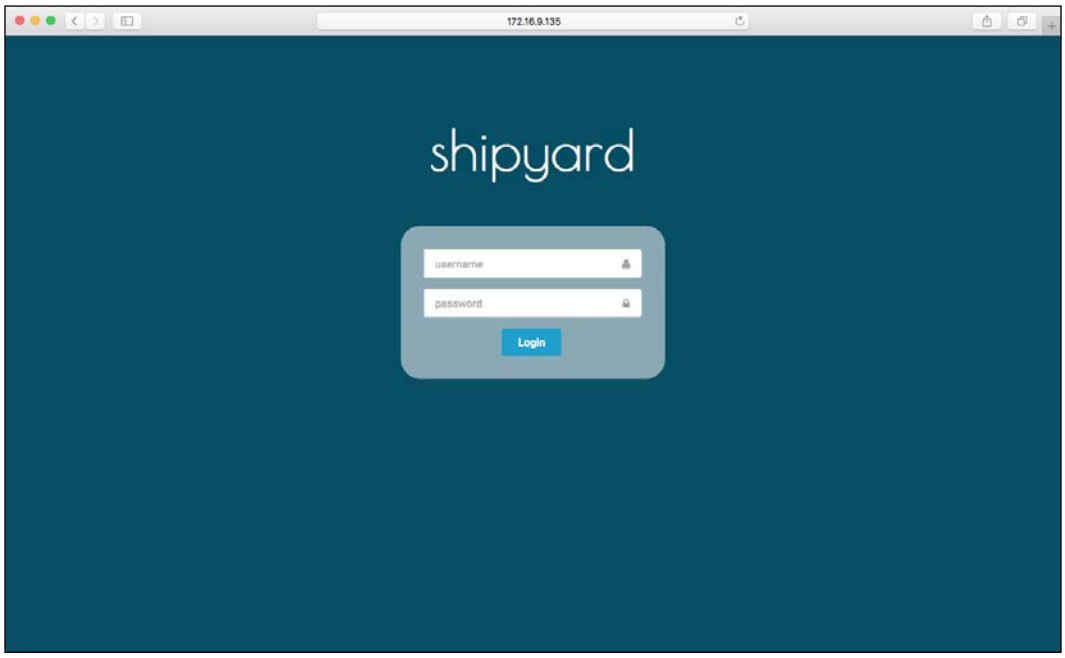
You will see a screen similar to the following screenshot while navigating your browser to the Shipyard website at `https://shipyard-project.com`:



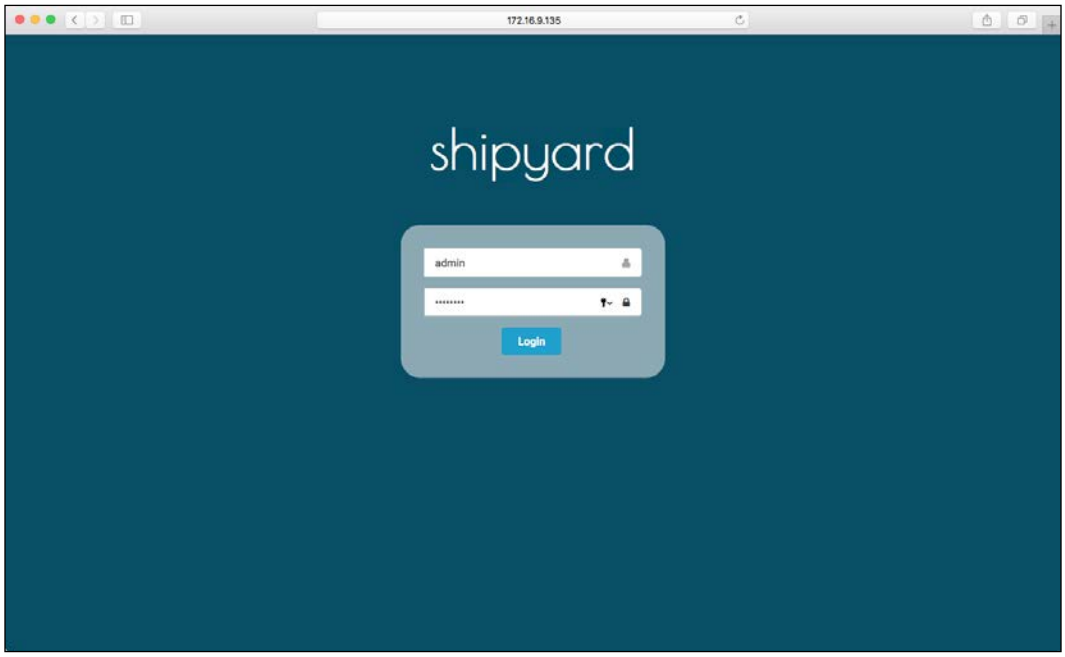
First, we need to get Shipyard up and running. To do this, we will execute the following commands:

```
$ 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:



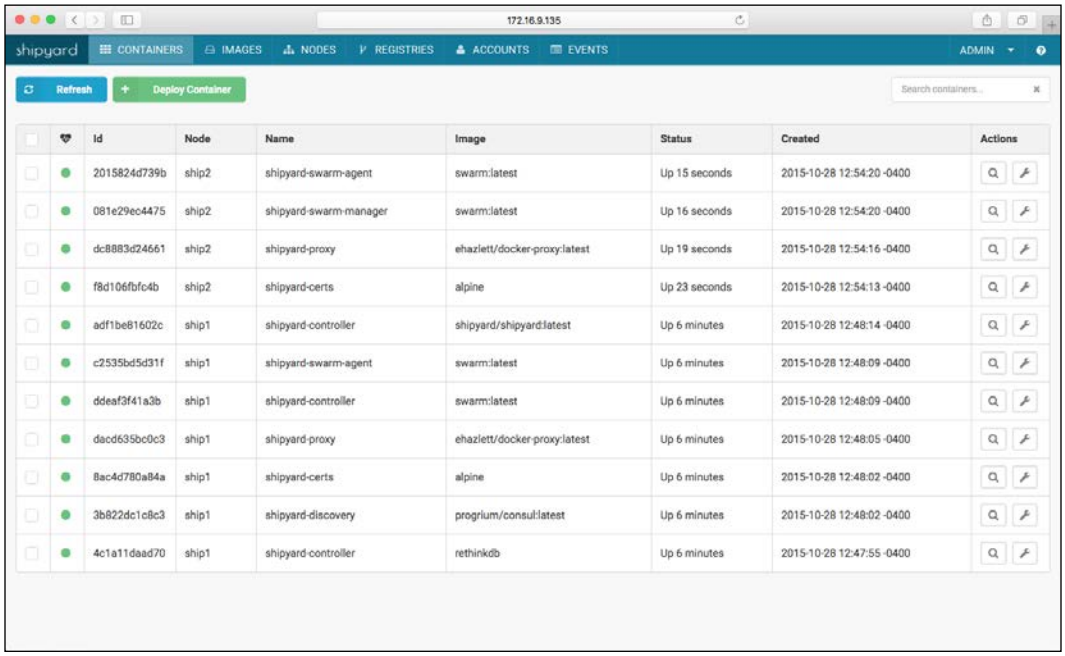
The URL is always the IP address of your Docker host. It runs on port `8080` (that is, `172.16.9.135:8080`).



The default username is `admin`. The default password is `shipyard`. Enter these details and click on **Login**.

Containers

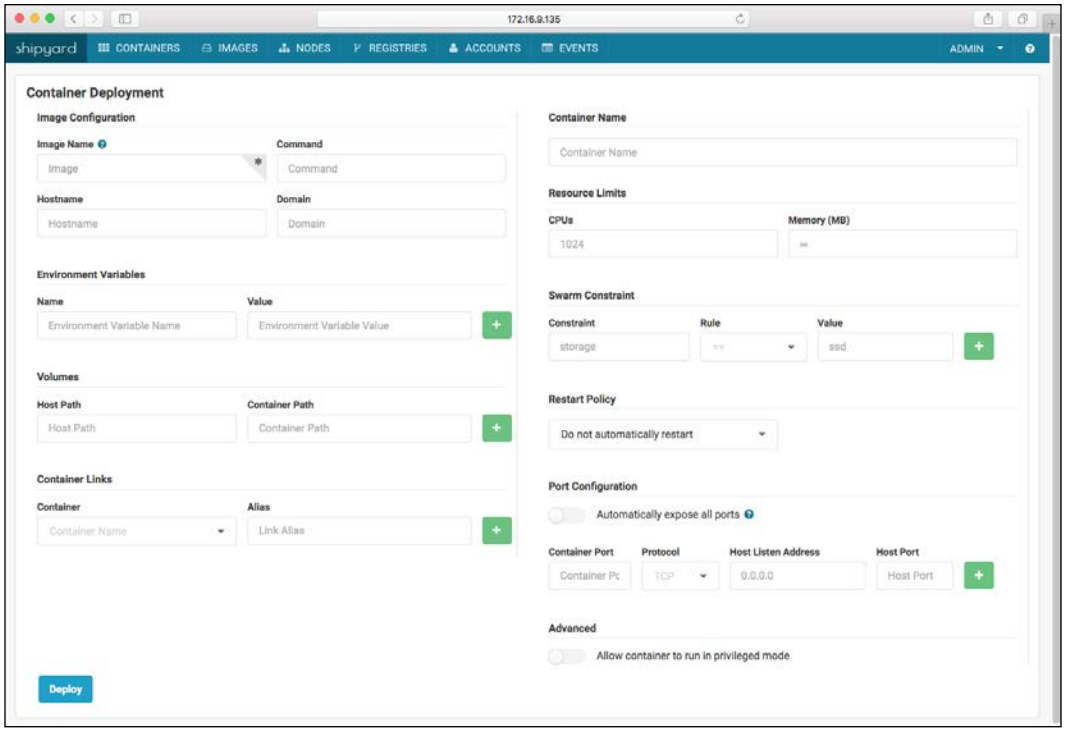
After logging in, you will be taken to the main dashboard or the **CONTAINERS** section as follows:



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

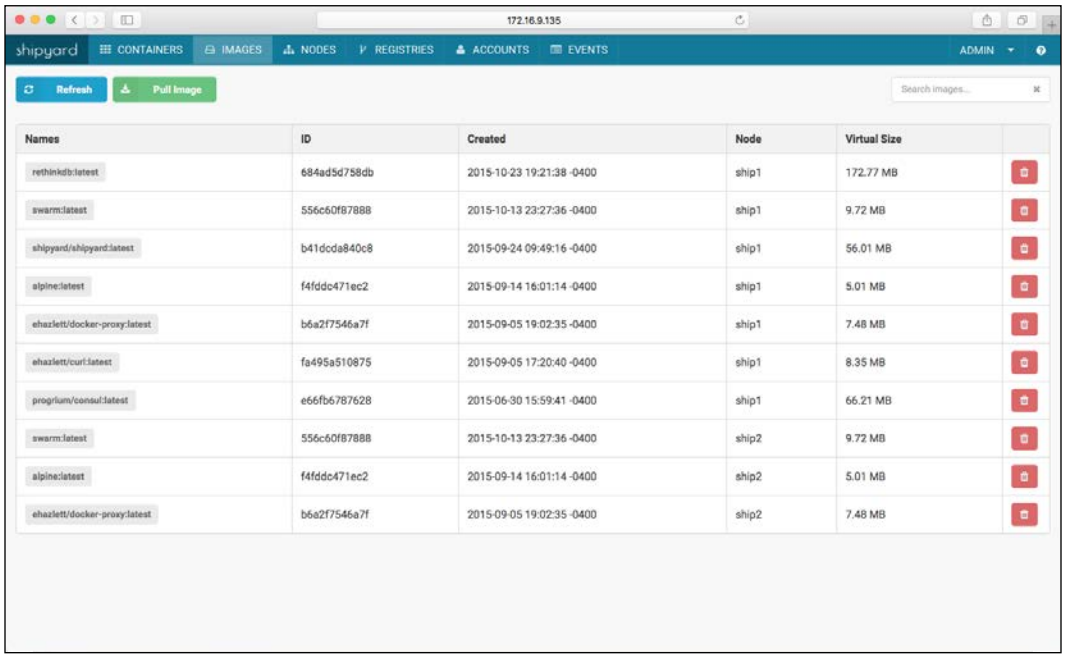
The first thing we will tackle on this page is the **Deploy Container** button.



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

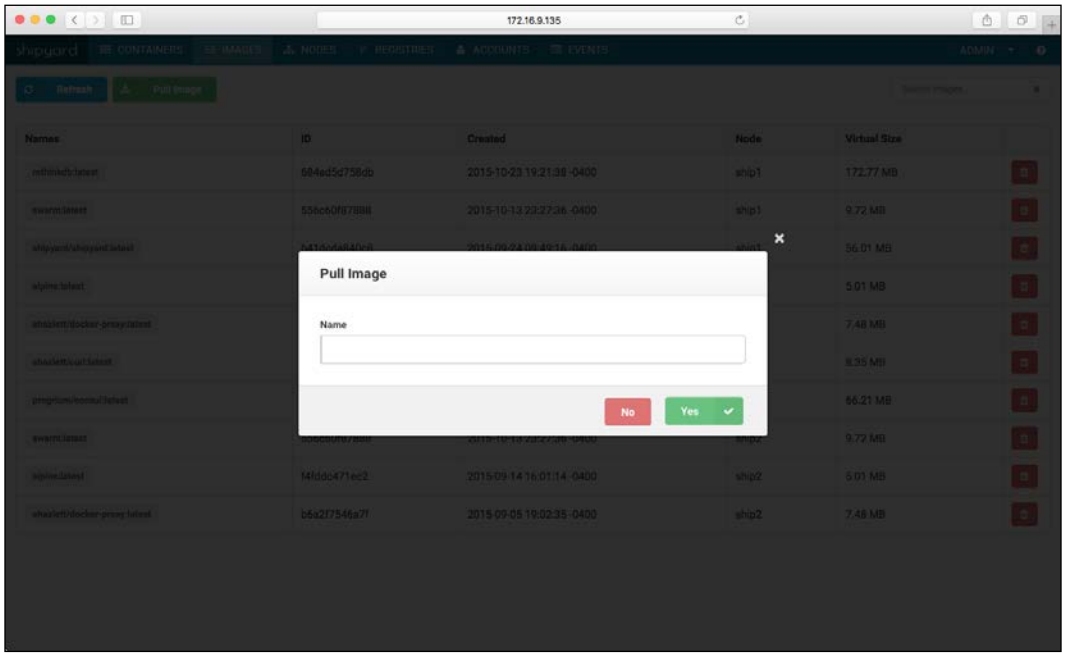
At the top of the screen, we can see a blue navigation bar. Moving on from the **CONTAINERS** section (for now), we will now cover the **IMAGES** section. In the **IMAGES** section, we can see all the images that are being used across our hosts.



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

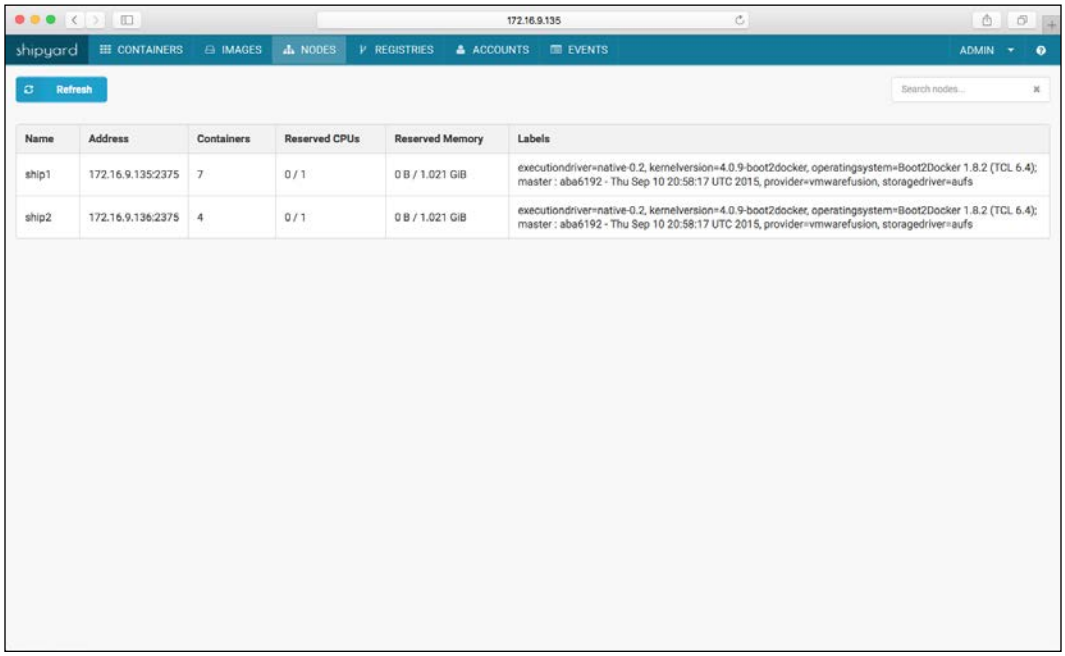
Now, one thing that we didn't cover was the **Pull Image** button. By clicking on this, you will be presented with the following screen:



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

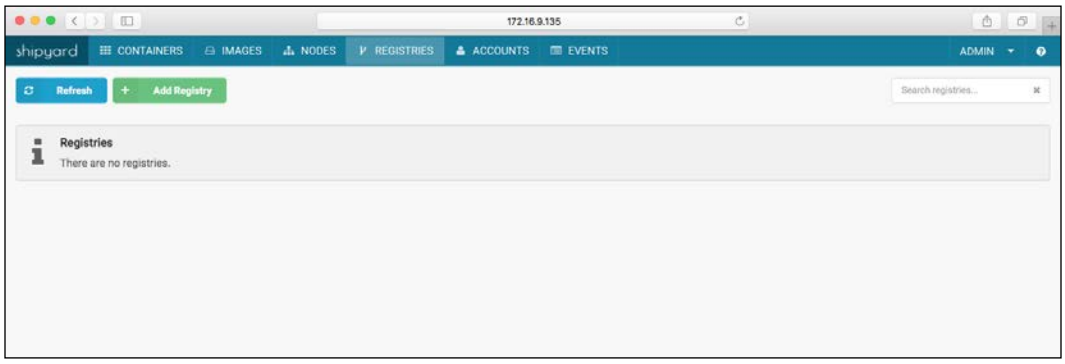
Next up is the **NODES** section. This section shows information on what nodes or Docker hosts you have connected to Shipyard.



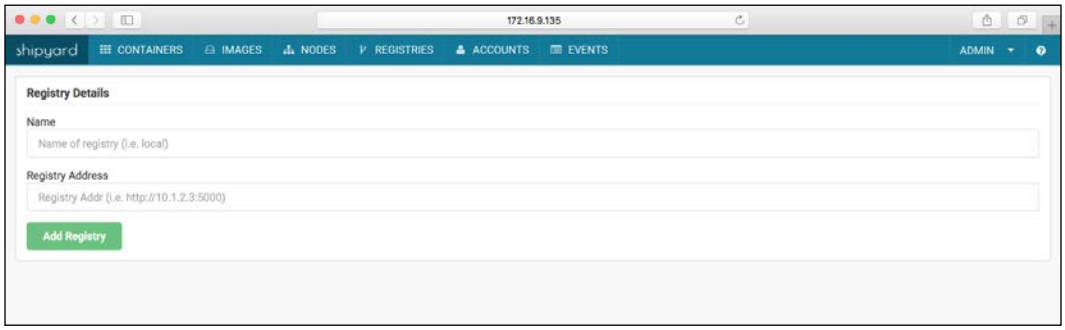
It will give you information such as the name of the node, its IP address, the number of reserved CPUs and memory, as well as the labels that provide information such as what version of the Linux kernel or Docker is being used.

REGISTRIES

Next up is the **REGISTRIES** tab. This is where you can add registries beyond Docker Hub.



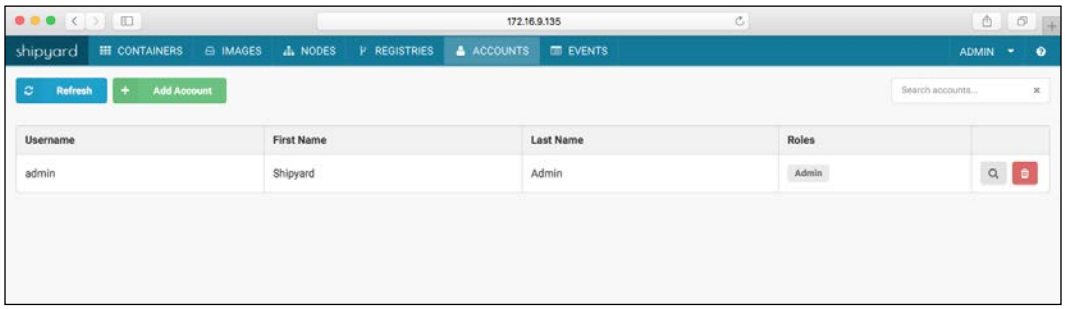
On clicking the **Add Registry** button, you will be taken to the following screen:



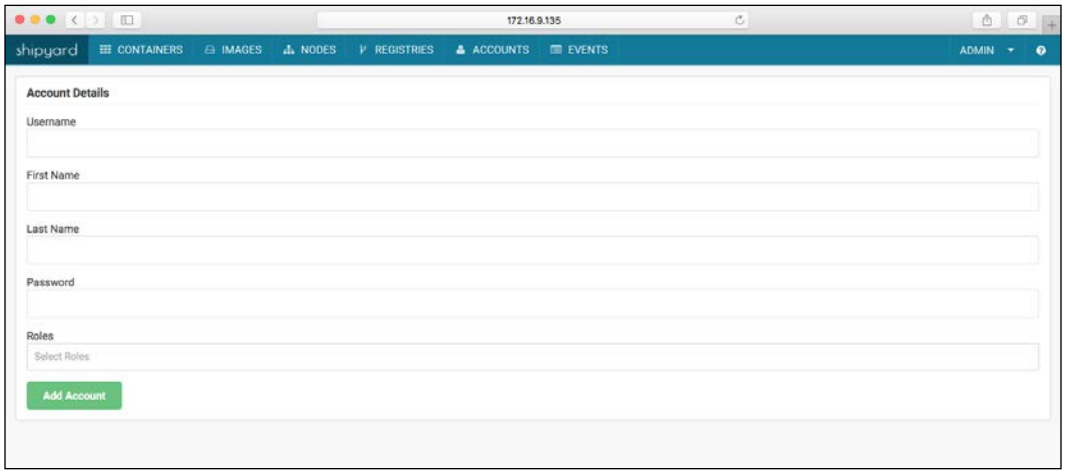
This will allow you to enter information about the registry such as its name and registry address, which would include the IP address or the DNS name and the port it is running on.

ACCOUNTS

Next up is the **ACCOUNTS** tab where—you guessed it—you can add or remove accounts.



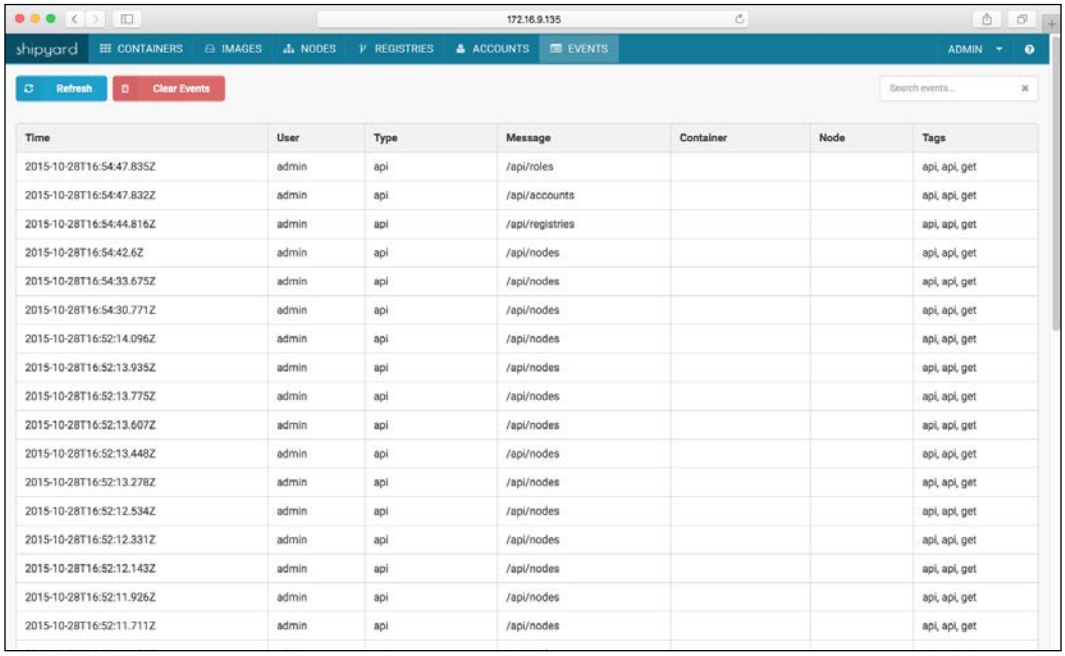
In the following screenshot, you can see what information is needed when you add a new account:



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

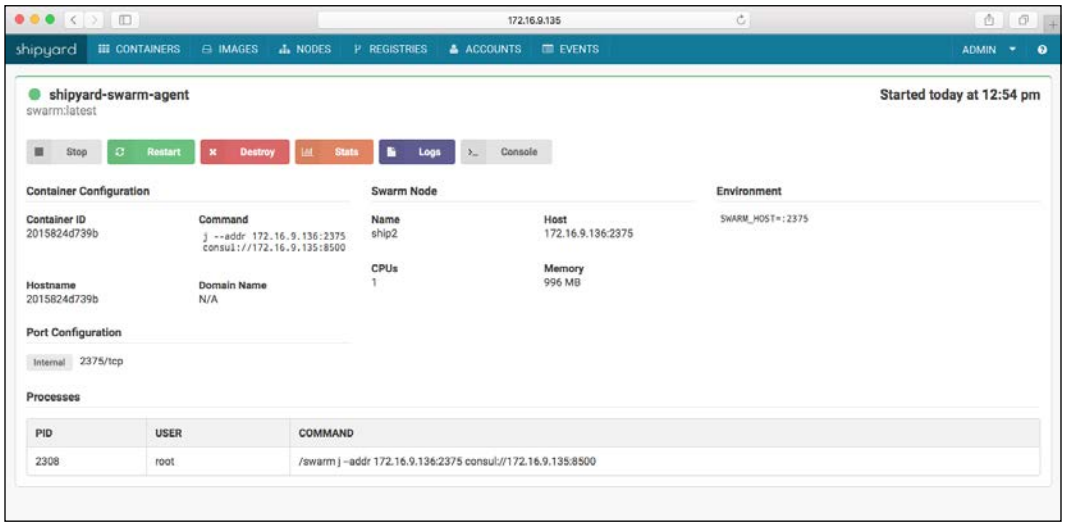
Okay, last up is the **EVENTS** tab that will display the following screen:



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.

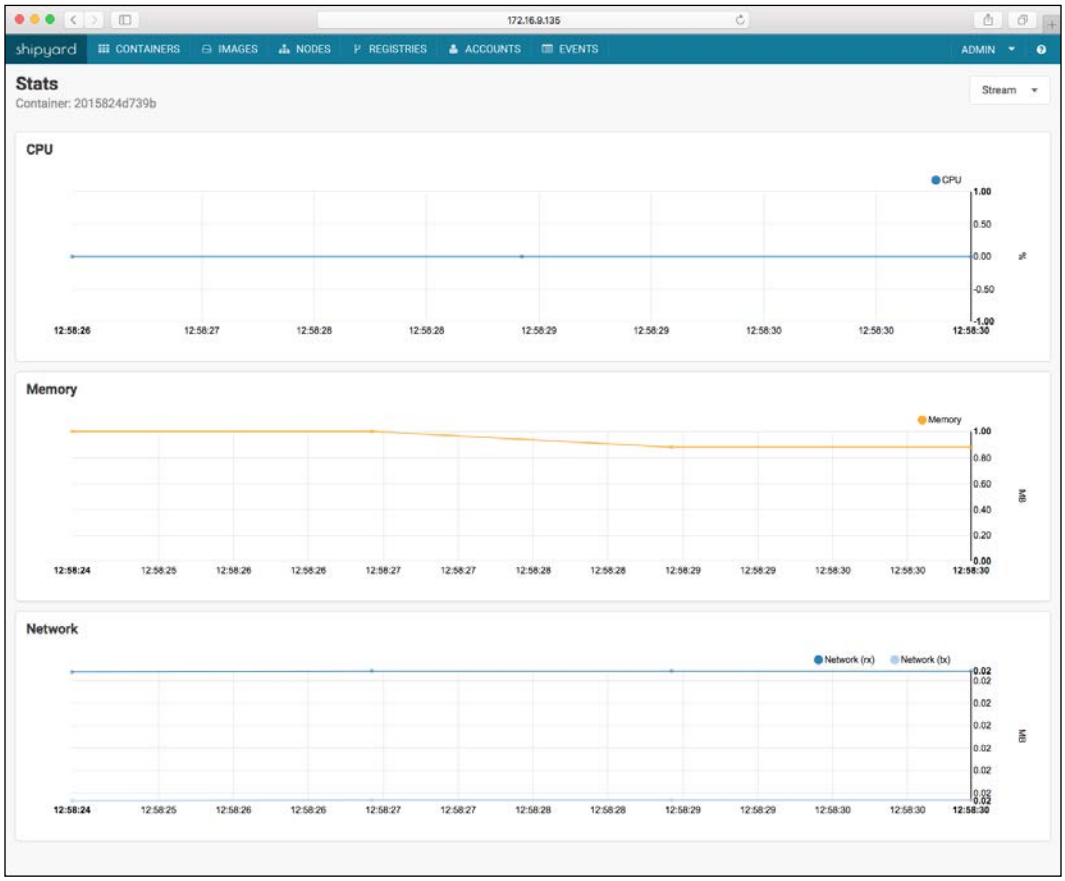
Back to CONTAINERS

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:

