Neependra Khare

Docker Cookbook

80 hands-on recipes to efficiently work with the Docker 1.6 environment on Linux

Packt>

Docker Cookbook

80 hands-on recipes to efficiently work with the Docker 1.6 environment on Linux

Neependra Khare

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Docker Cookbook

First published: June 2015

Production reference: 1080615

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

ISBN 978-1-78398-486-2

`www.packtpub.com`

Credits

Author
Neependra Khare

Reviewers
Scott Collier
Julien Duponchelle
Allan Espinosa
Vishnu Gopal

Acquisition Editor
Kevin Colaco

Content Development Editor
Rohit Kumar Singh

Technical Editor
Vivek Arora

Copy Editors
Puja Lalwani
Laxmi Subramanian

Project Coordinator
Mary Alex

Proofreaders
Stephen Copestake
Safis Editing

Indexer
Tejal Soni

Graphics
Abhinash Sahu

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

About the Author

Neependra Khare is currently working as a principal performance engineer in Red Hat's system design and engineering team. He has more than 11 years of IT experience. Earlier, he worked as a system administrator, support engineer, and filesystem developer. He loves teaching. He has conducted a few corporate training sessions and taught full semester courses. He is also a co-organizer of the Docker Meetup Group, in Bangalore, India.

He lives with his wife and two-year-old daughter in Bangalore, India. His Twitter handle is `@neependra` and his personal website is `http://neependra.net/`. He has also created a website for the book, which you can visit at `http://dockercookbook.github.io/`.

I would like to thank my parents who fought all odds and made me the person I am today. My wife, Kanika, has been the first and best support for this book. She took care of our daughter, Navya, when I was away writing this book.

It has been a bigger piece than expected, as new features in Docker and the ecosystems around it were evolving very fast. I have learned a lot during the process and the Docker community (blogs, IRC, mailing list, and meet ups) has been of great help.

I am thankful to all the reviewers of the book, who gave their valuable time to go through the content, pointed out my mistakes, and provided me with great suggestions.

I would also like to thank my current employer who gave me some time to work on this book and my coworker Jeremy Eder (`@jeremyeder`) from whom I learned a lot.

About the Reviewers

Scott Collier is a senior principal system engineer in the systems design and engineering team at Red Hat. He is currently focused on product integration for anything that has to do with containers. He is a Red Hat Certified Architect (RHCA) with over 18 years of experience in IT.

He was also a technical reviewer on *The Docker Book*.

I would like to say thanks to Neependra for giving me the opportunity to collaborate on this book. It was a pleasure! I'd also like to thank my wife, Laura, for giving me the weekends to do this review.

Julien Duponchelle is a French engineer. He is a graduate of Epitech. During his work experience, he contributed to several open source projects and focused on tools, which make the work of IT teams easier.

After he directed the educational area at ETNA, a French IT school, he has accompanied several start-ups as a lead backend engineer and participated in many significant and successful fund raising events (Plizy and Youboox).

I would like to warmly thank, Maëlig, my girlfriend, for her benevolence and great patience at the time when I was working on this book or on open source projects in general, over so many evenings.

Allan Espinosa is an active open source contributor to various distributed system tools such as Docker and Chef. He maintains several Docker images for popular open source software that were popular before the official release from the upstream open source groups themselves.

He completed his master's of science in computer science from the University of Chicago. There, he worked on scaling data-intensive applications across supercomputing centers in the United States.

> I would like to thank my wife, Kana, for the continuous support that allowed me to spend significant time with this review project.

Vishnu Gopal has a degree in Human-Computer Interaction from University College London, and was a part of the team that built SlideShare, which was then acquired by LinkedIn. He has picked up a variety of skills in his career, from having worked as a software engineer to architecting products that have served millions of users a day. He blogs at `http://vishnugopal.com` and still likes to be known by his GitHub profile at `http://github.com/vishnugopal`.

> I would like to thank my wife, Uma, for her smiles and constant distractions that keep me sane.

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 the latest Docker version 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.

Docker and its ecosystem are evolving at a very high pace, so it is very important to understand the basics and build group up to adopt to new concepts and tools. With step-by-step instructions to practical and applicable recipes, *Docker Cookbook* will not only help you with the current version of Docker (1.6), but with the accompanying text it, will provide you with conceptual information to cope up with the minor changes in the new versions of Docker.
To know more about the book, visit `http://dockercookbook.github.io/`.

Docker™ is a registered trademark of Docker, Inc.

What this book covers

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

Chapter 2, *Working with Docker Containers*, covers most of the container-related recipes such as starting, stopping, and deleting containers. It also helps you to get low-level information about containers.

Chapter 3, *Working with Docker Images*, explains image-related operations such as pulling, pushing, exporting, importing, base image creation, and image creation using Dockerfiles. We also set up a private registry.

Chapter 4, *Network and Data Management for Containers*, covers recipes to connect a container with another container, in the external world. It also covers how we can share external storage from other containers and the host system.

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

Chapter 6, *Docker APIs and Language Bindings*, covers Docker remote APIs and Python language bindings as examples.

Chapter 7, *Docker Performance*, explains the performance 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. We look at CoreOS and Project Atomic as container-hosting platforms and then 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 get help related to Docker administration and development.

What you need for this book

The recipes in this cookbook will definitely run on Fedora 21-installed physical machines or VMs, as I used that configuration as the primary environment. As Docker can run on many platforms and distributions, you should be able to run most of the recipes without any problem. For a few recipes, you will also need Vagrant (`https://www.vagrantup.com/`) and Oracle Virtual Box (`https://www.virtualbox.org/`).

Who this book is for

Docker Cookbook is for developers, system administrators, and DevOps engineers who want to use Docker in his/her development, QA, or production environments.

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

Any experience in virtualization technologies such as KVM, XEN, and VMware will help the reader to relate with container technologies better, but it is not required.

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, we 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 the reader more knowledgeable about the recipe.

See also

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

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "You can use the `--driver/-d` option to create choosing one of many endpoints available for deployment."

A block of code is set as follows:

```
[Unit]
Description=MyApp
After=docker.service
Requires=docker.service

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill busybox1
ExecStartPre=-/usr/bin/docker rm busybox1
ExecStartPre=/usr/bin/docker pull busybox
ExecStart=/usr/bin/docker run --name busybox1 busybox /bin/sh -c
"while true; do echo Hello World; sleep 1; done"
```

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

```
[Service]
Type=notify
EnvironmentFile=-/etc/sysconfig/docker
EnvironmentFile=-/etc/sysconfig/docker-storage
ExecStart=/usr/bin/docker -d -H fd:// $OPTIONS
$DOCKER_STORAGE_OPTIONS
LimitNOFILE=1048576
LimitNPROC=1048576

[Install]
WantedBy=multi-user.target
```