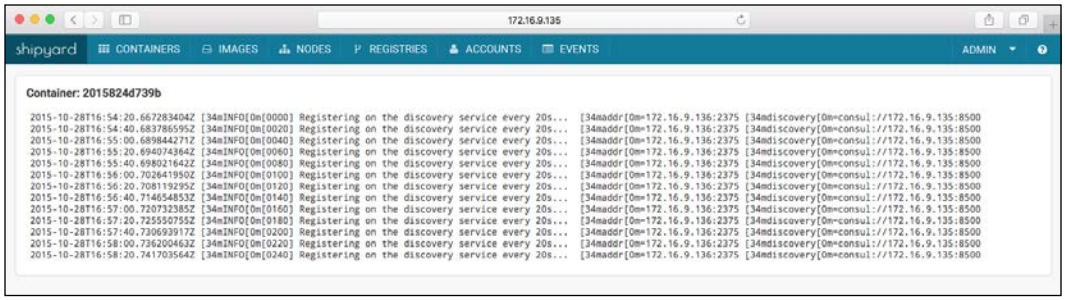


We can then get information on that running container and manipulate it. We can stop, restart, or destroy (or remove) it. We can also see information on it such as the command that it's running, its port, its IP address, and its node name.

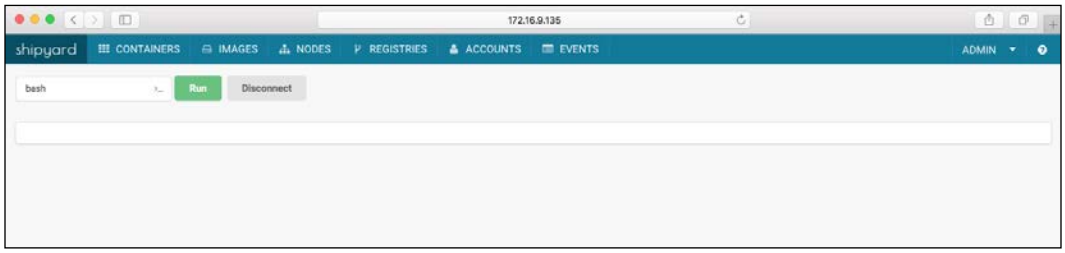
Clicking on the **Stats** button, we can see information pertaining to the running container such as the CPU, memory, and network information.



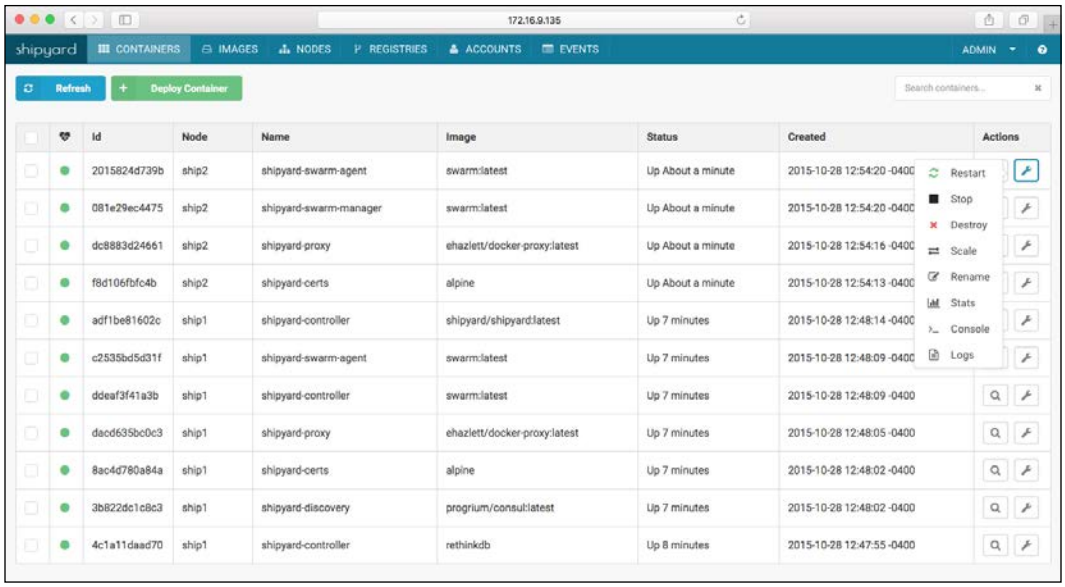
Clicking on the **Logs** button will show you everything that is going on with the container. In this case, the container is polling consult for new information ever so often.



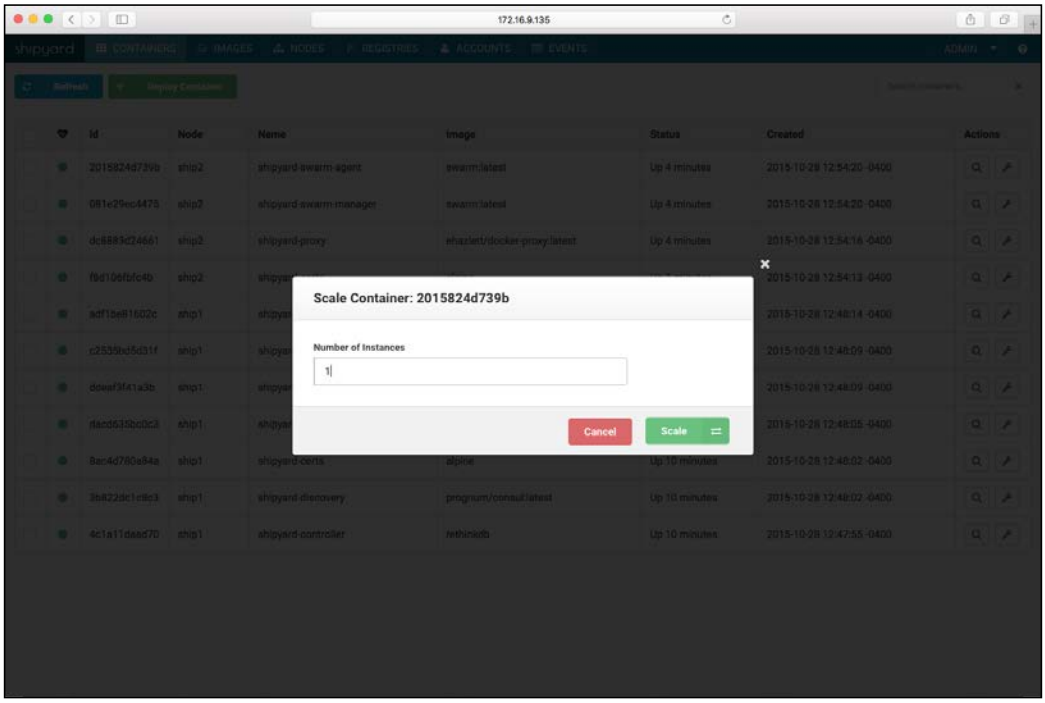
Now, the **Console** button is interesting. It will allow you to actually run a command against the container and provide the output from that command.



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.

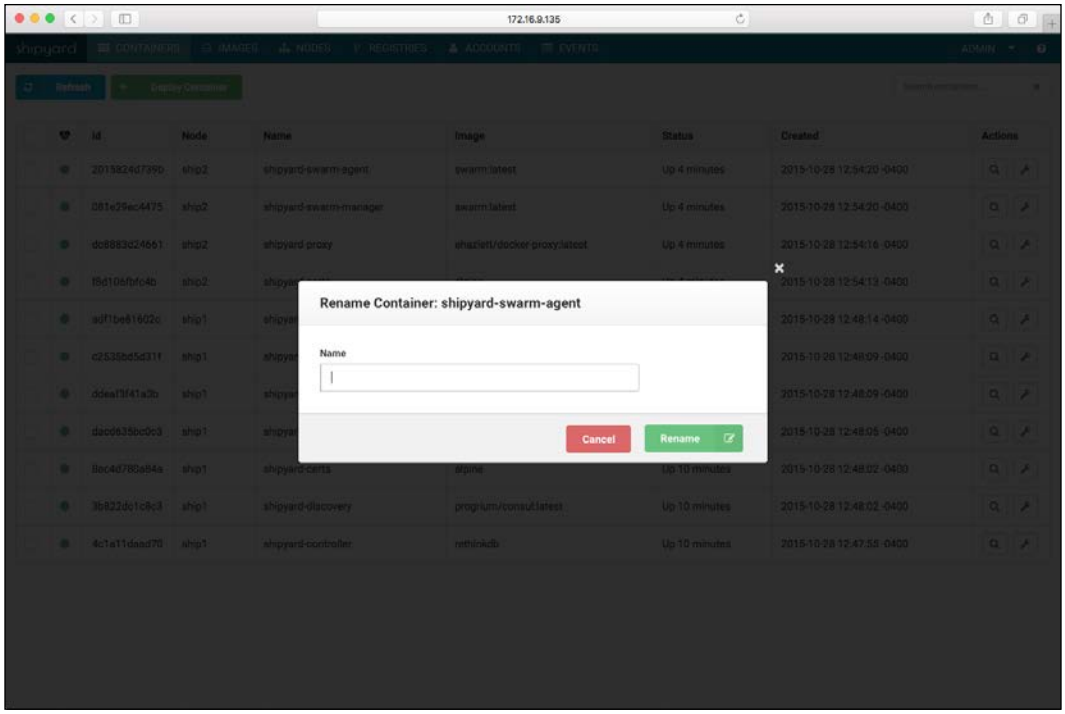


We can also scale or rename the container and get to the other areas we saw earlier such as **Stats**, **Console**, or **Logs**.

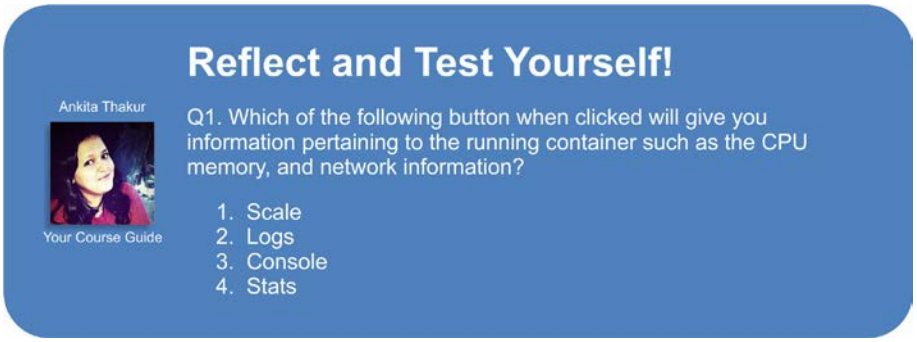


You will be taken to this section if you click on the **Scale** option. This will allow you to enter a numerical value and scale the instance up as far as you like.

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 of Module 5 Chapter 2

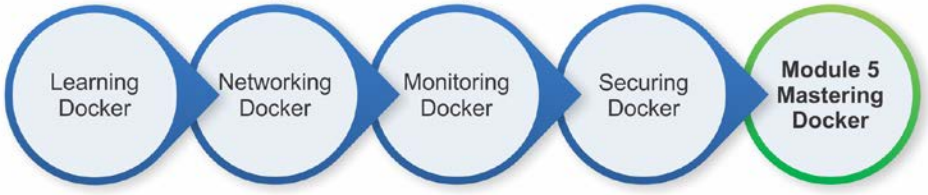
Ankita Thakur

Your Course Guide

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**.

Your Progress through the Course So Far



3
Panamax

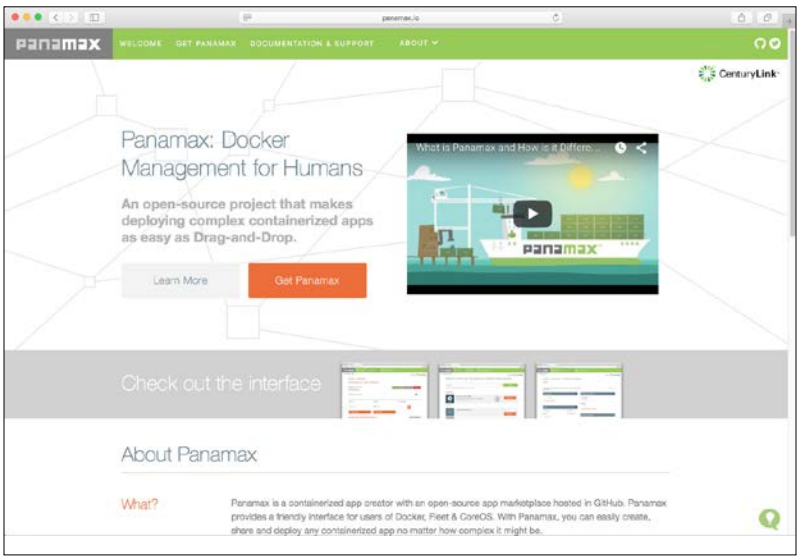
Panamax is another open source project that helps with deploying Docker environments by using a GUI interface to allow you to control just about everything that you can with the CLI.

In this chapter, we will cover:

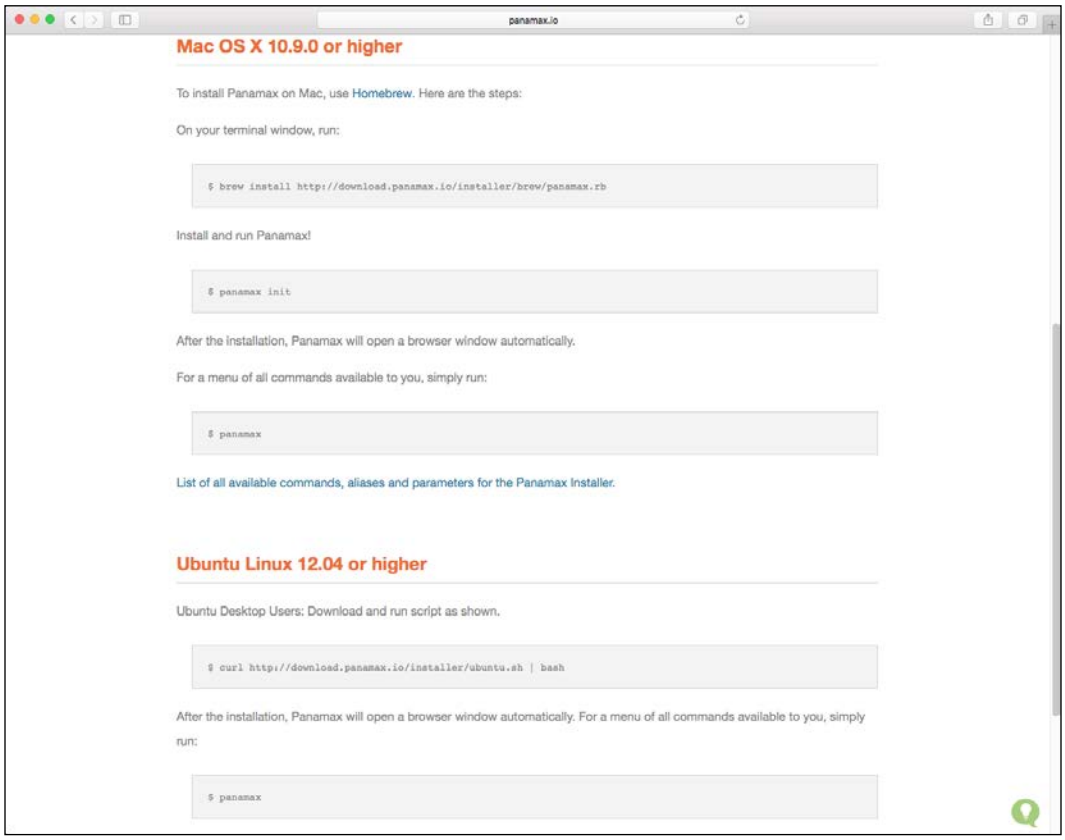
- Installing Panamax
- What after installing?

Installing Panamax

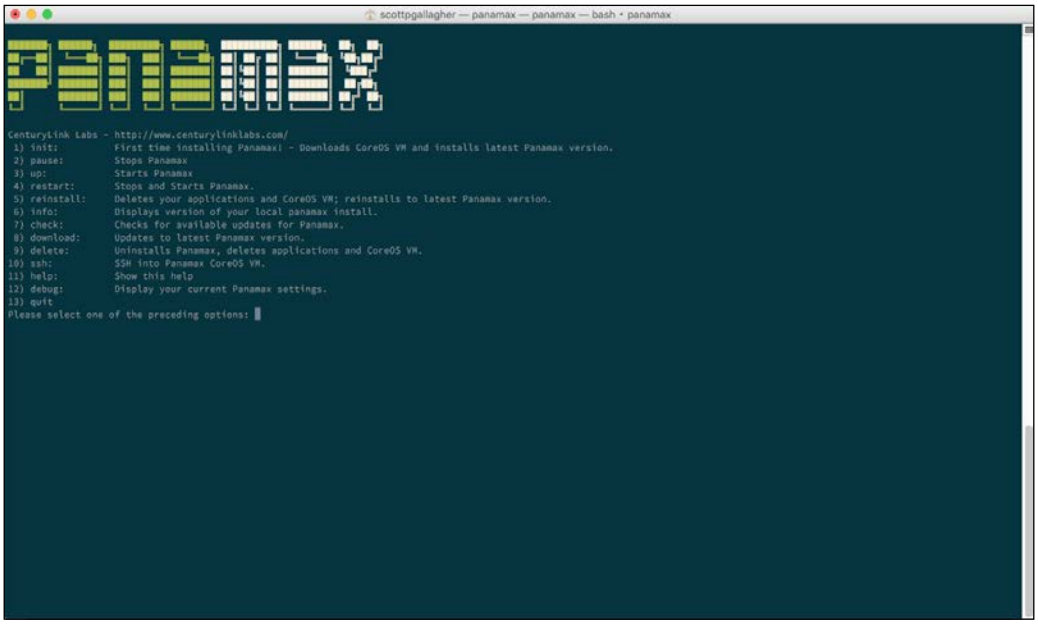
You will see the following page while navigating to the Panamax website at `http://panamax.io/`:



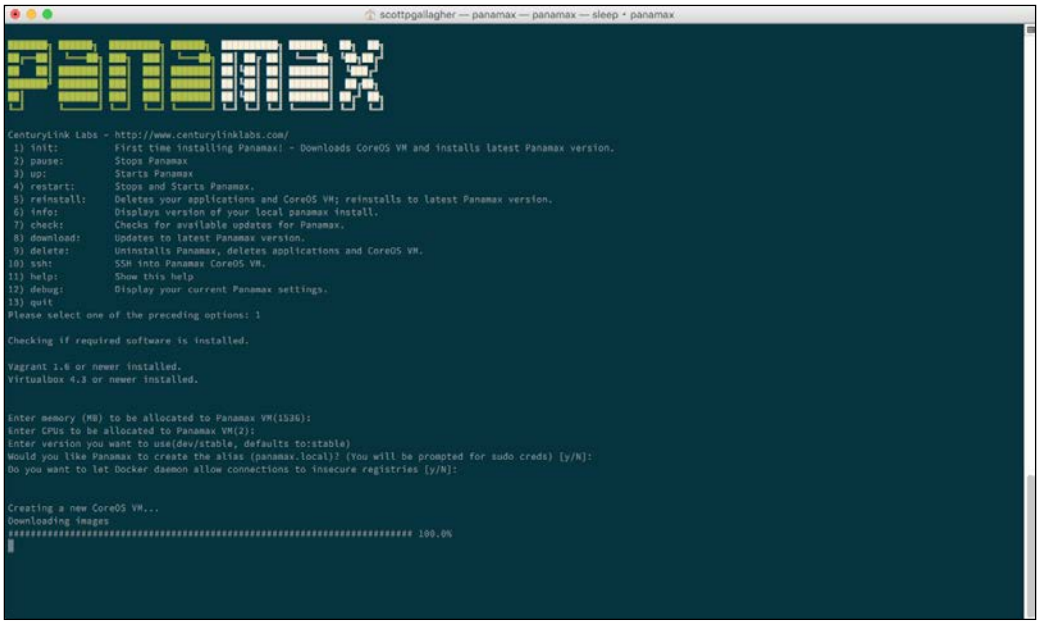
Next, you will see the instructions to install Panamax on both Mac OS X and Ubuntu:



After running the `panamax init` command and then the `panamax` command, you will see the following options:

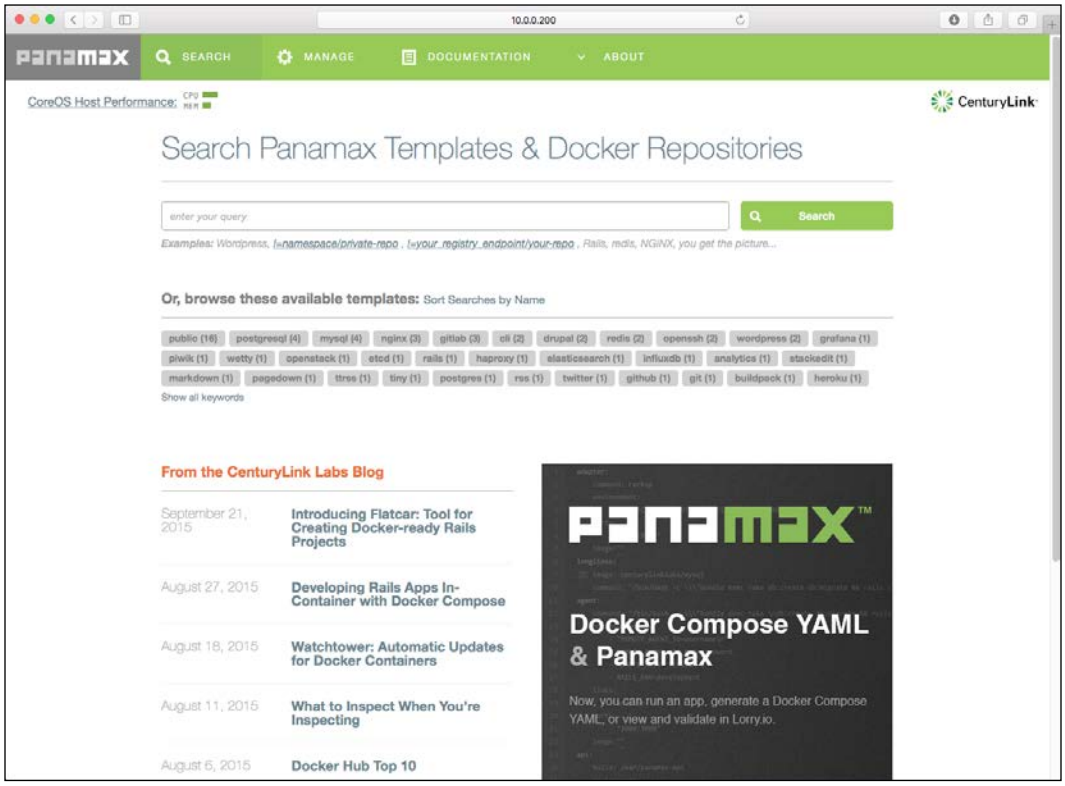


Upon selecting the first selection `init`, all the magic starts to happen.



Once all the magic is complete, you will be taken to the Panamax dashboard.

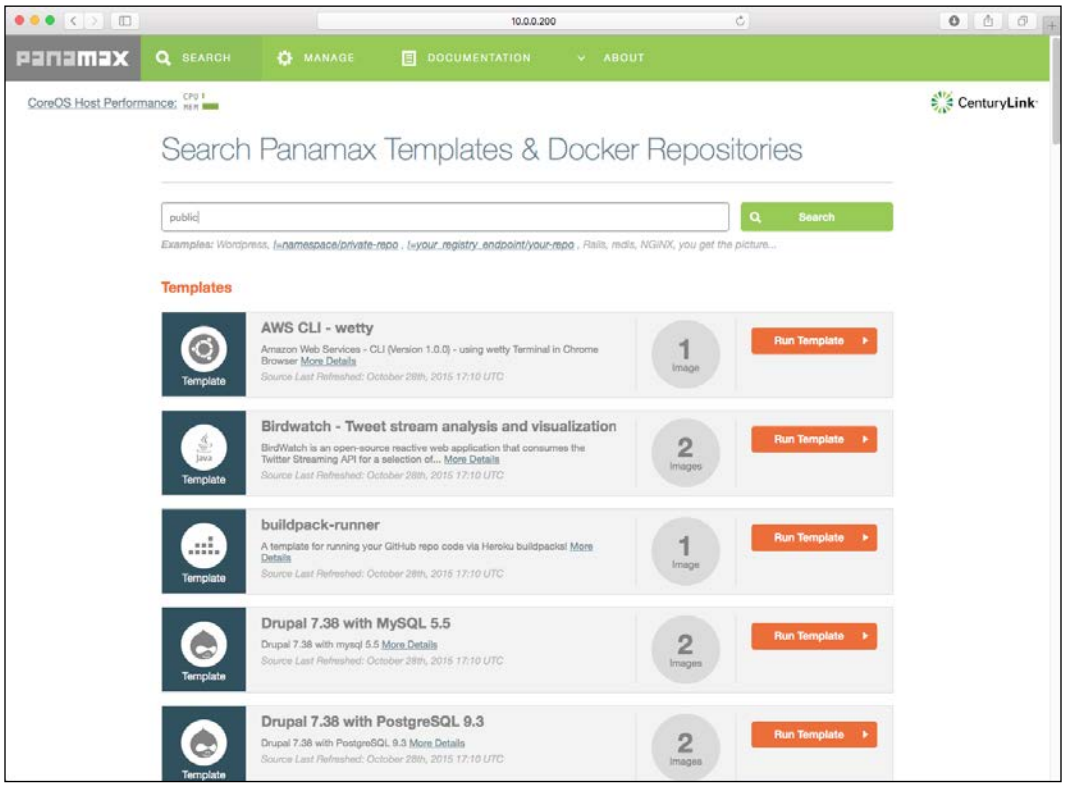
The following screenshot shows you what you will see once the installation has been completed and the browser page has been loaded for you:



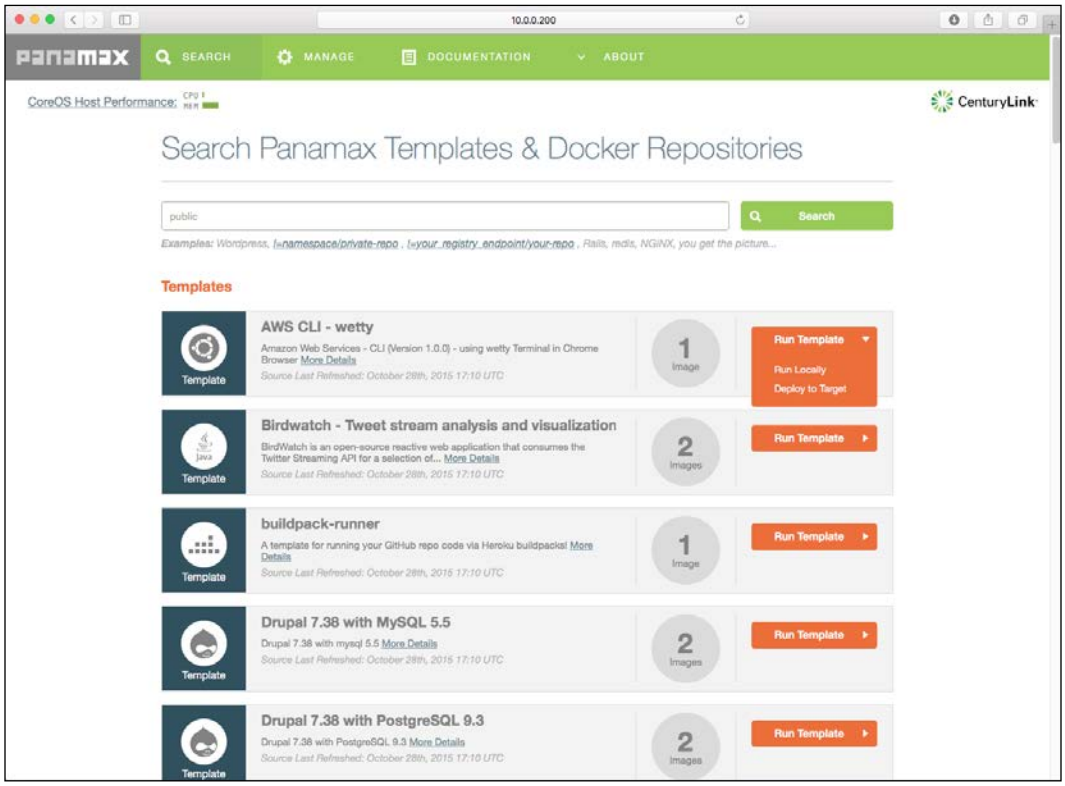
On this page, you can search for images that are on Docker Hub or browse the available templates that Panamax has to offer. You can also see the performance of the host that is running Panamax at the top with information such as the CPU and memory usage.

An example

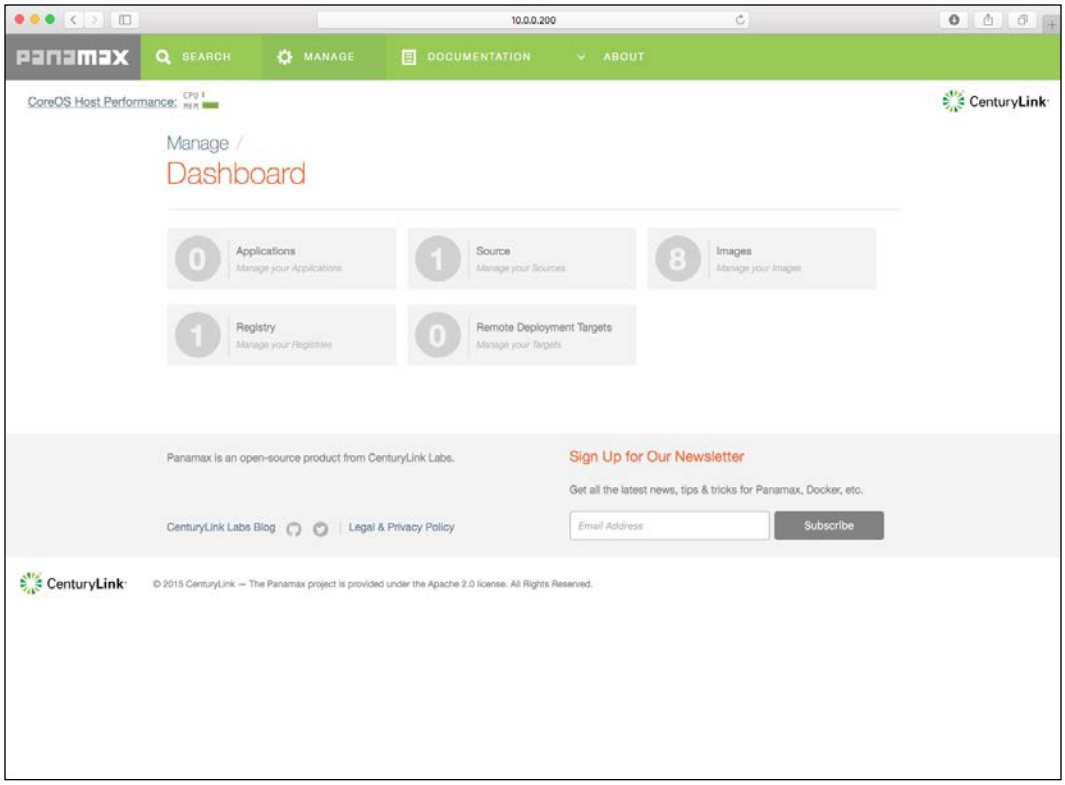
For this example, we select `public` from its available templates and use the `AWS CLI - wetty` image to run.



You can see information such as the image name, the description, how many images it will contain, and the option to run the template.



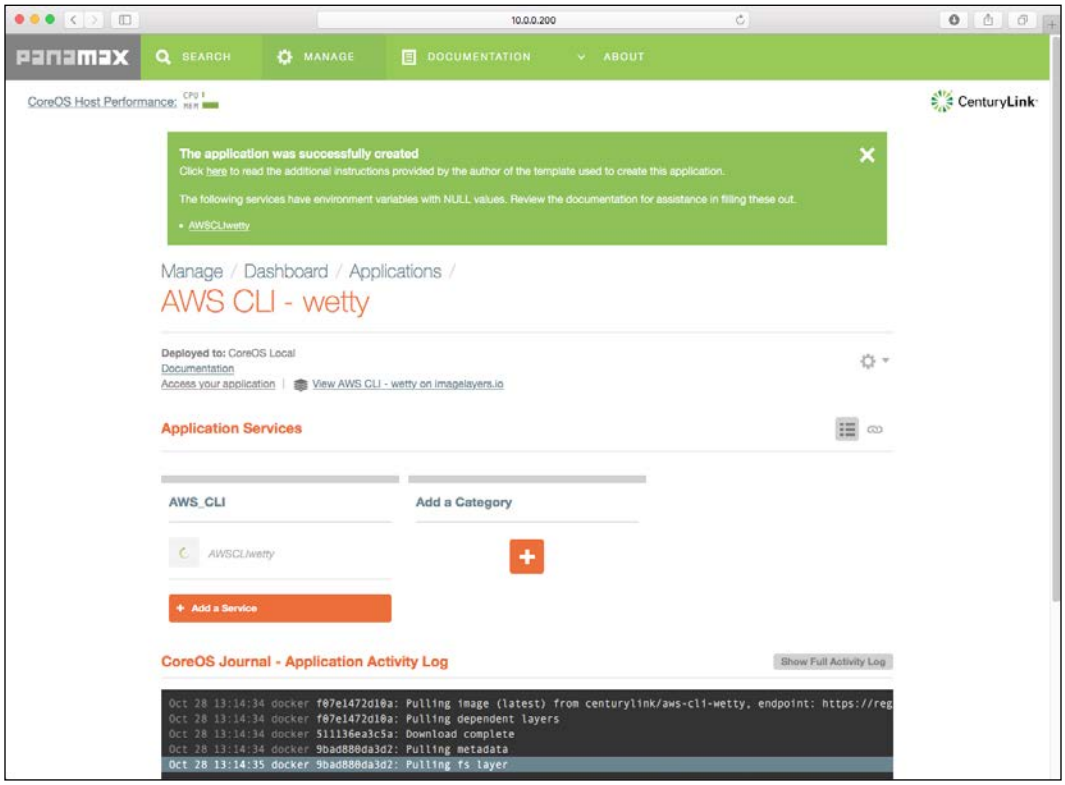
Upon clicking the **Run Template** button, you will get two options. You can run it locally or deploy it to a target, such as the cloud. For this example, we will choose to run it locally.



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

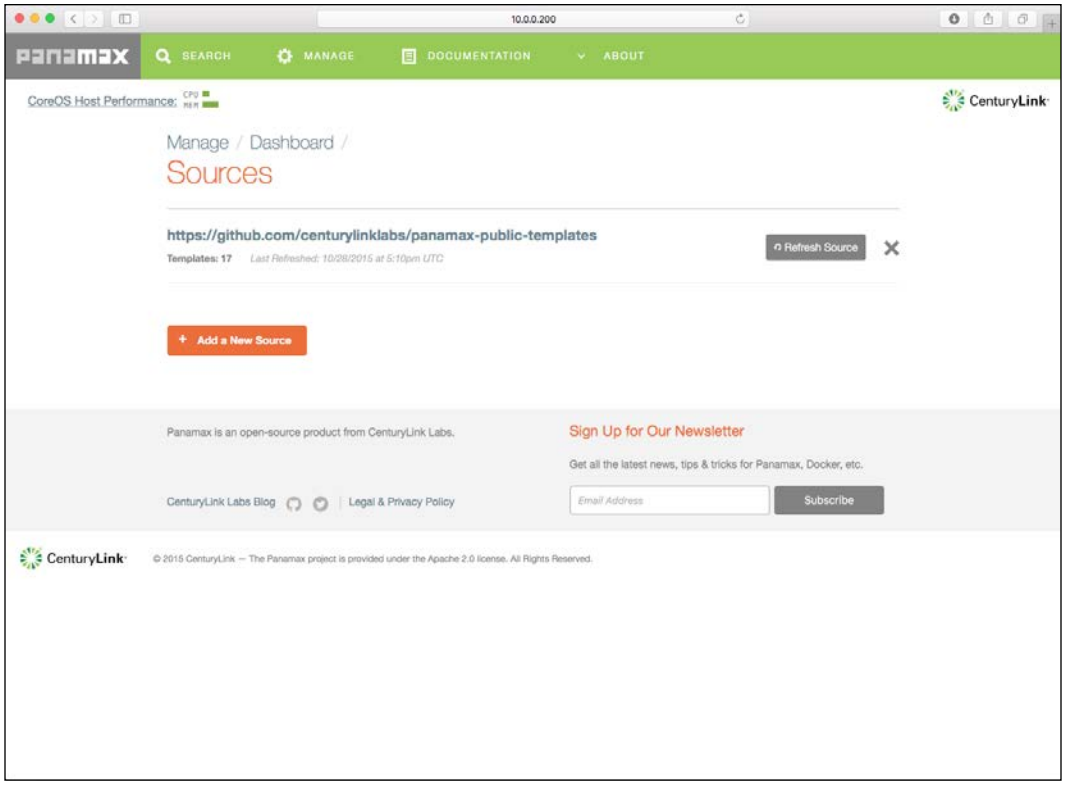
First up is the **Applications** section. Upon entering this one, we can see the application we launched earlier is now in here.



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

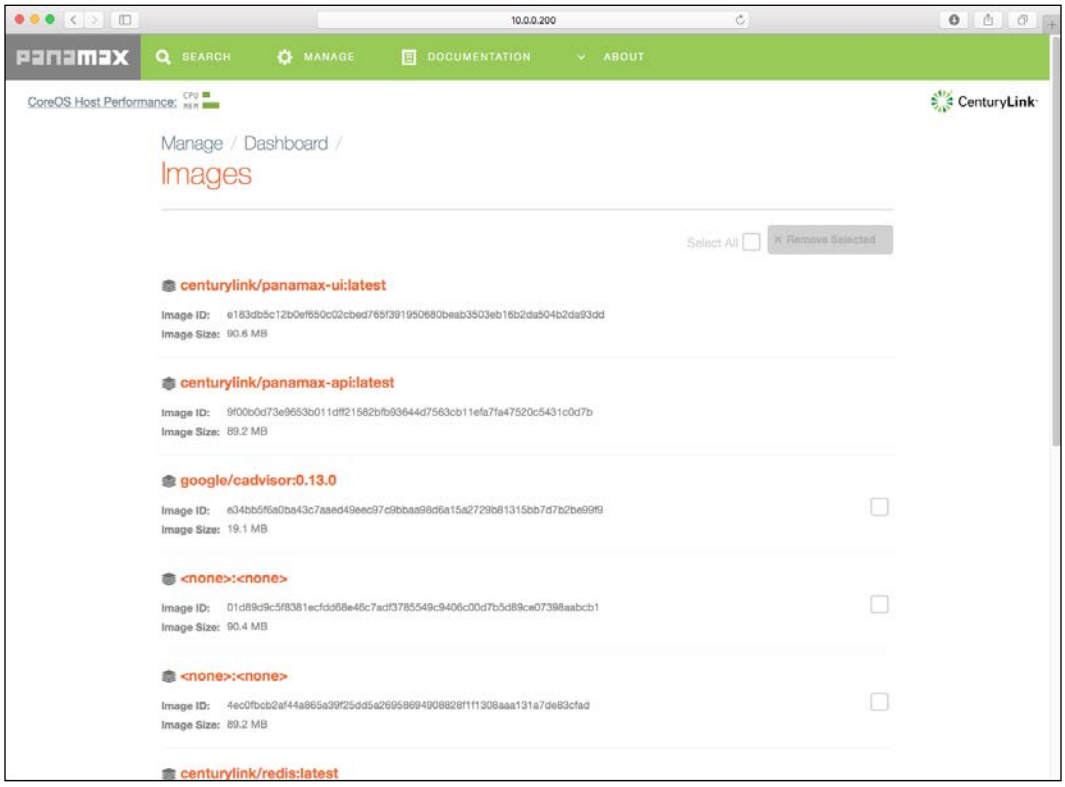
The **Sources** section shows you what resources are currently loaded into the system.



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

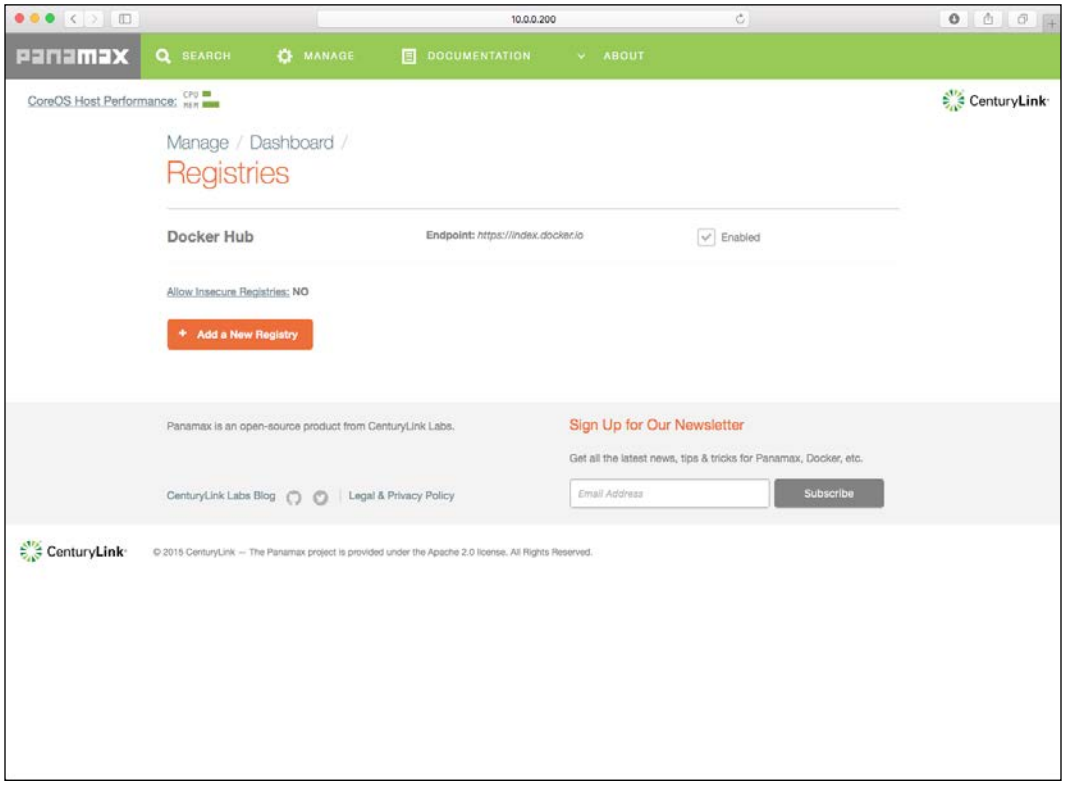
In the next section, the **Images** section, you can see all the images that are currently being used.



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

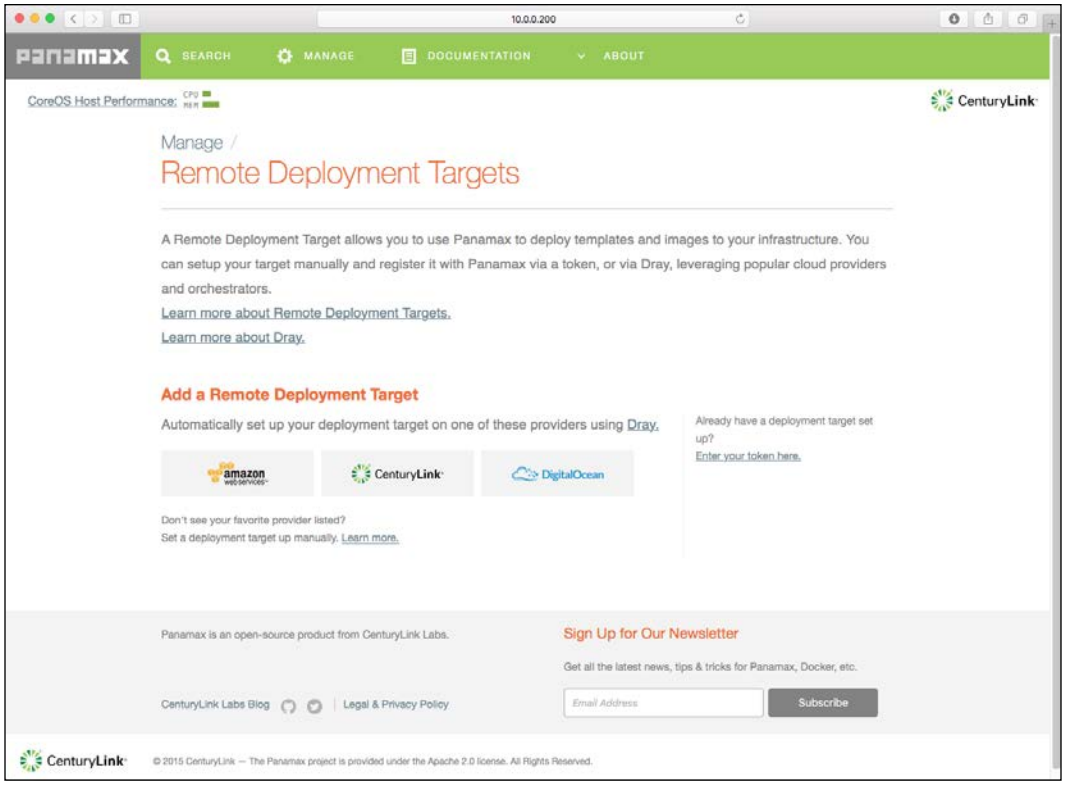
The next section deals with the registries that you can search for templates and images. By default, it searches Docker Hub and includes insecure registries along with secure registries.



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**.

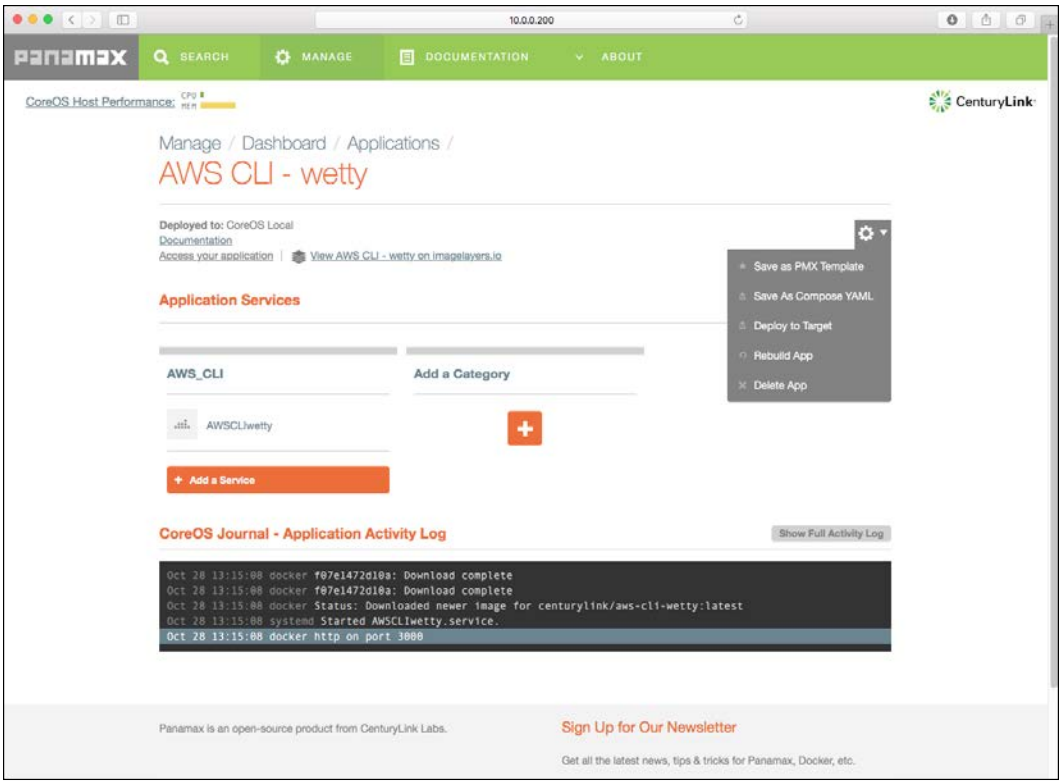


These are items such as cloud hosts that may include AWS, CenturyLink, and DigitalOcean.

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

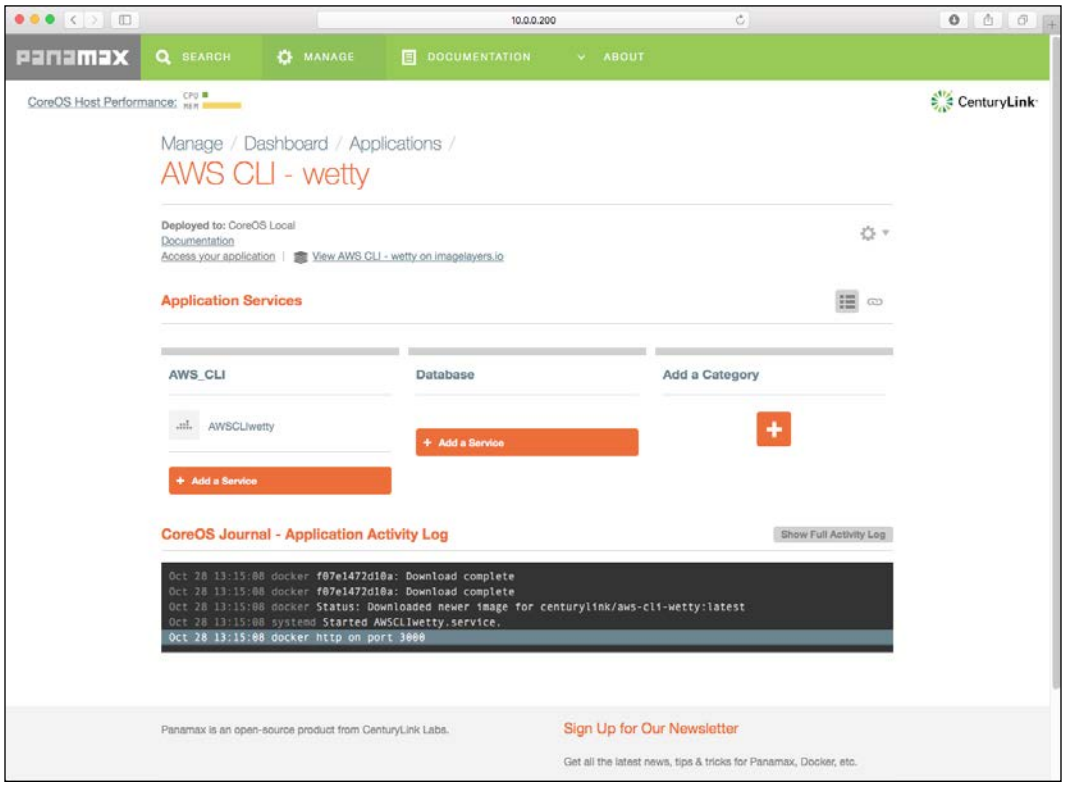
Back in our **Applications** section under the application that we deployed earlier, the `AWS CLI - wetty` image, we can click on the gear icon on the right-hand side of the screen. Given some options such as saving as a PMX template that will allow you to share it with others that are using Panamax, 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.

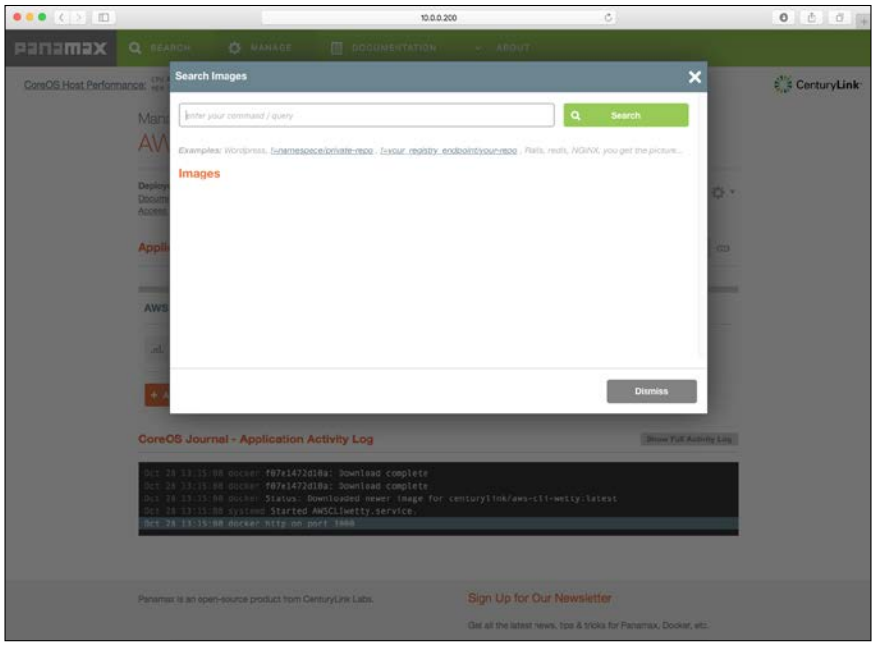


Adding a service

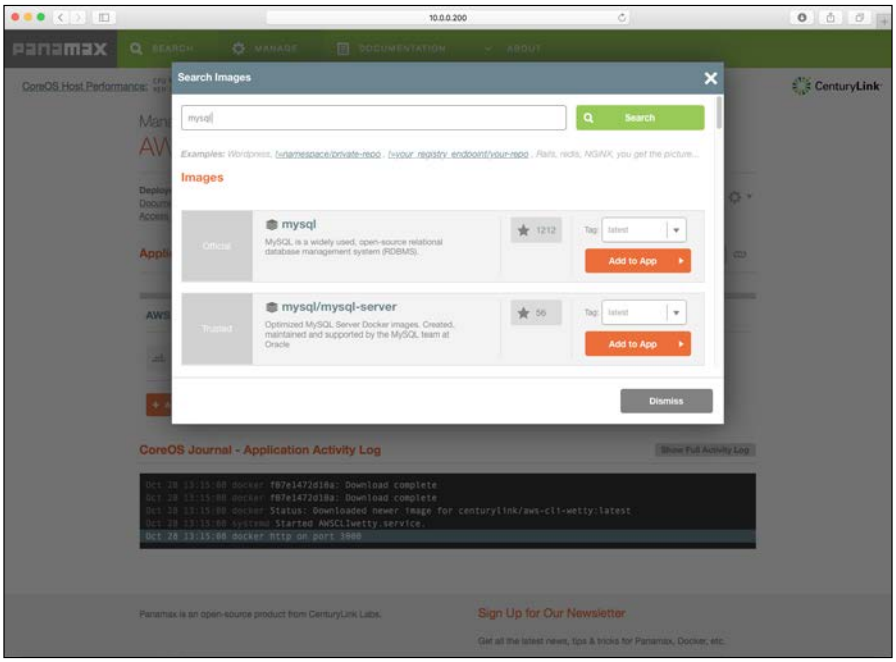
Next, we are going to add a service to our application. To do so, we will click on the **+** button and then give it a name.

In our case, we are going to add a database, so we will name this section `Database`.





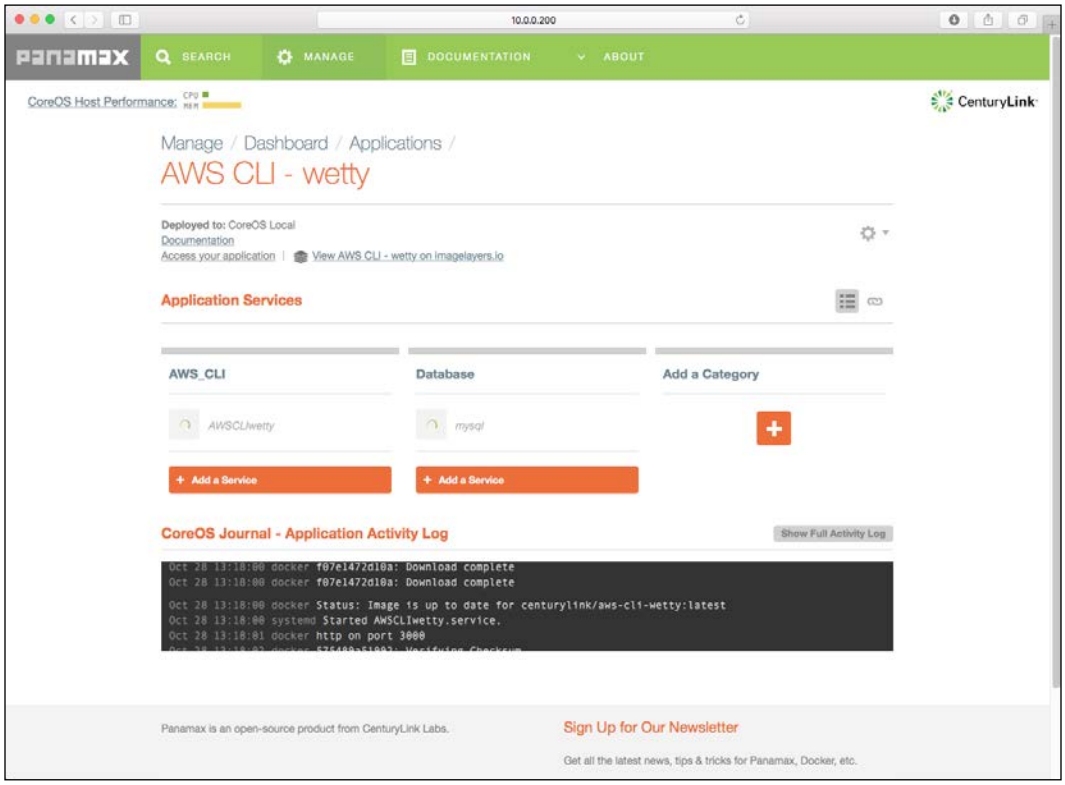
After this, we will click on **+ Add a Service** to the database's application services and will need to search for an image that we want to use.



Since this is a database application and MySQL is known by almost everyone, we will search for it and add it to the app.

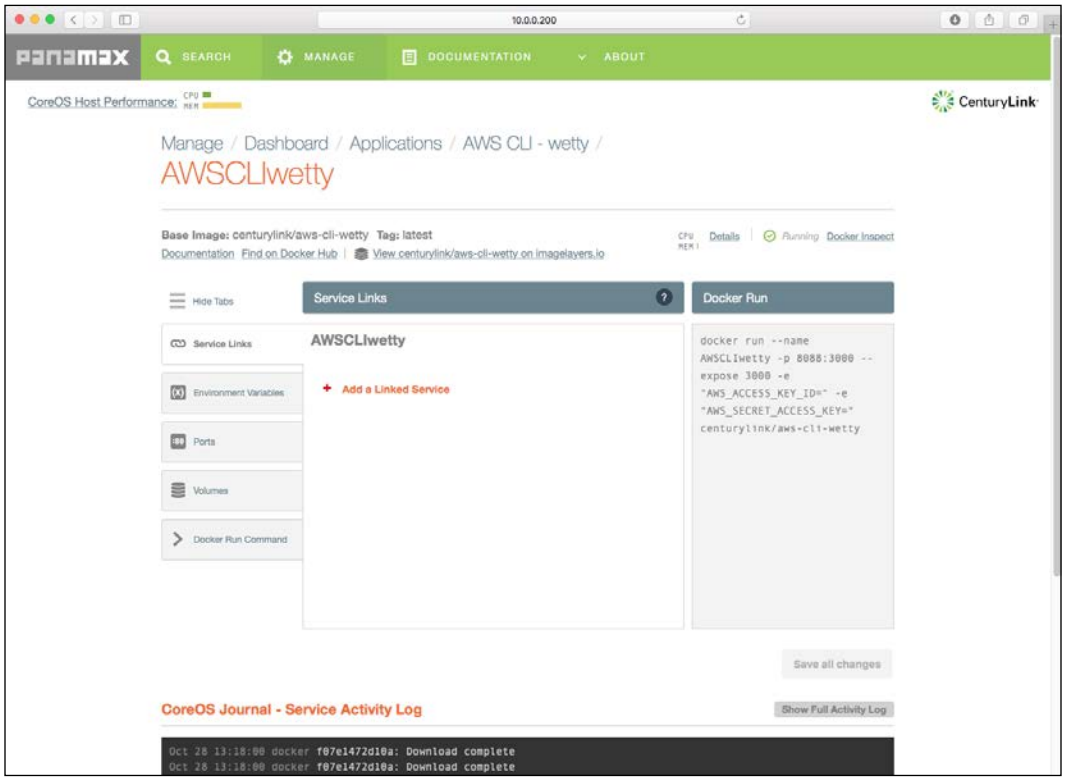
Configuring the application

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

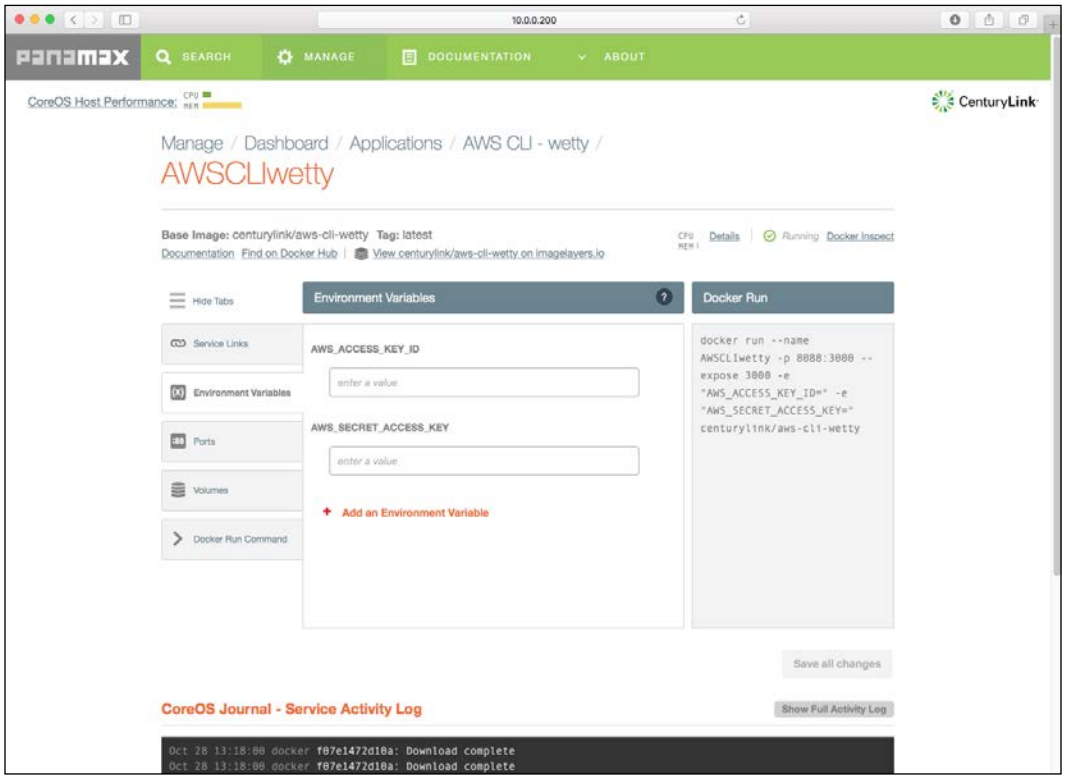
If you want to configure each application service, you can click on it and you will be taken into a submenu.



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

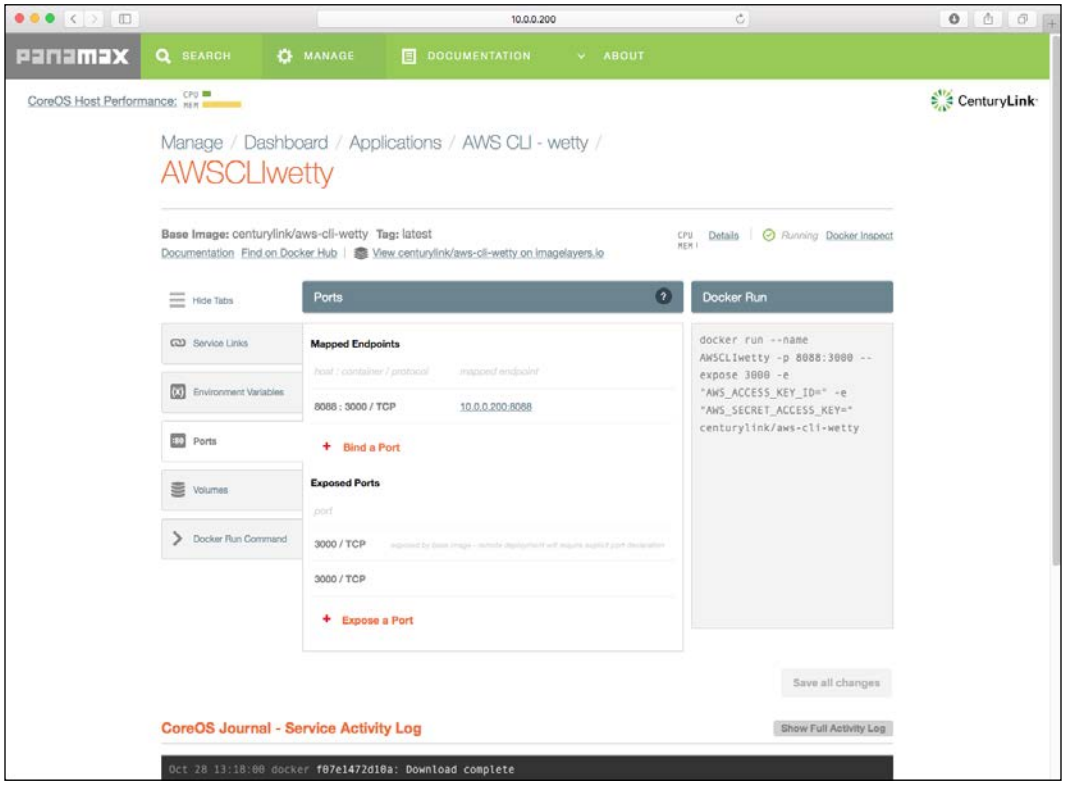
Next are the environmental variables. For this image, it would ask us to supply our AWS access key ID and our AWS secret access key.



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

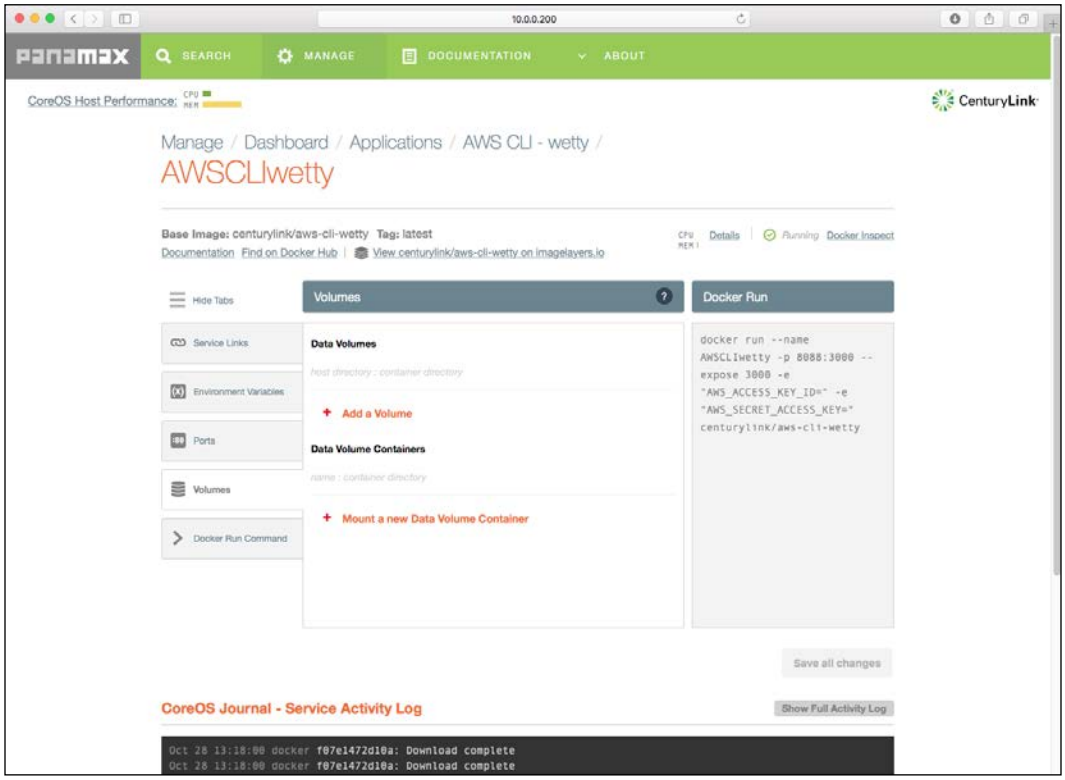
Next, you can view or configure the port configuration that each service uses.



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

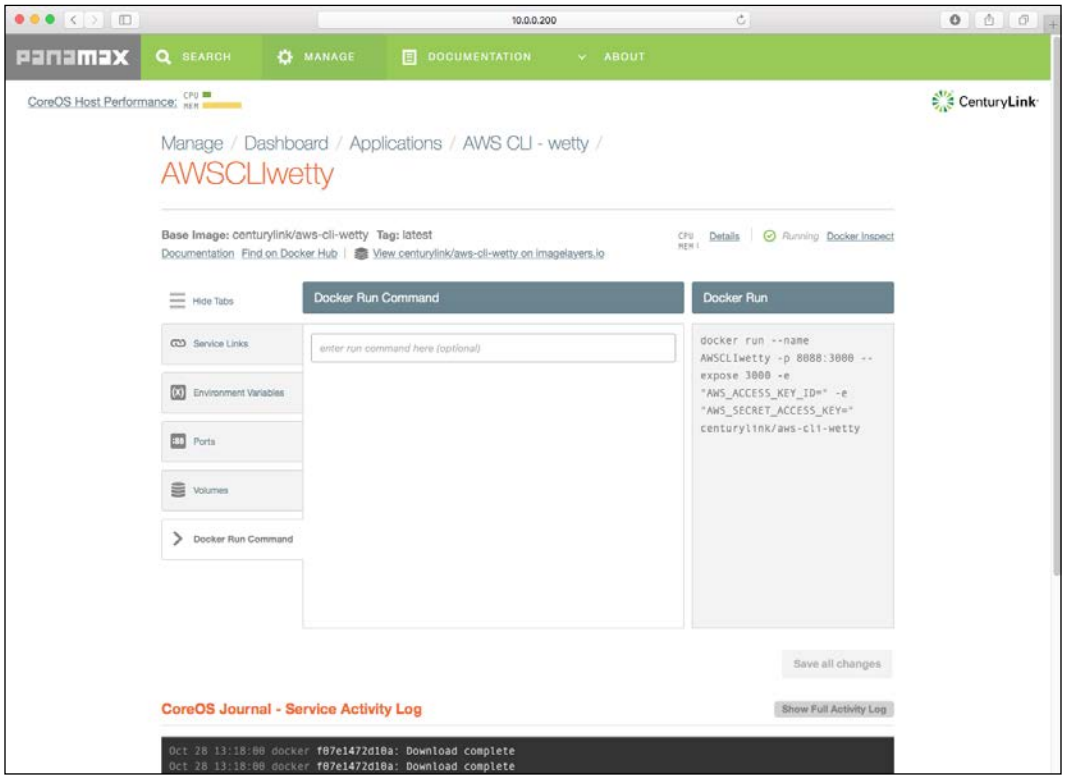
Next, we can see the volume configuration for each service.



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 of Module 5 Chapter 3

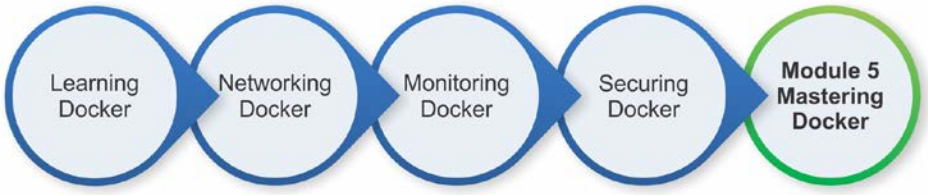
Ankita Thakur

Your Course Guide

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.

Your Progress through the Course So Far



4
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.

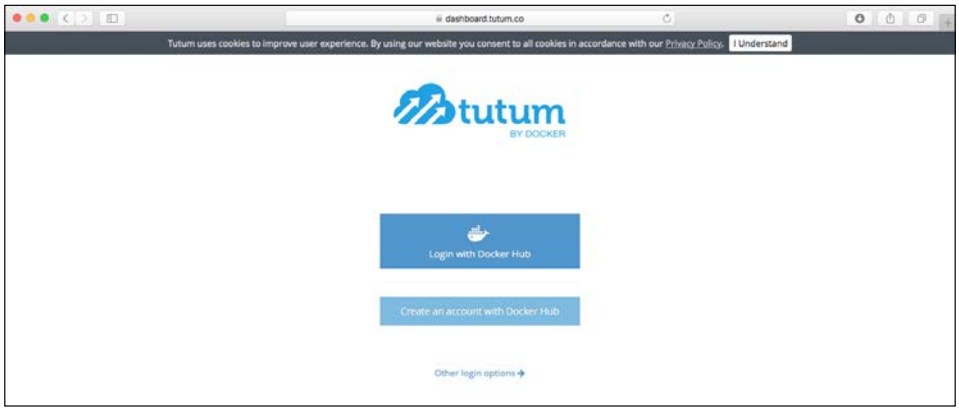
In this chapter, we will cover how to:

- 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`.

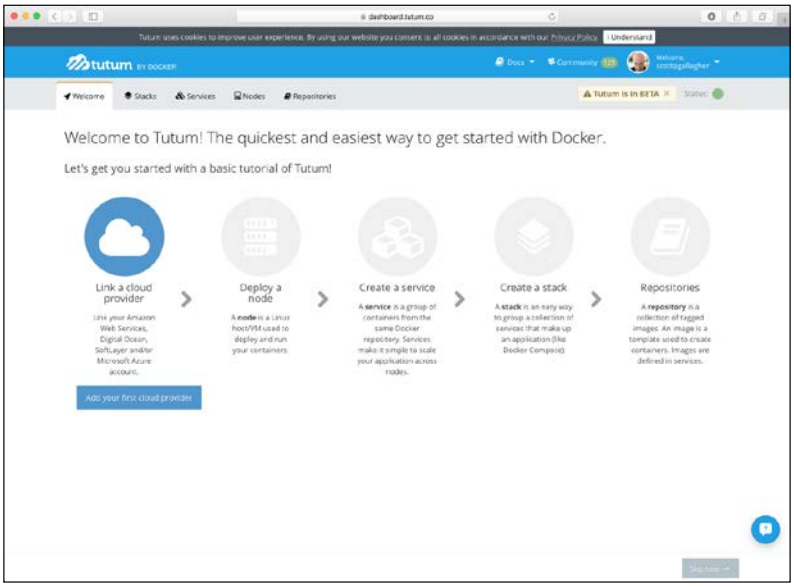
Upon clicking **Get started for free!** or the **Login** link, you will be presented with the following screen:



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

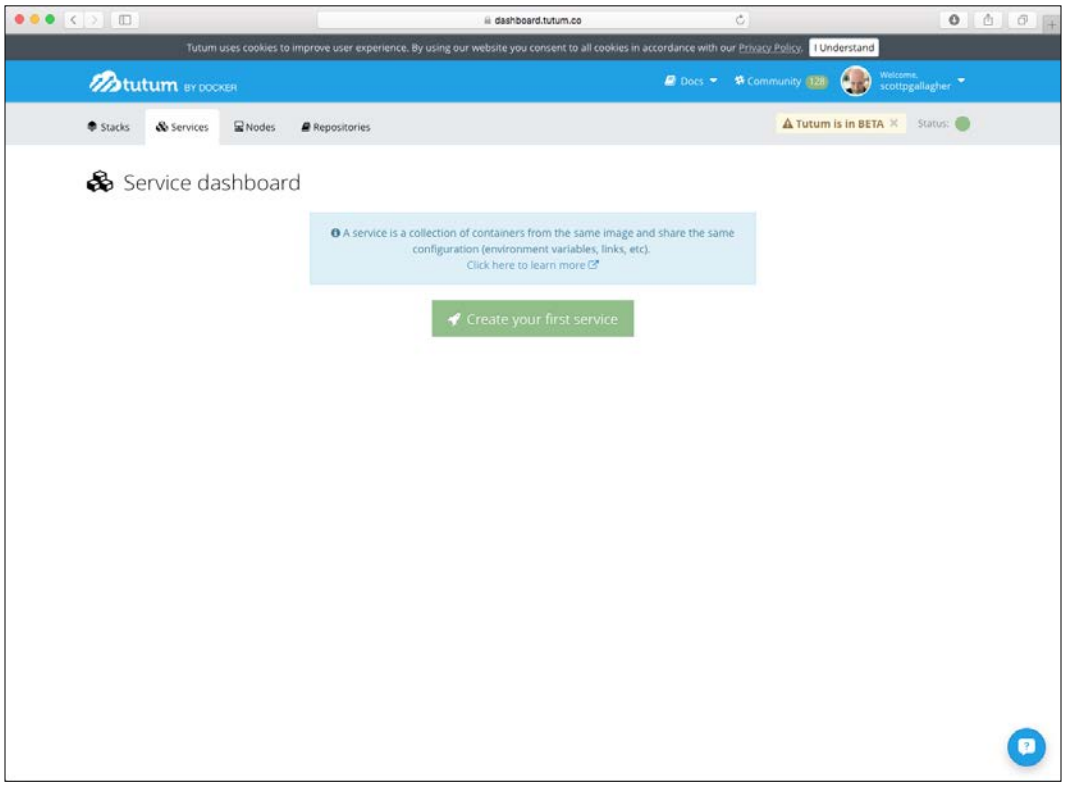
You will be presented with the tutorial page that will provide a tour of Tutum if you wish.



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.

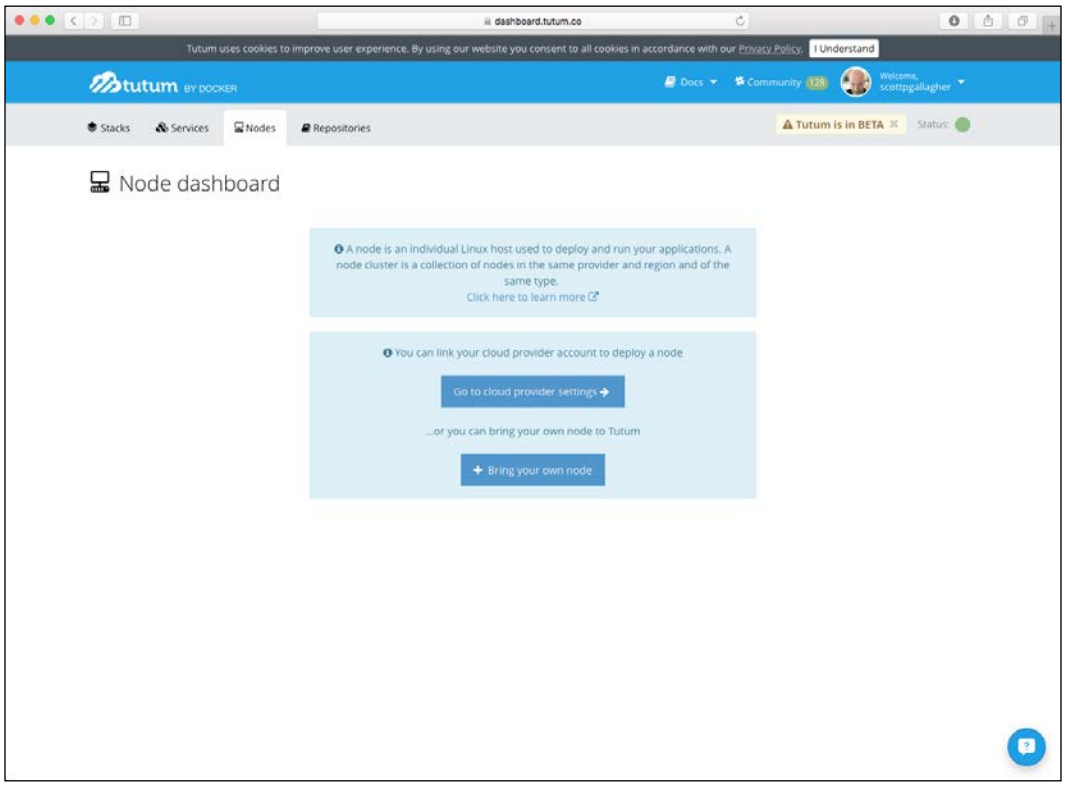
The Service dashboard

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

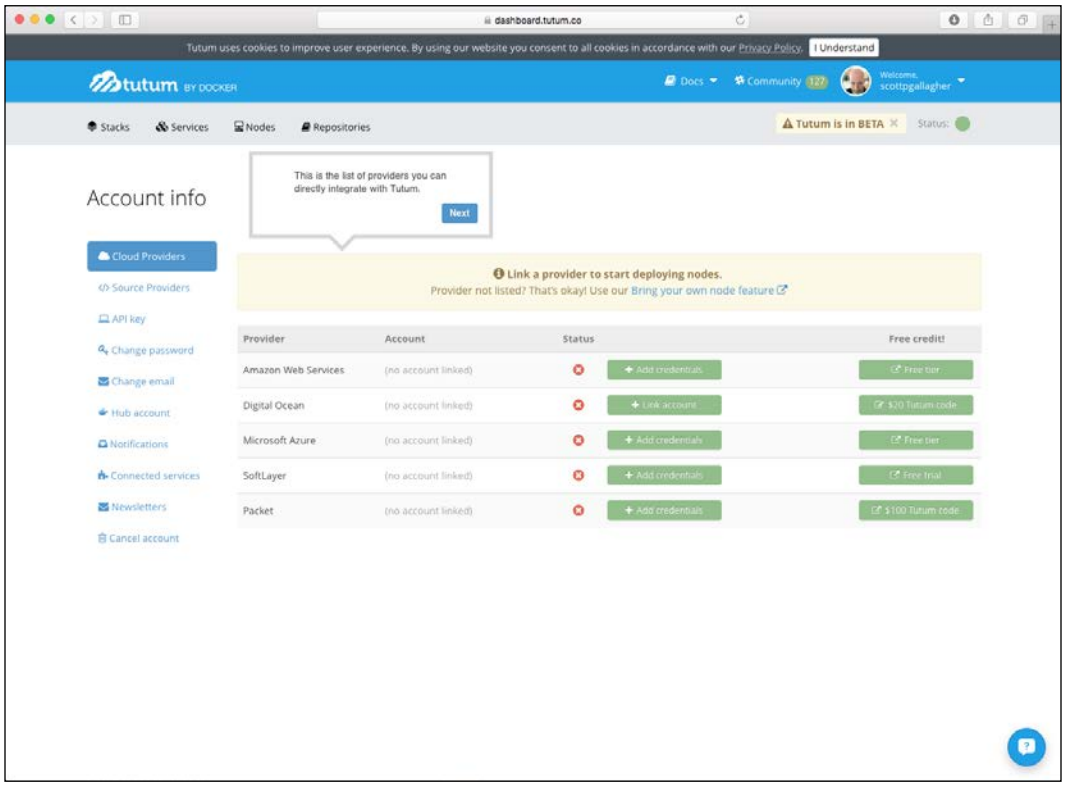
If you click on the **Nodes** section in the navigation bar, you can start adding your cloud provider or you can bring your own node.



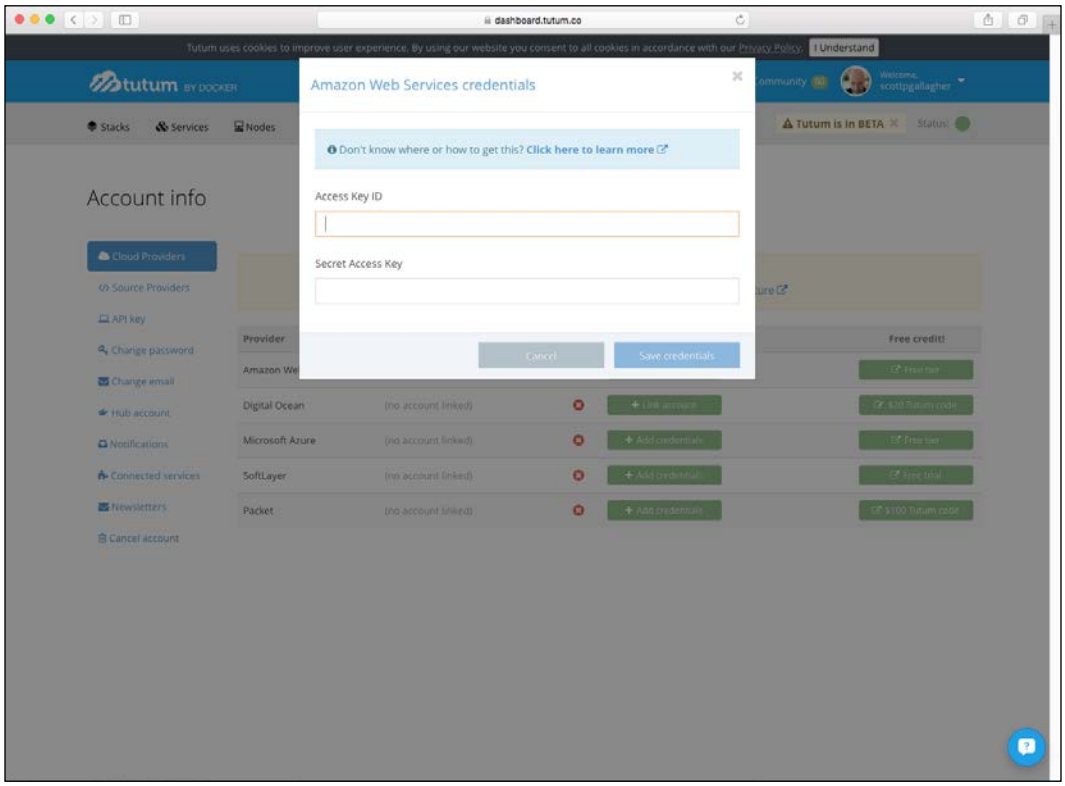
If you wish to bring your own node, you will need to install a client that Tutum uses to communicate with your node. For this example, we are going to stick with using a cloud provider: AWS in this case.

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:

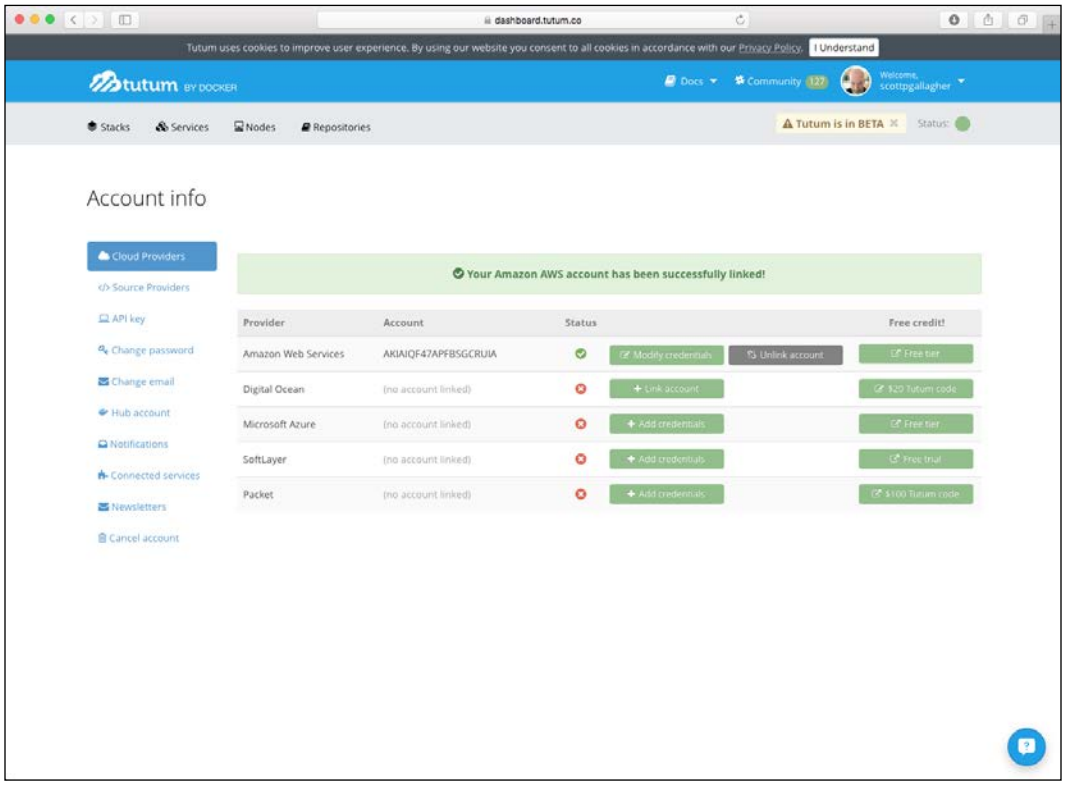


Here we would provide our AWS Access Key ID as well as our Secret Access Key:

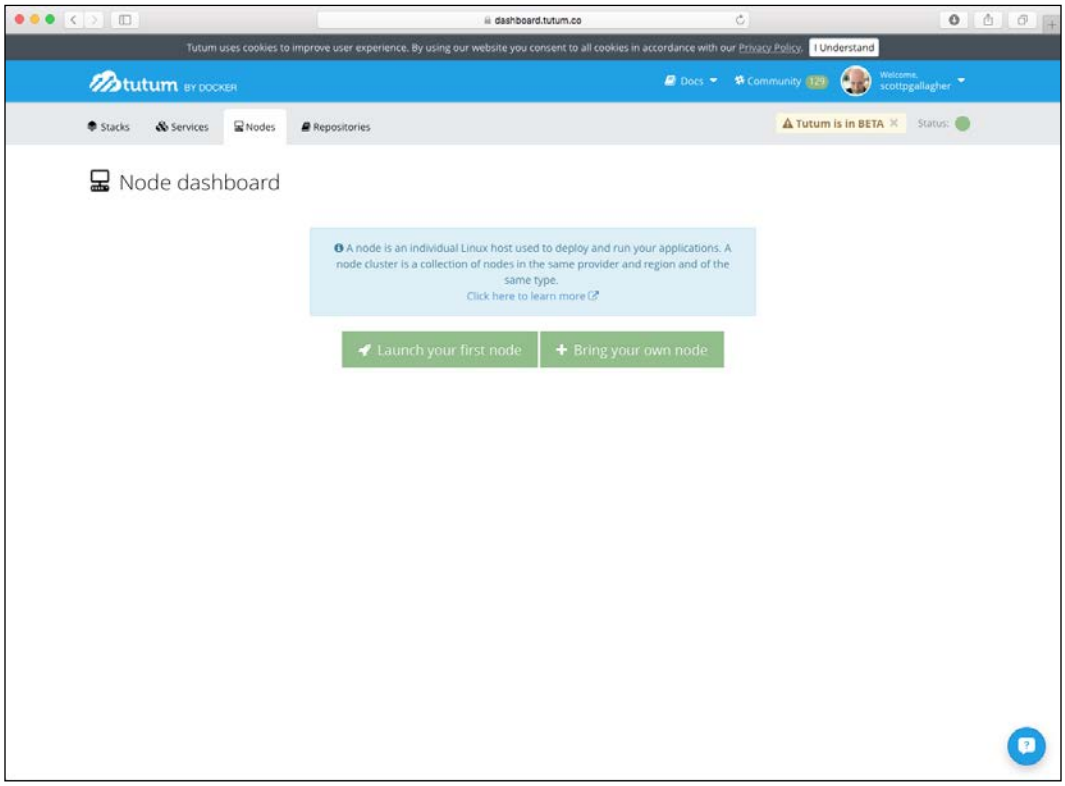


AWS uses your access key ID as well as your secret access key to authenticate against AWS. You can enter these details and then click on the **Save credentials** button.

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.



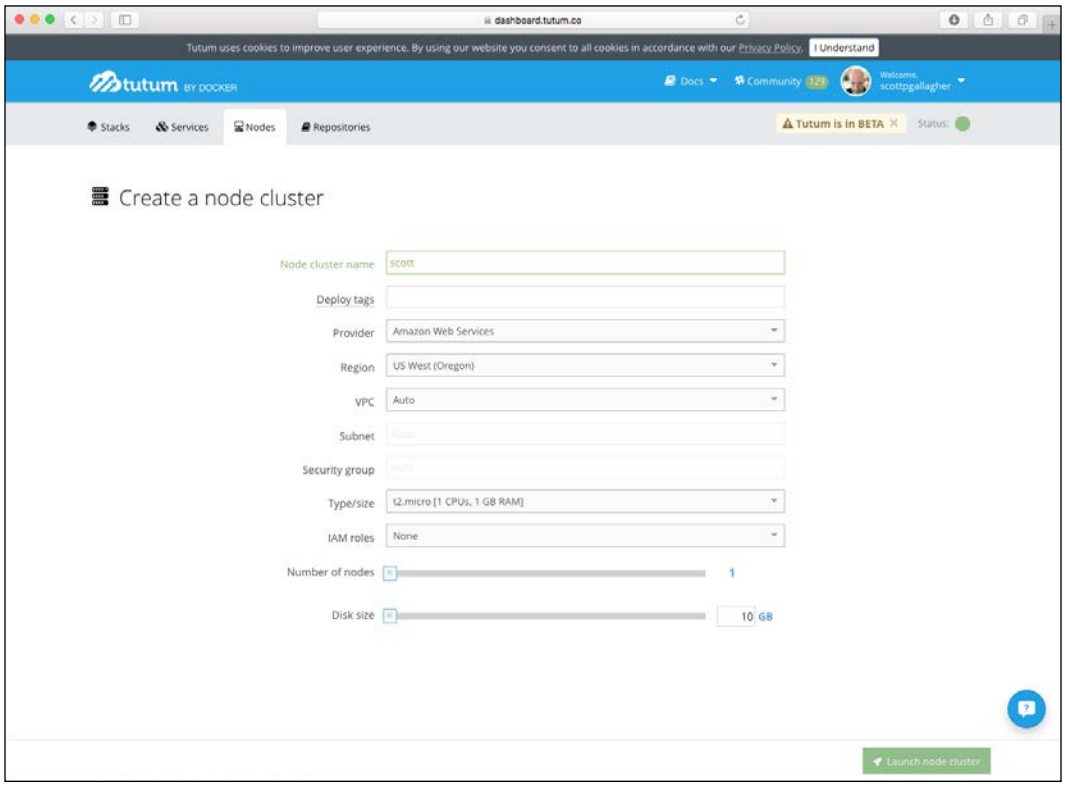
Now that we have a cloud provider to run our service on, we can launch our first node on the cloud now by clicking on the **Launch your first node** button:



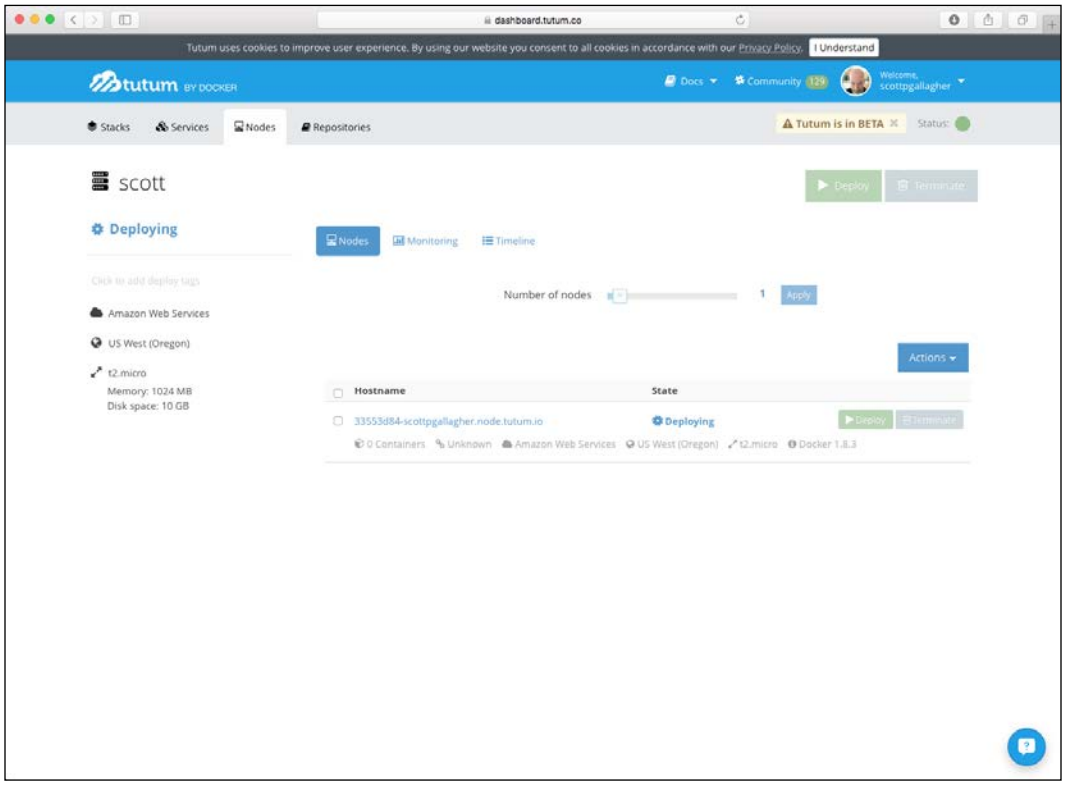
We will navigate back to the **Nodes** screen.

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.

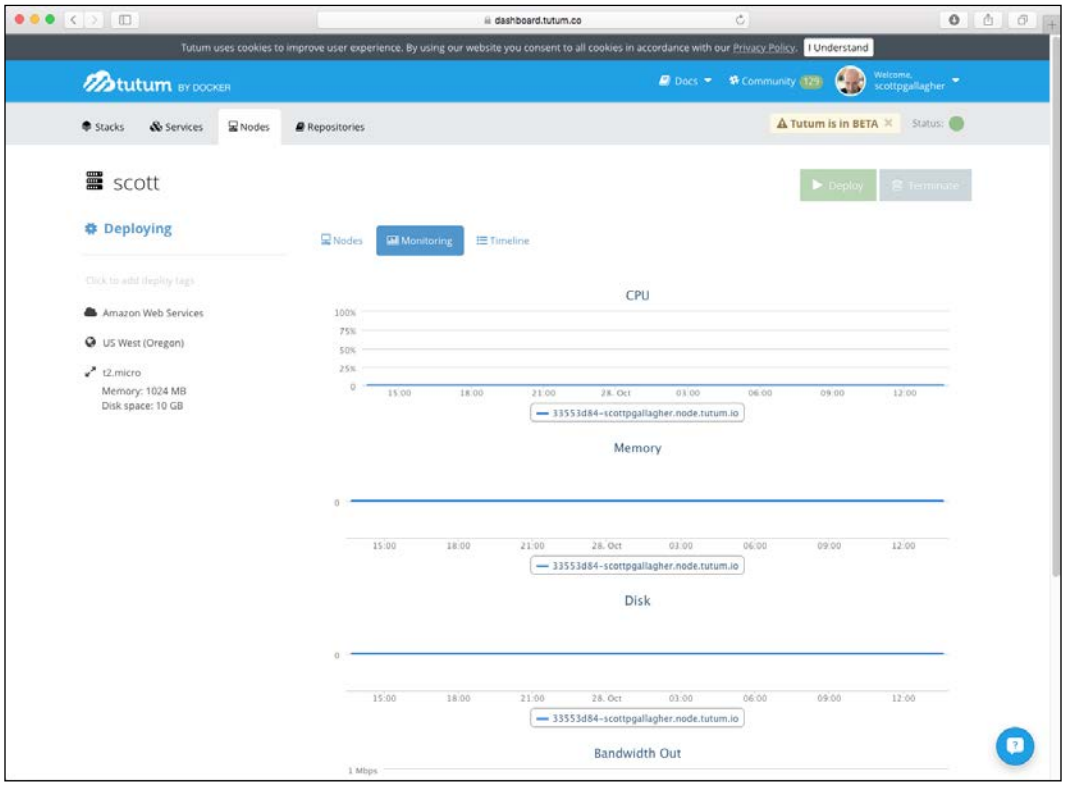


For our example, we mainly kept the default, only lowering the disk size to the minimum size of 10 GB.

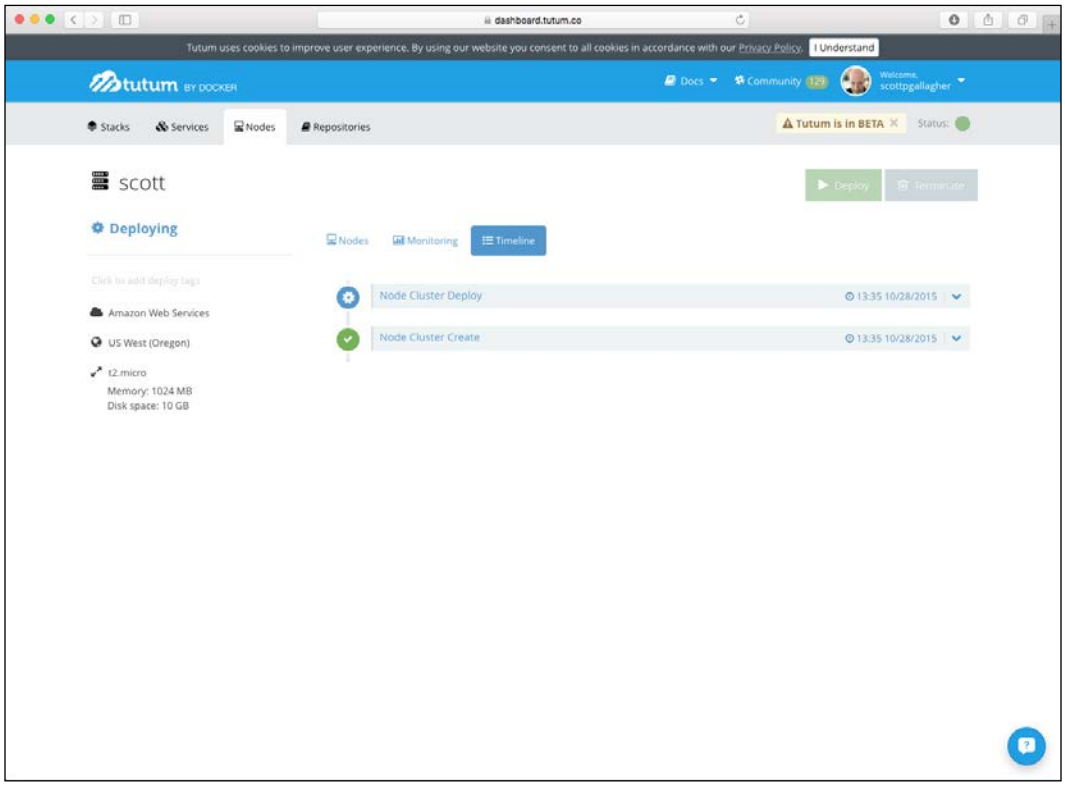


Once you have clicked on the **Launch node cluster** button, you will see the status of the node; in this case, it's **Deploying**. We also have some other items we can check out while it's deploying.

We can view the **Monitoring** tab and see information pertaining to the node such as **CPU**, **Memory**, **Disk**, and **Bandwidth Out**.

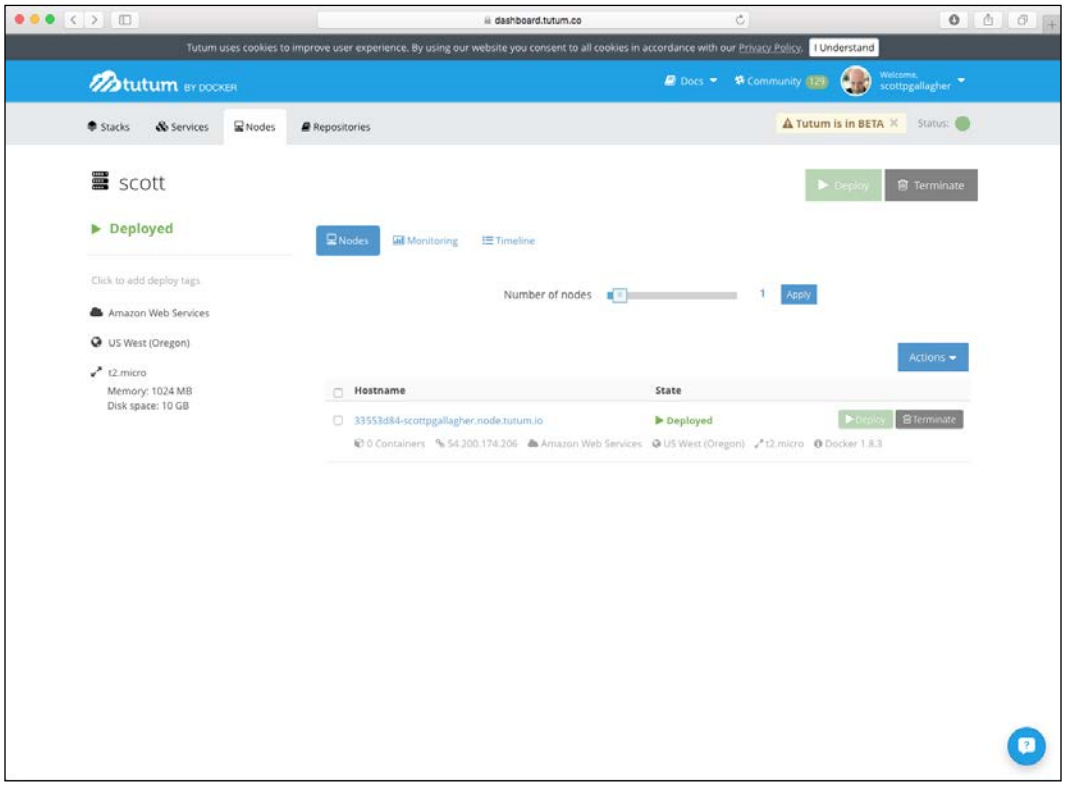


We can also view the timeline of our node. Now, at first, this will be very short as it's just showing us that we created the node and are deploying it.



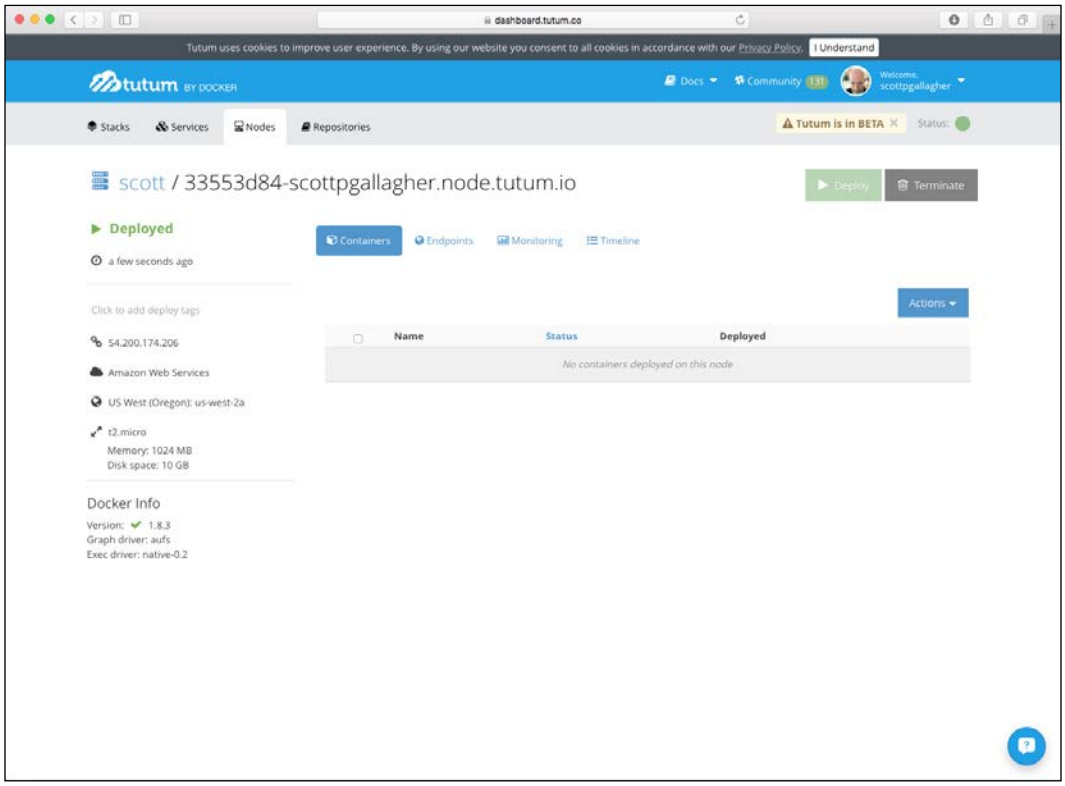
Over time, this timeline will grow and show you the progress of your node.

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.

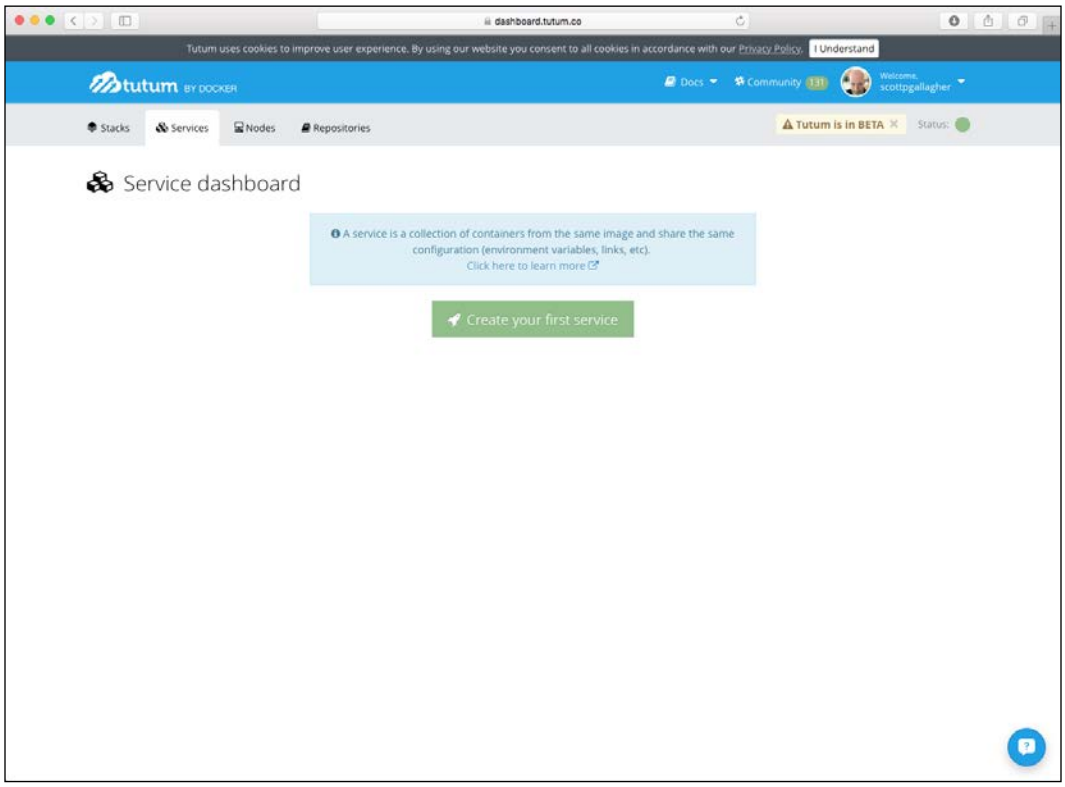
If we drill down into the node itself by clicking on its hostname, we can see some more information provided to us.



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.

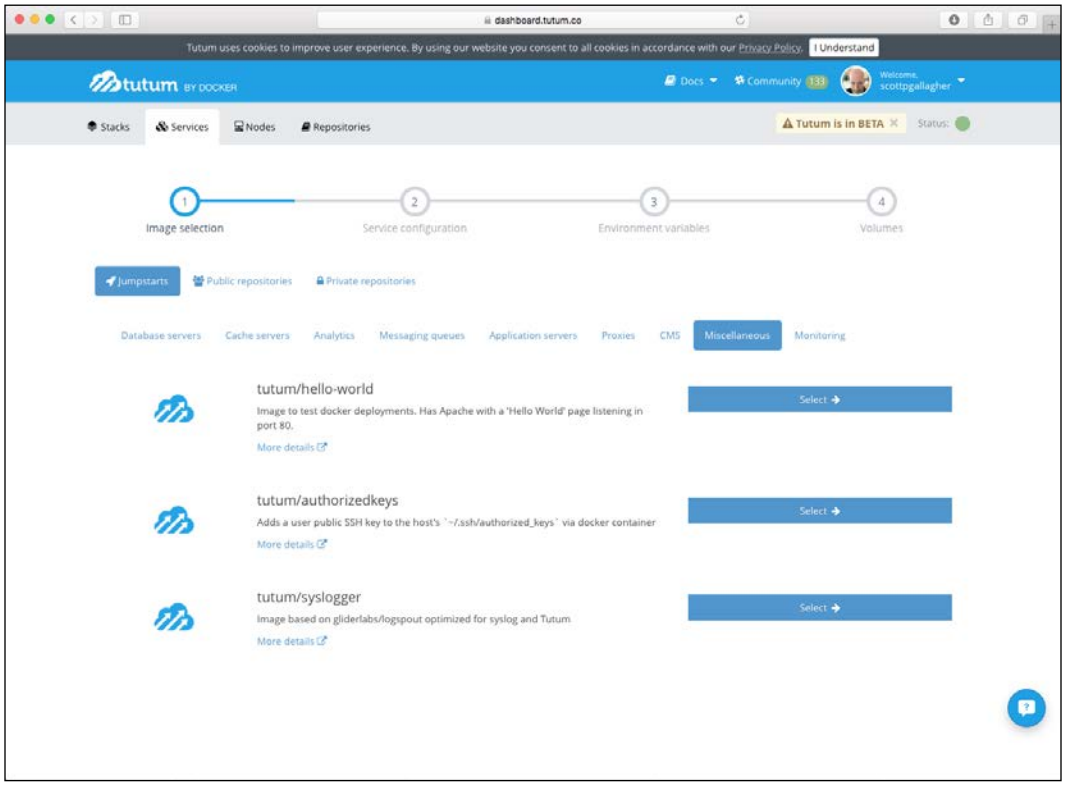
Back to the Services section

Now, it's time for us to launch a service and get some containers running on this node.

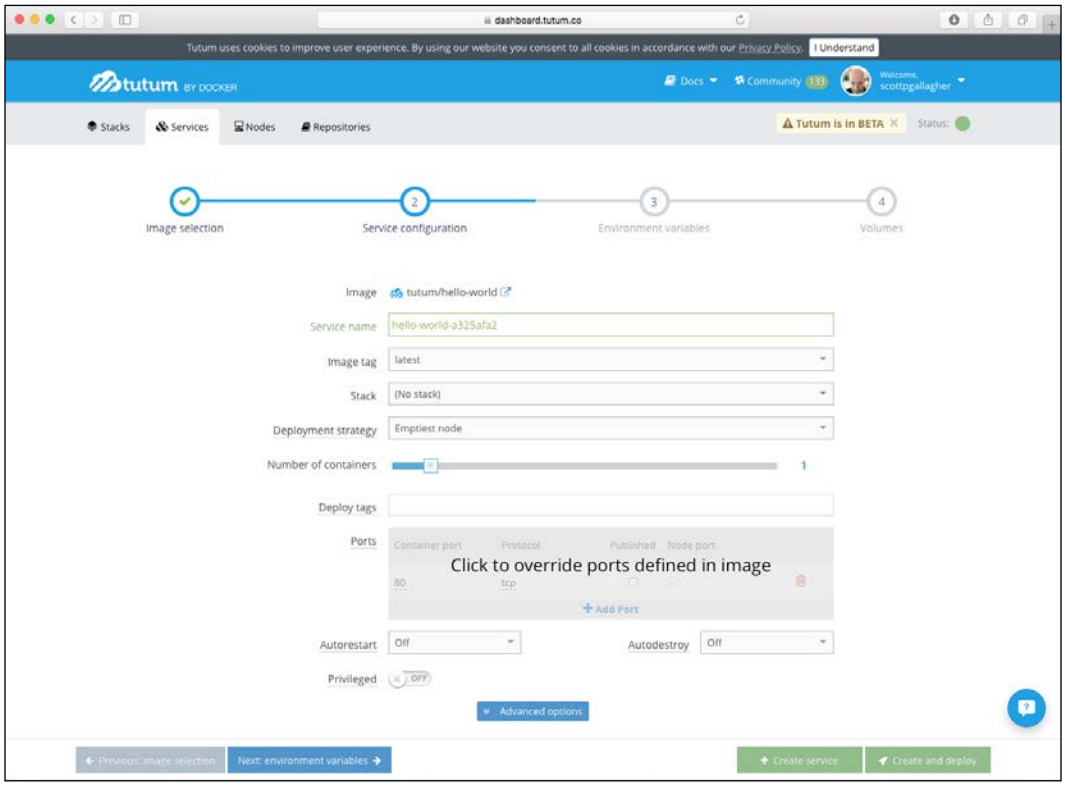


By clicking on the **Services** tab, we will be taken to the previous screen, where we can deploy a service.

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.

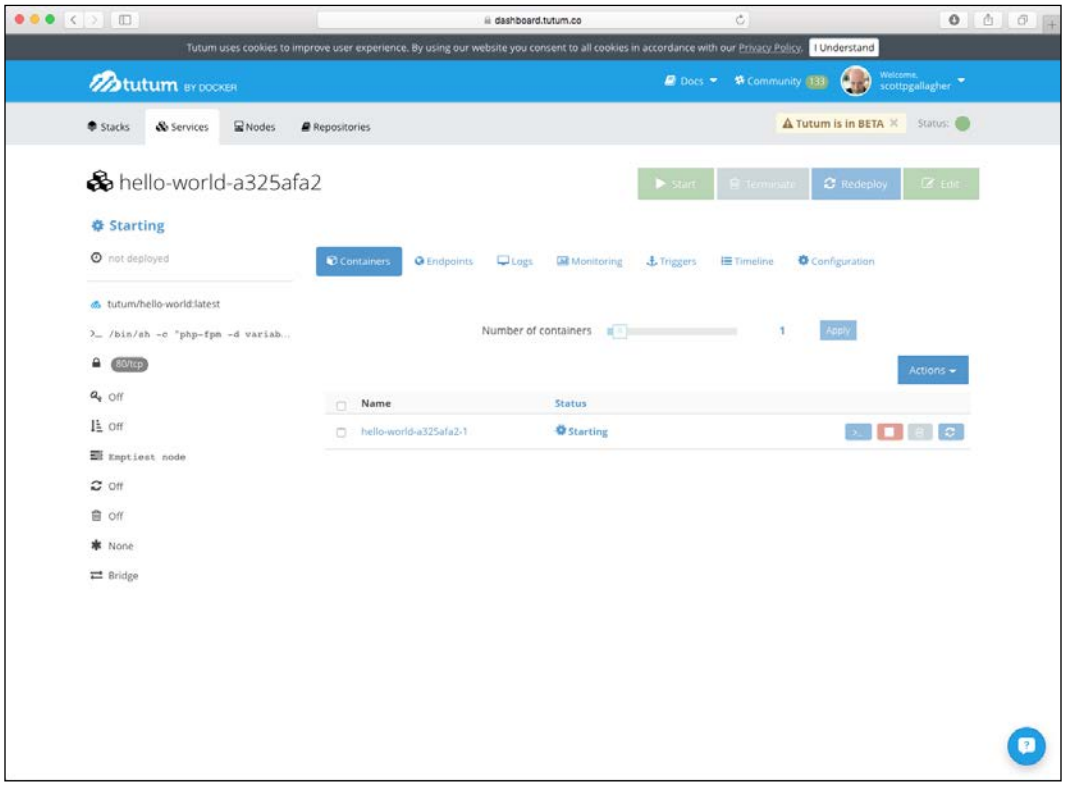


After clicking the **Select** button for it, we are taken to a screen similar to the following one; yours will vary depending upon what image you have selected.



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 module 1 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!

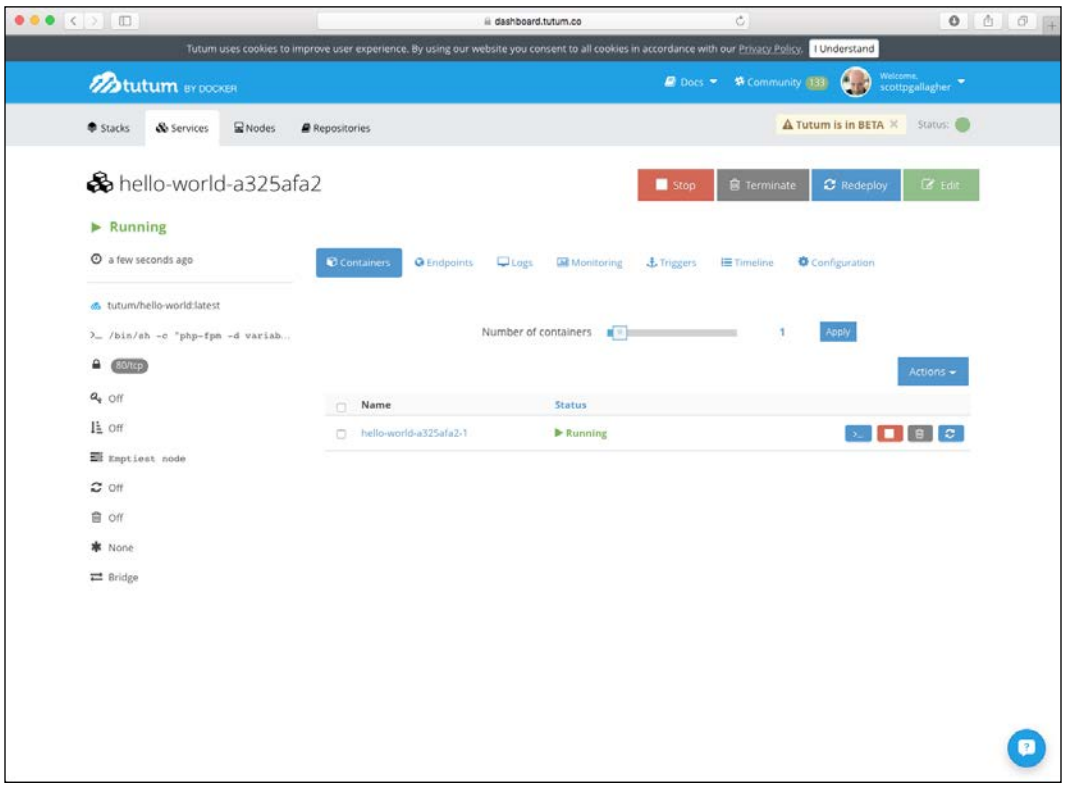
After we click on the button, we are taken to a screen similar to the one we saw when we were deploying our host node.



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

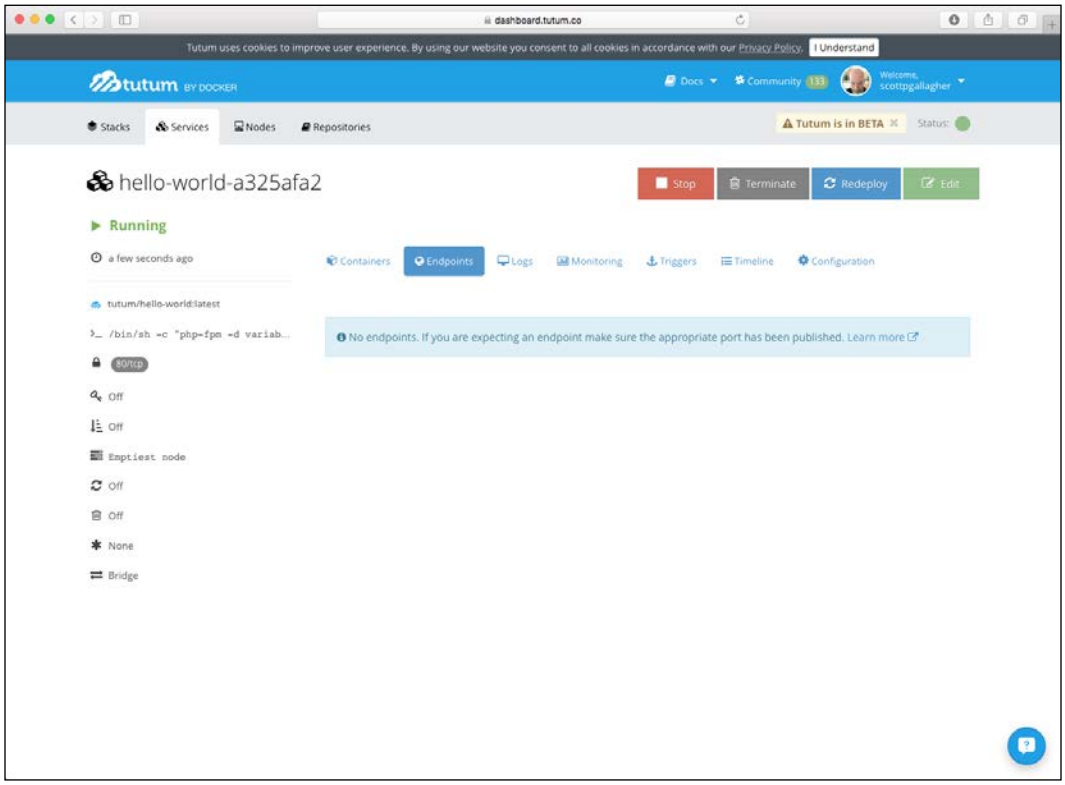
Once it has finished starting and is now in the running state, we can manipulate the container and do things such as stop, terminate, redeploy, or even edit the configuration of the container, and expand the number of containers that are running.



Now, let's take a look at the navigation menu for containers.

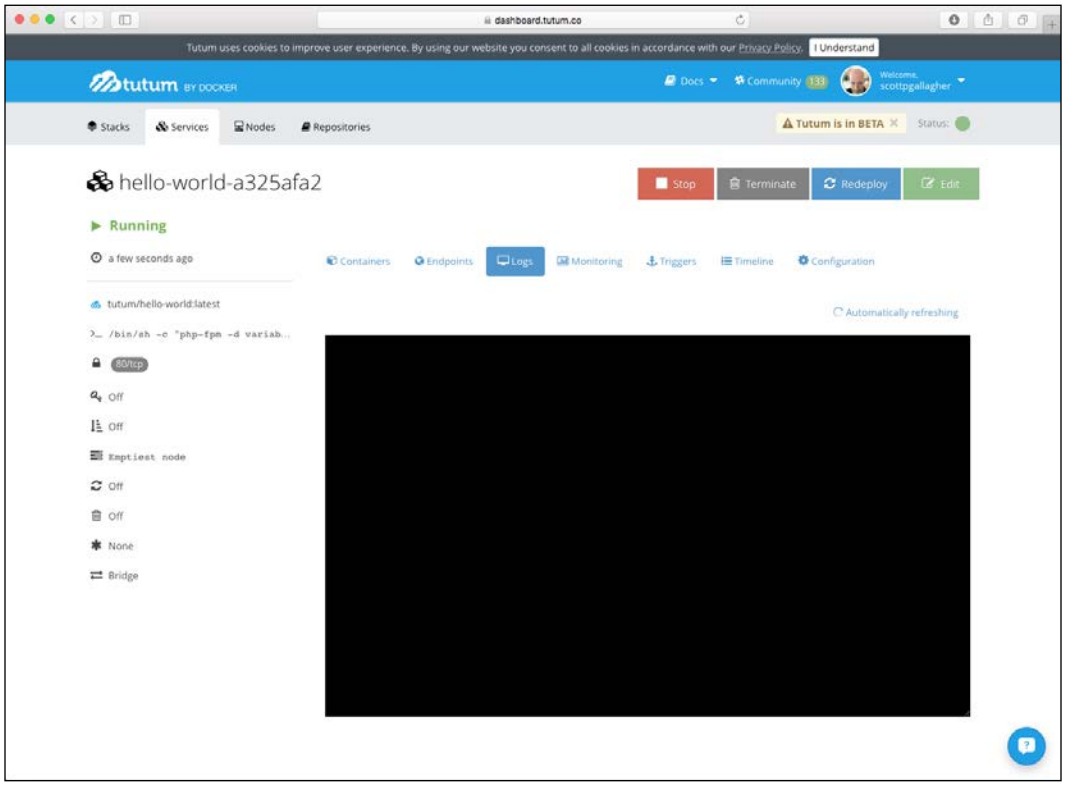
Endpoints

Again, the **Endpoints** screenshot will show us any port information pertaining to the running container.



Logs

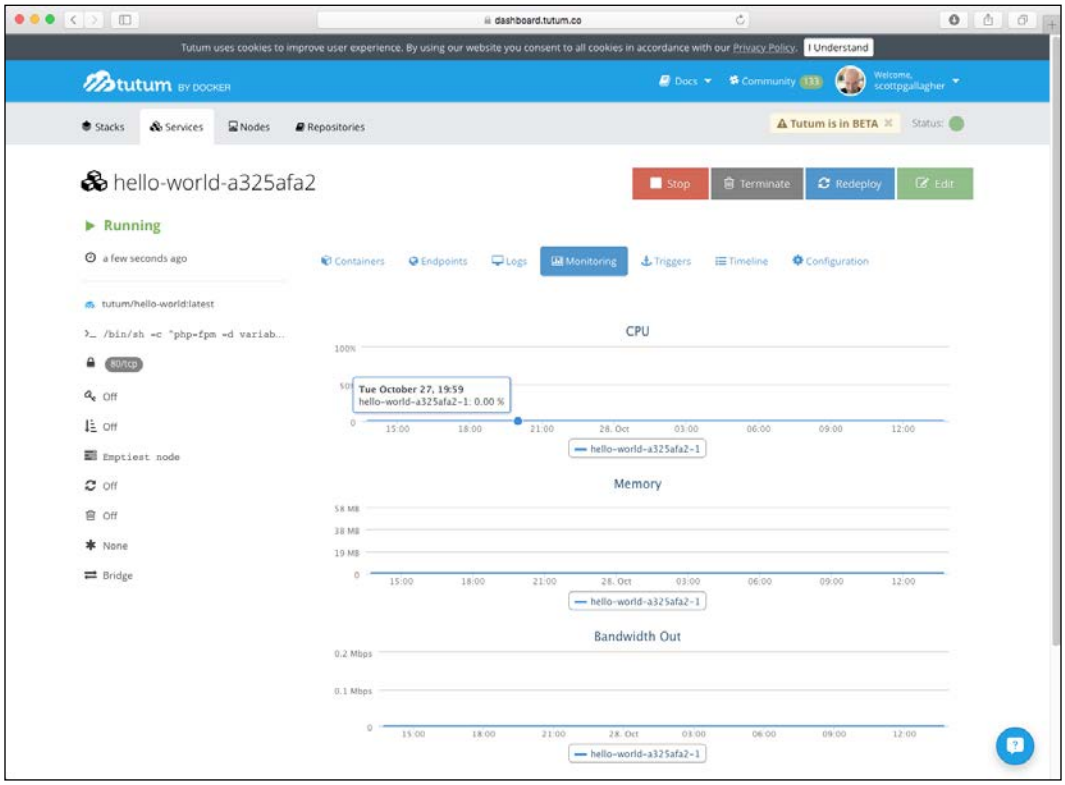
The **Logs** section will show us a running log of the screen output the container would have.



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

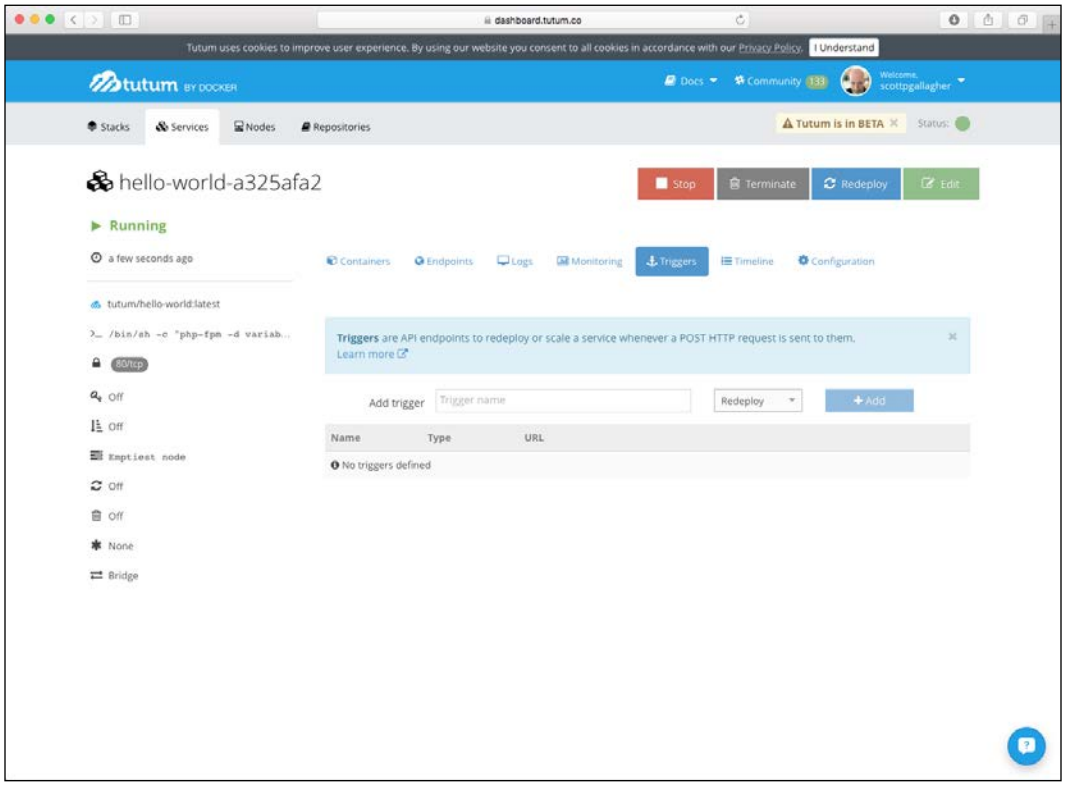
Next, we have the monitoring section that can show us the information we saw before in the **Nodes** section.



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

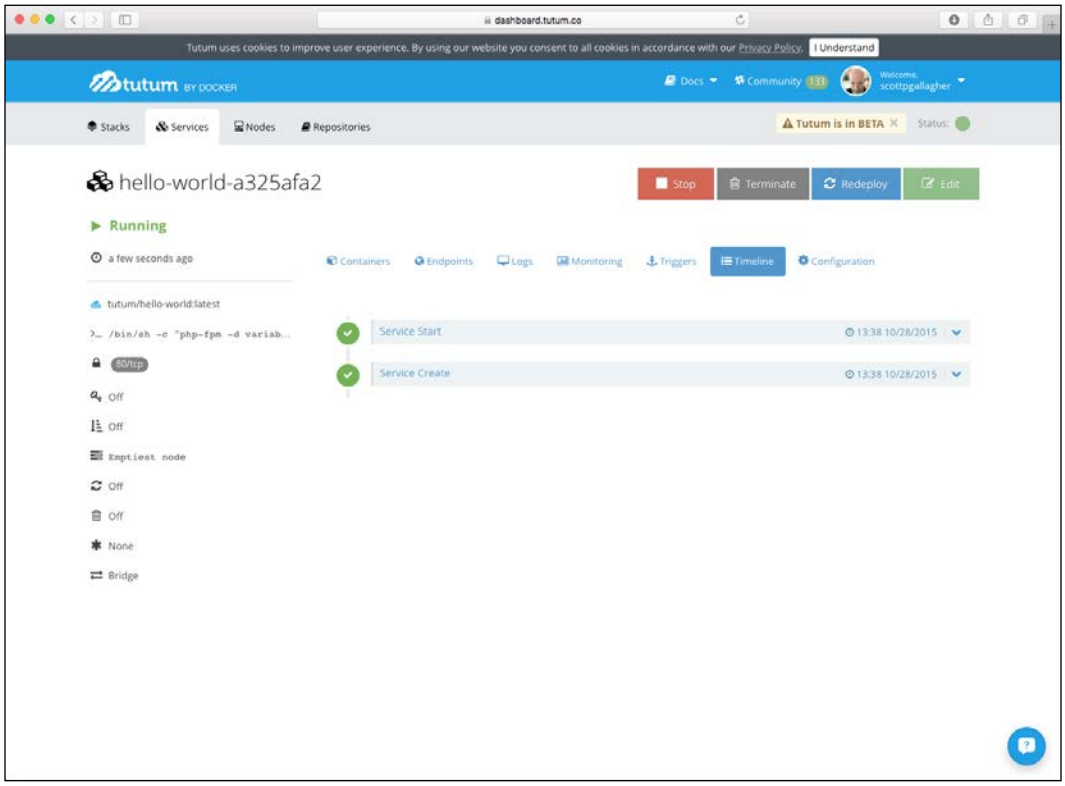
Next up is the **Triggers** section. Now, this section can come in handy if you are looking at scaling something based on the CPU usage that a container has.



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).

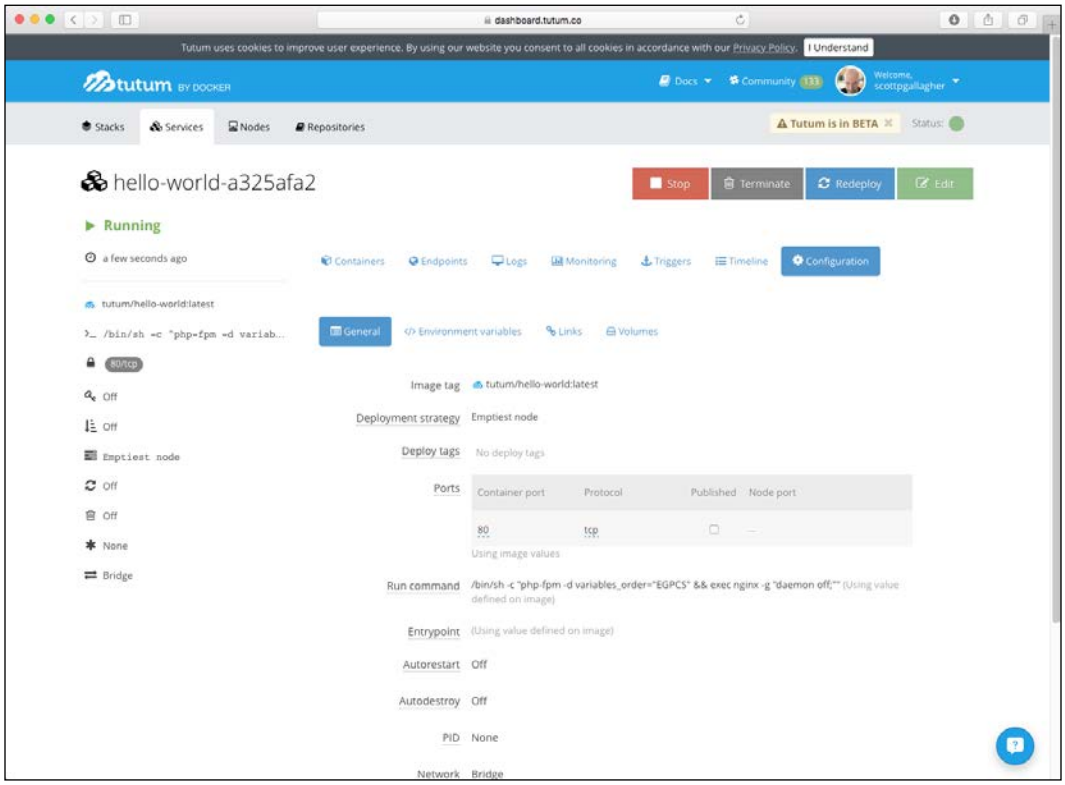
Timeline

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

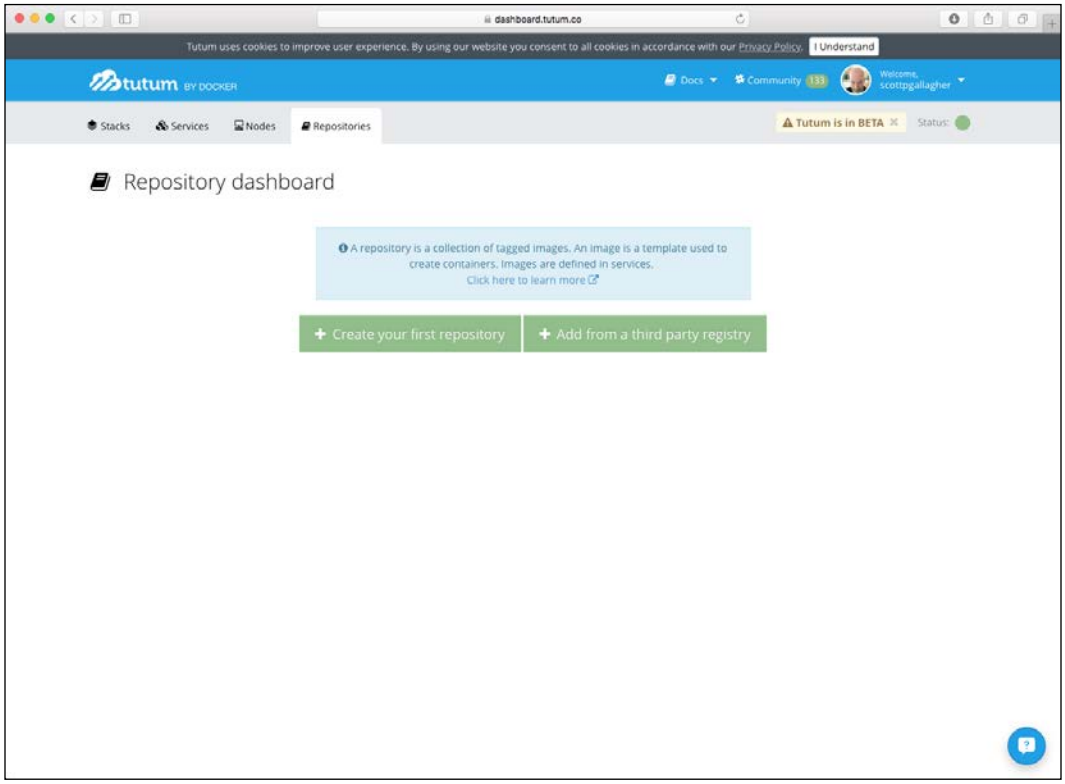
Lastly, we have the **Configuration** section that shows an overview of the container as a whole.



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

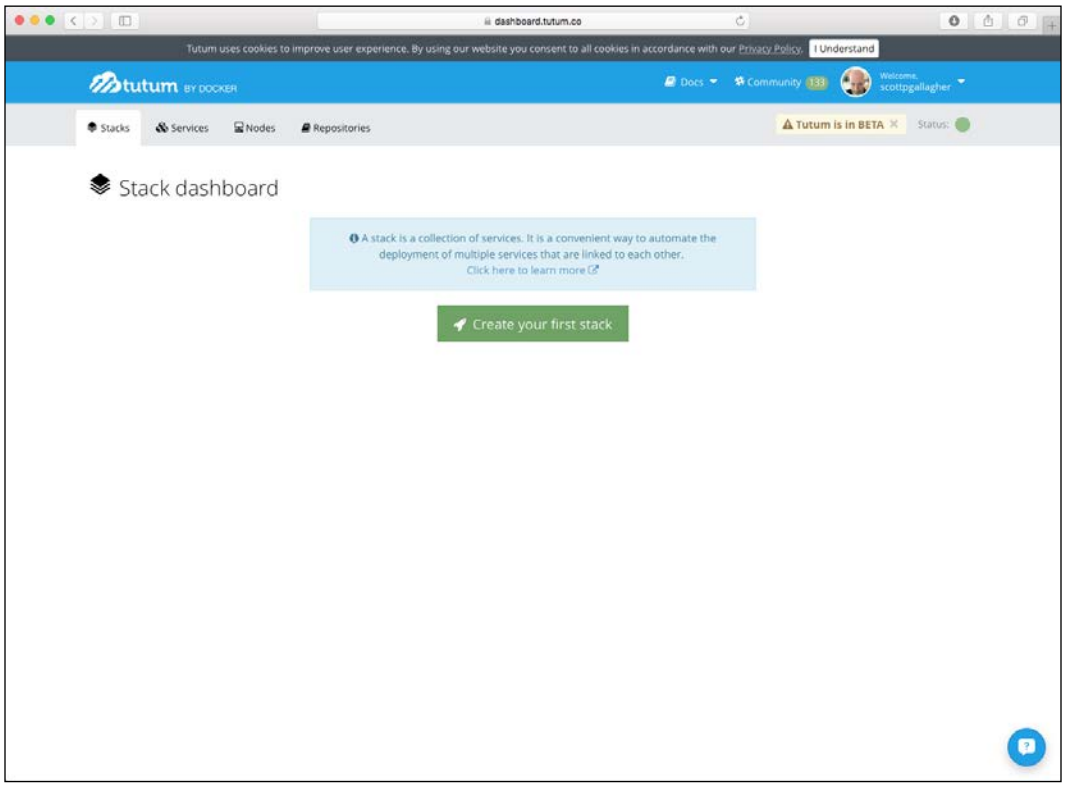
Let's take a look back at the navigation bar at the top and click on the **Repositories** tab.



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.

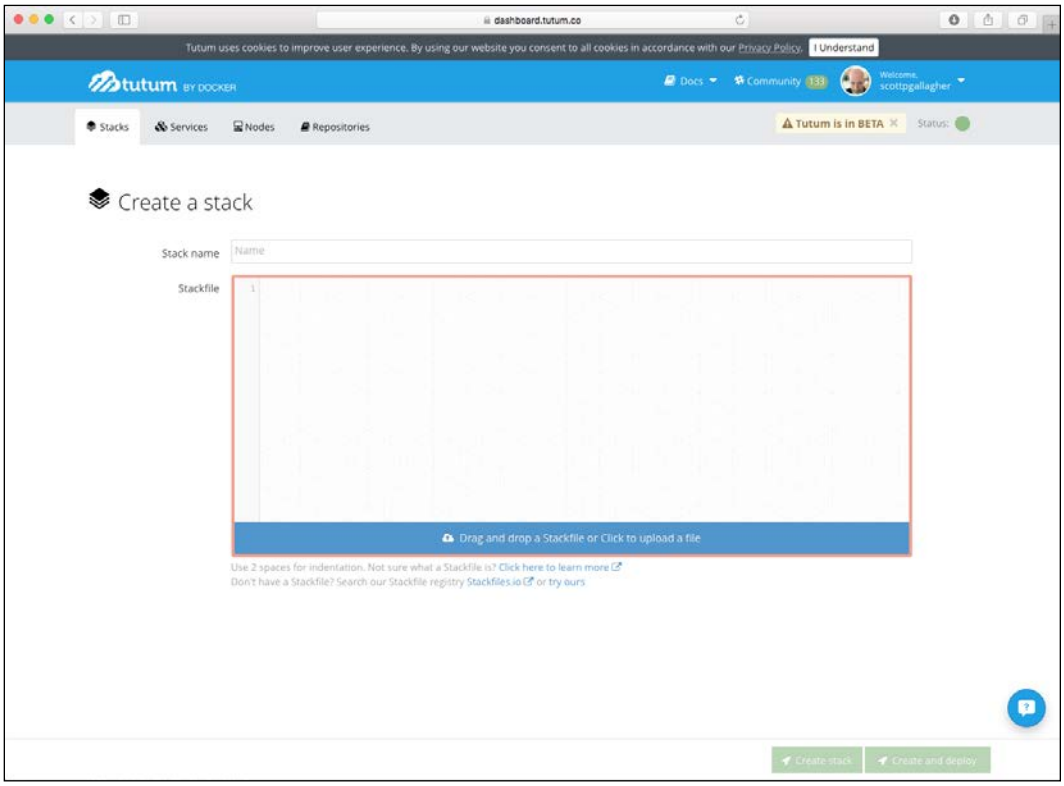
Stacks

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.



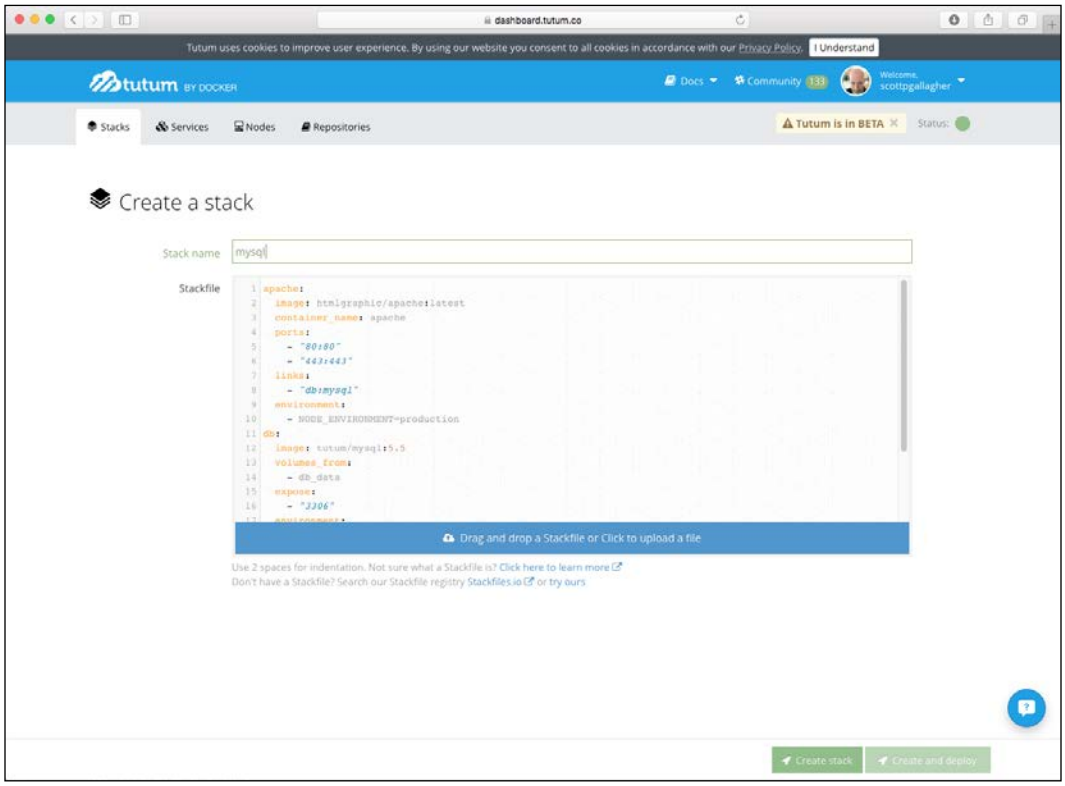
Let's take a look at this section, because it can be very useful while using development environments or for testing.

After we click on **Create your first stack**, we are taken to a page that is similar to the following screenshot:



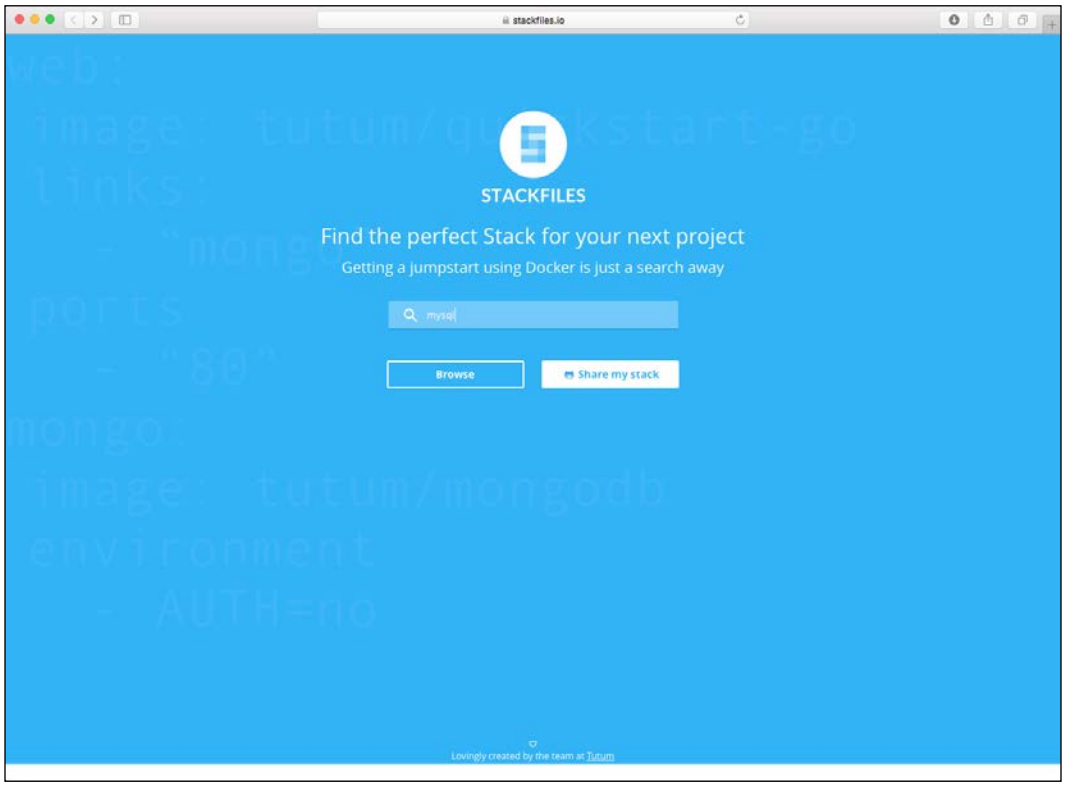
In this screenshot, we can see that we need pieces of information.

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`.



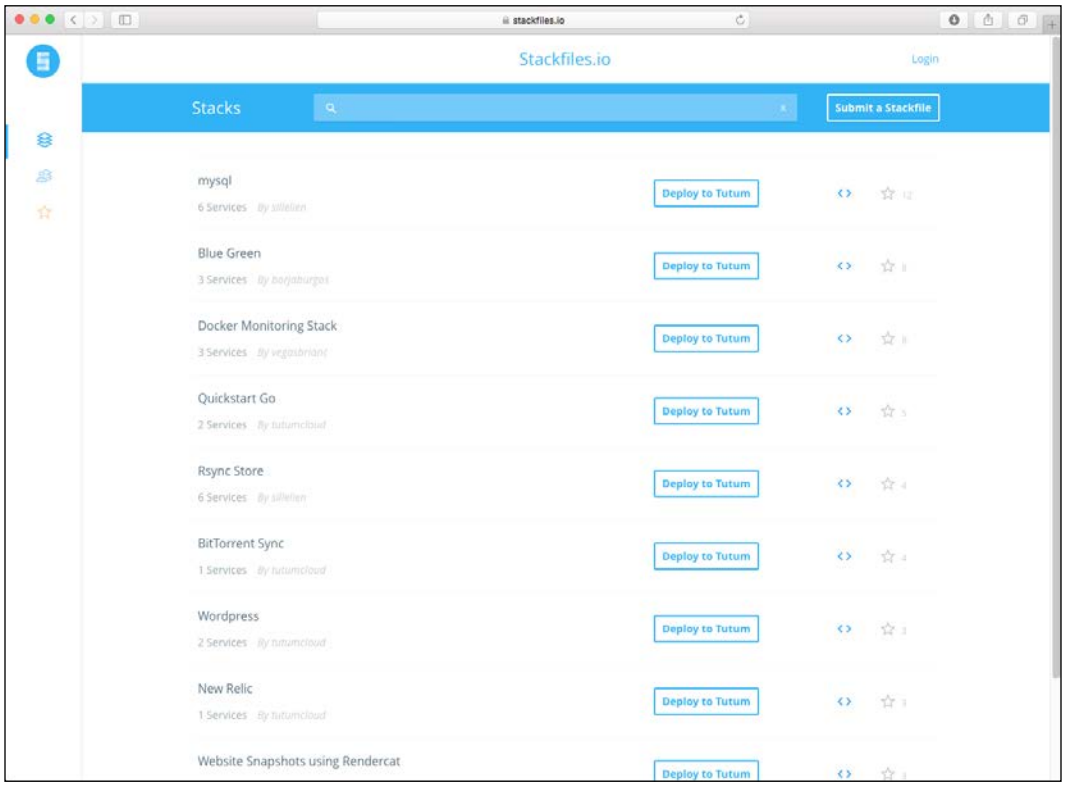
For our stackfile, we are going to use one of the resources that Tutum encourages us to explore. In the bottom section under the **Stackfile** field, there is an option to get a Stackfile from the Stackfile registry, which is located at `https://Stackfiles.io`.

Upon entering `stackfiles.io`, we are presented with an easy search box.



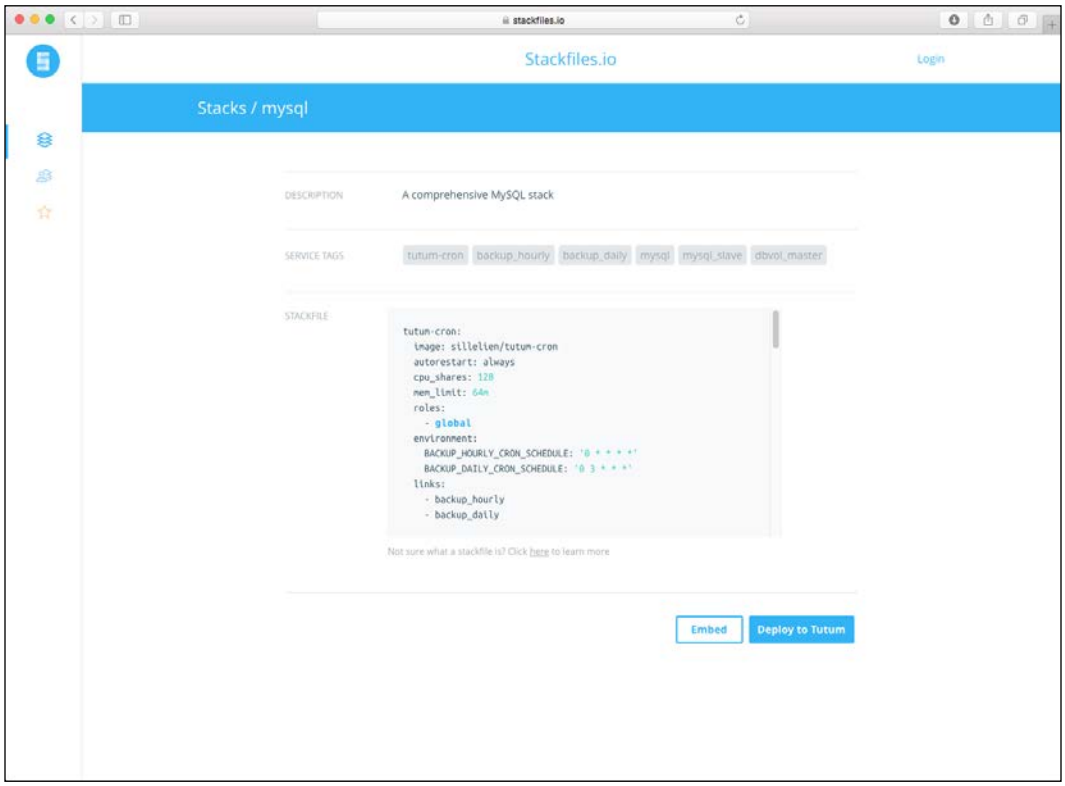
Again, for our test, we want to find the `mysql` stackfile, so we enter `mysql` in the box and click on **Browse**.

Now, for our example, we want a `mysql` one and we can see it right on the top.



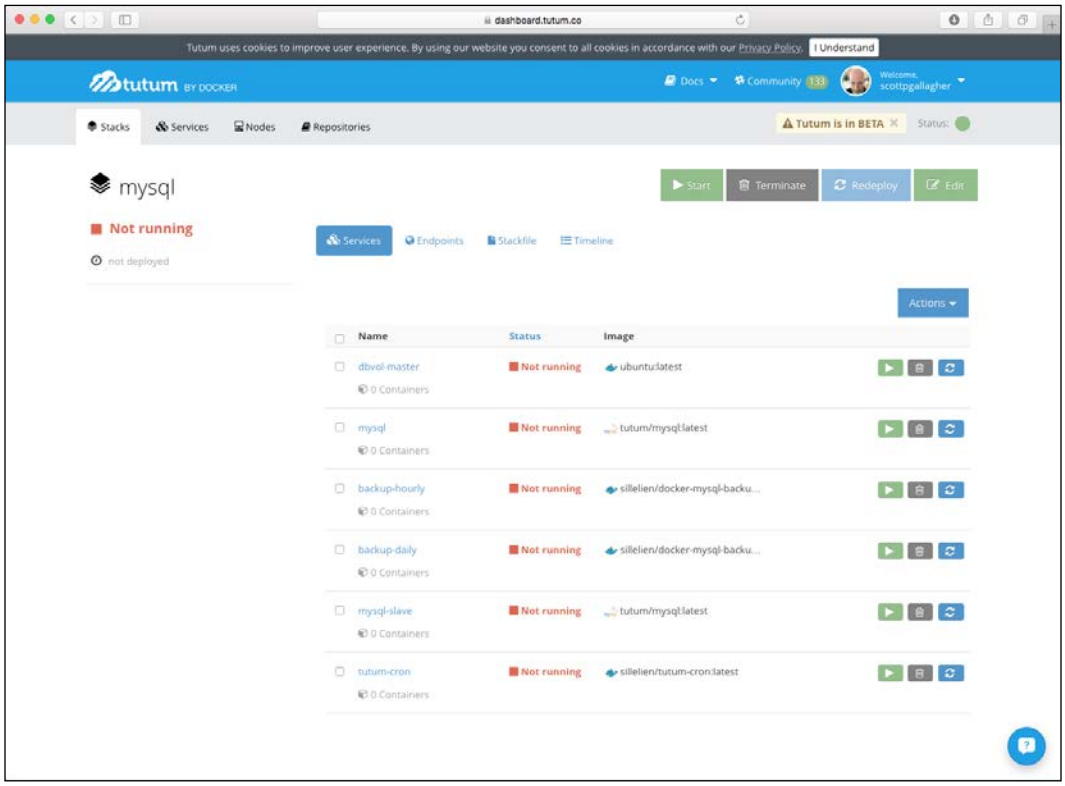
However, you could use a different one or search for one as well to see if there is one already done for you. Again, always work smarter, not harder!

So, if you drill into the `mysql` stackfile, you can see what all it is doing.



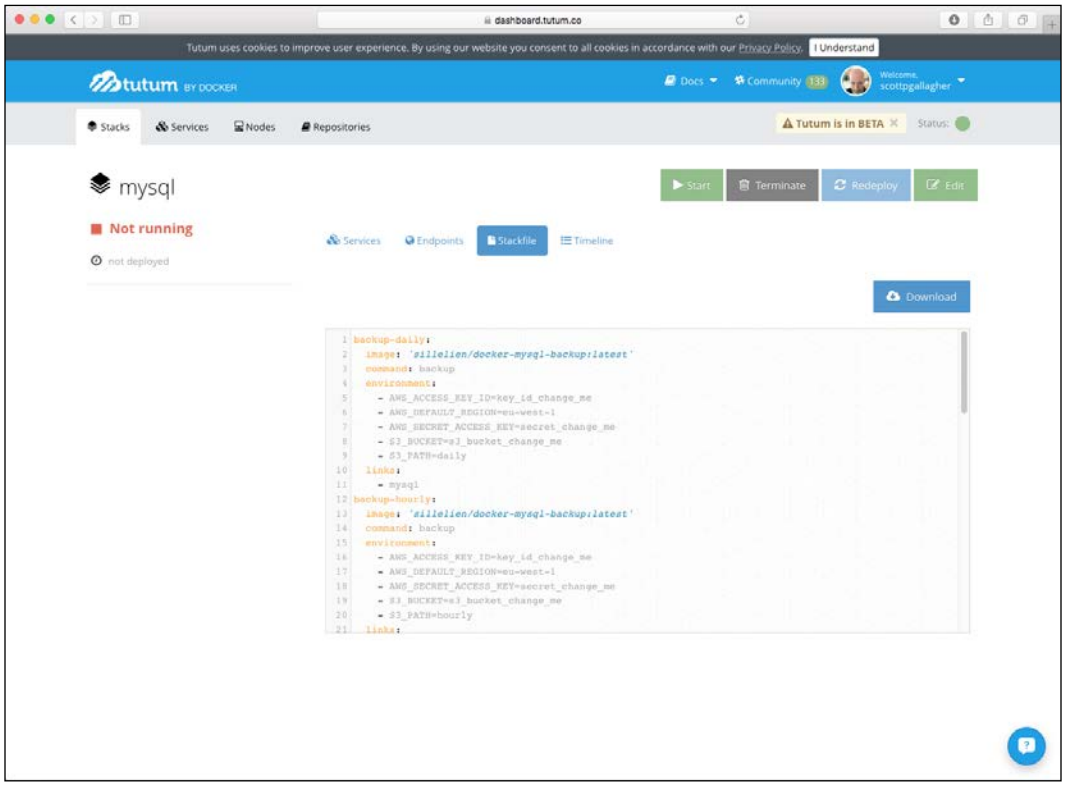
In our case, we are just going to copy this, go back to our Tutum stack deployment page, and paste it among the contents of the stackfile.

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 of Module 5 Chapter 4

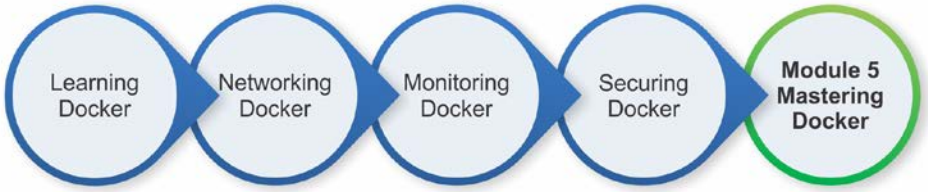
Ankita Thakur

Your Course Guide

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.

Your Progress through the Course So Far



5
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 module 1, 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 2*, *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/`.

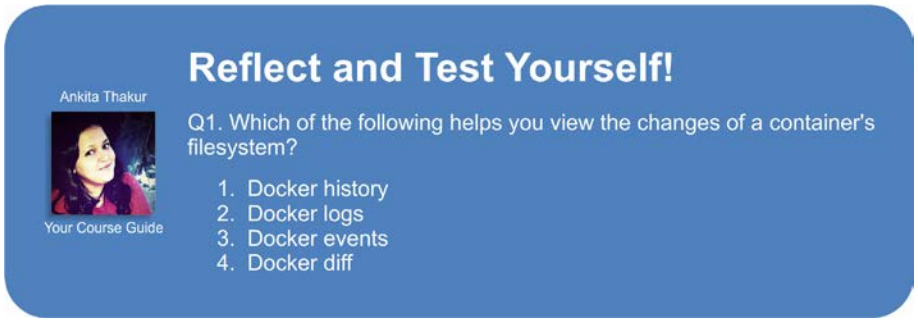
Remote API

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
```

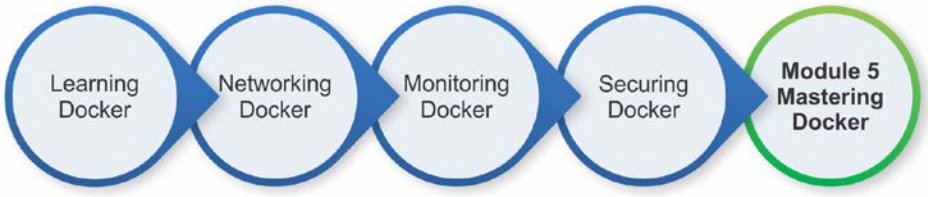
Summary of Module 5 Chapter 5

Ankita Thakur

Your Course Guide

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.

Your Progress through the Course So Far



A Final Run-Through

Here, we come to the end of our learning journey. Congratulations for doing well so far! I hope you had a smooth journey and gained a lot of knowledge on Docker. If you ever wanted to get started with Docker or you already knew about Docker but wanted to explore more, this course was designed in the way to help you achieve it. I'm sure you must have gained a lot of information and you'll start implementing it.

Now, let's take a small recap of what we have learned through this course. We covered five modules:

Ankita Thakur

Your Course Guide

We started off with the Learning Docker module wherein we dealt with the fundamentals of Docker. We talked about the Docker platform and how it simplifies and speeds up the process of realizing containerized workloads to be readily deployed and run on a variety of platforms. We covered step-by-step details on installing the Docker engine, creating a Docker container out of that image, and troubleshooting the Docker container. We provided the basic Docker terminologies needed to understand the output of Docker commands. We also covered other important aspects of Docker in detail: Docker Machine, Docker Swarm, and Docker Compose. Finally, we learned how to test and debug our applications.

After covering the fundamentals of Docker, we ventured into the Networking Docker module. We explained the essential components of Docker networking, which have evolved from coupling simple Docker abstractions and powerful network components, such as Linux bridges, Open vSwitch, and so on. We learned about IPv4, IPv6, and DNS configurations in Docker. We showed you how Docker networking works for Docker Swarm and Kubernetes. Finally, we explained Project Calico in detail, which provides scalable networking solutions that are based out of libnetwork and provides integration with Docker, Kubernetes, Mesos, bare-metal, and VMs, primarily.

Next, we moved on to the Monitoring Docker module that explains how different it is to monitor containers compared to more traditional servers such as virtual machines, bare metal machines, and cloud instances (Pets versus Cattle and Chickens versus Snowflakes). We introduced cAdvisor, how to install cAdvisor and start collecting metrics. We also looked at a traditional tool for monitoring services. Now, you should know your way around Zabbix and the various ways you can monitor your containers. Later on, you learned how to use Sysdig to both view your containers' performance metrics in real time and also record sessions to query later. Finally, we looked at the next steps you can take in monitoring your containers by talking about the benefits of adding alerting to your monitoring. Also, we covered some different scenarios and look at which type of monitoring is appropriate for each of them.

Moving on to the next module, Securing Docker, where we started off by discussing how to secure the first part of getting your Docker environment up and running, and that is by focusing on your Docker hosts. We explained hardening guides that are out there as well as different security measures/methods you can use to help secure the kernel that is being used to run your containers as it's important to secure it. Later on, we informed how well you have set up your Docker environment with the Docker Bench Security application, get recommendations for where you should focus your efforts to fix right away, and what you don't really have to fix right now, but should keep yourself aware of.

Ankita Thakur

Your Course Guide

Having sufficient knowledge of Docker, it was time to deploy Docker in your production environment and also learn some advanced concepts. We did this in our final module, Mastering Docker. We also focused on three GUI applications (Shipyard, Panamax, and Tutum) that you can utilize to set up and manage your Docker containers and images. Finally, we explained how to scale up your environment using discovery services, various Docker APIs, and how to contribute to Docker.
Also, we had interesting challenges and quizzes throughout this course. How did you find them? Hope it was interesting. Keep writing to us in case you have any feedback or queries. I wish you all the best for your future projects.

Keep learning and exploring until we meet again!

Reflect and Test Yourself! Answers

Module 1: Learning Docker

Chapter 1: Getting Started with Docker

Q1	3
Q2	2

Chapter 6: Running Your Private Docker Infrastructure

Q1	4
Q2	3

Chapter 7: Running Services In a Container

Q1	2
Q2	4
Q3	1

Chapter 8: Sharing Data with Containers

Q1	4

Chapter 9: Docker Machine

Q1	2

Chapter 10: Orchestrating Docker

Q1	3

Chapter 11: Docker Swarm

Q1	1

Chapter 12: Testing with Docker

Q1	4

Chapter 13: Debugging Containers

Q1	4
Q2	2

Module 2: Networking Docker

Chapter 1: Docker Networking Primer

Q1	4
Q2	3
Q3	3
Q4	2

Chapter 2: Docker Networking Internals

Q1	3
Q2	2
Q3	4
Q4	1
Q5	3
Q6	2

Chapter 3: Building Your First Docker Network

Q1	1

Chapter 4: Networking in a Docker Cluster

Q1	4
Q2	3
Q3	2
Q4	4
Q5	3

Chapter 5: Next Generation Networking Stack for Docker – libnetwork

Q1	3
Q2	2
Q3	4
Q4	2
Q5	3

Module 3: Monitoring Docker

Chapter 1: Introduction to Docker Monitoring

Q1	3

Chapter 3: Advanced Container Resource Analysis

Q1	2
Q2	1
Q3	1

Chapter 4: A Traditional Approach to Monitoring Containers

Q1	4
Q2	3

Chapter 5: Querying with Sysdig

Q1	2

Chapter 6: Exploring Third-Party Options

Q1	4
Q2	3
Q3	4

Chapter 7: Collecting Application Logs from within the Container

Q1	3

Module 4: Securing Docker

Chapter 2: Securing Docker Components

Q1	2

Chapter 3: Securing and Hardening Linux Kernels

Q1	2
Q2	3

Chapter 4, Docker Bench for Security

Q1	1
Q2	3

Chapter 5, Monitoring and Reporting Docker Security Incidents

Q1	1
Q2	3

Chapter 6, Using Docker's Built-in Security Features

Q1	2

Chapter 7, Securing Docker with Third-party Tools

Q1	5
Q2	2
Q3	3

Chapter 8, Keeping up Security

Q1	2
Q2	3

Module 5: Mastering Docker

Chapter 1, Docker in Production

Q1	2

Chapter 2, Shipyard

Q1	4

Chapter 5, Advanced Docker

Q1	4

Bibliography

This course is a blend of text and quizzes, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *Learning Docker*, *Pethuru Raj*, *Jeeva S. Chelladhurai*, and *Vinod Singh*
- *Orchestrating Docker*, *Shrikrishna Holla*
- *Build Your Own PaaS with Docker, Oskar Hane*
- *Docker Cookbook, Neependra Khare*
- *Learning Docker Networking*, *Rajdeep Dua*, *Vaibhav Kohli*, and *Santosh Kumar Konduri*
- *Monitoring Docker*, *Russ McKendrick*
- *Docker High Performance*, *Allan Espinosa*
- *Securing Docker*, *Scott Gallagher*
- *Mastering Docker*, *Scott Gallagher*

Thank you for buying

Docker

Creating Structured Containers

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, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `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, 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.