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

```
$ docker pull fedora
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Go to the project home page and under the **APIs & auth** section, select **APIs**, and enable Google **Compute Engine API**."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

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

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

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

Customer support

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

Errata

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

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

Piracy

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

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

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

Questions

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

1
Introduction and Installation

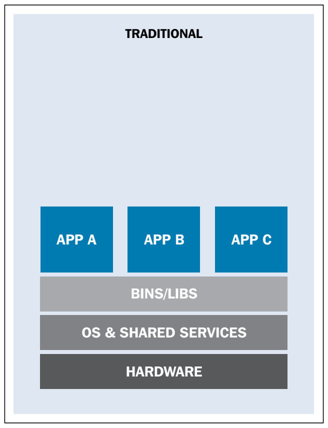
In this chapter, we will cover the following recipes:

- Verifying the requirements for Docker installation
- Installing Docker
- Pulling an image and running a container
- Adding a nonroot user to administer Docker
- Setting up the Docker host with Docker Machine
- Finding help with the Docker command line

Introduction

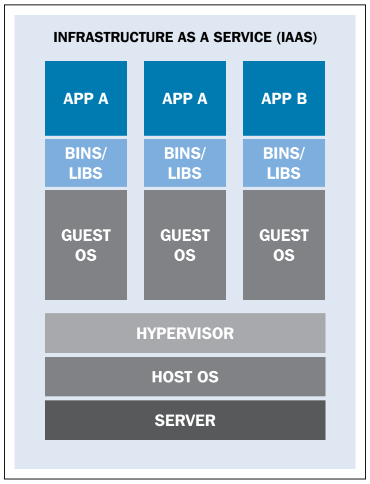
At the very start of the IT revolution, most applications were deployed directly on physical hardware, over the host OS. Because of that single user space, runtime was shared between applications. The deployment was stable, hardware-centric, and had a long maintenance cycle. It was mostly managed by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

The following diagram depicts such a setup:



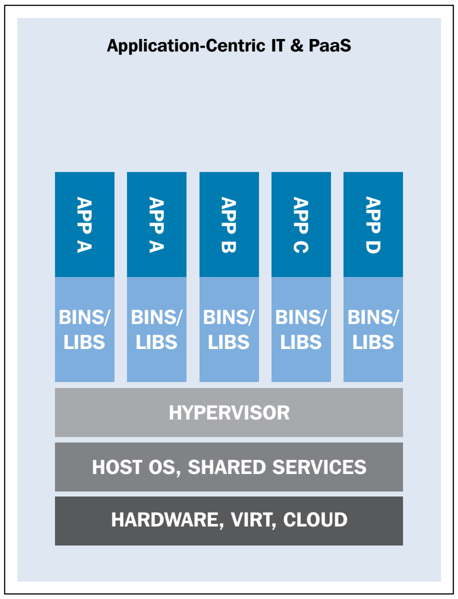
Traditional application deployment (`https://rhsummit.files.wordpress.com/2014/04/rhsummit2014-application-centric_packaging_with_docker_and_linux_containers-20140412riek7.pdf`)

To overcome the limitations set by traditional deployment, 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; that means we are responsible for managing the patches, security, and performance of that VM. With virtualization, applications are isolated at VM level and defined by the life cycle of VMs. This gives better return on 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 (`https://rhsummit.files.wordpress.com/2014/04/rhsummit2014-application-centric_packaging_with_docker_and_linux_containers-20140412riek7.pdf`)

After virtualization, we are now 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 a Linux OS. 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 reasons why we have not seen the mass adoption of containers in the last few years, even though we had the technology.

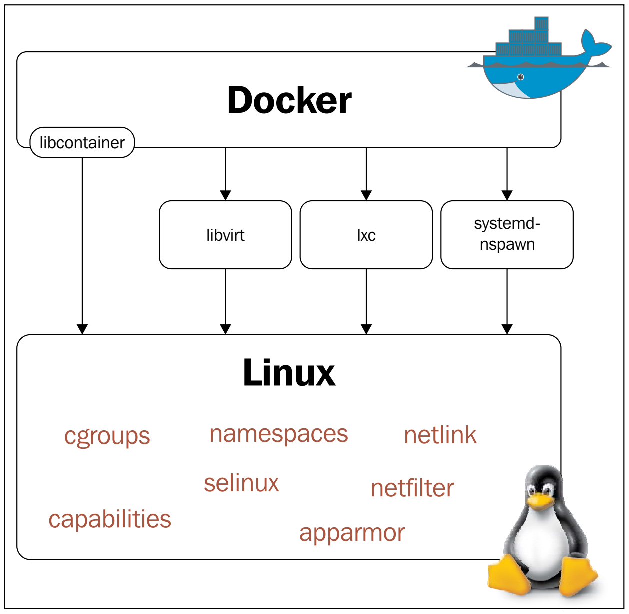


Application deployment with containers (`https://rhsummit.files.wordpress.com/2014/04/rhsummit2014-application-centric_packaging_with_docker_and_linux_containers-20140412riek7.pdf`)

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 Solomon Hykes, who is the current CTO of Docker, Inc., at dotCloud. 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 Linux's underlying kernel features which enable containerization. The following diagram depicts the execution drivers and kernel features used by Docker. We'll talk about execution drivers later. Let's look at some of the major kernel features that Docker uses:



The execution drivers and kernel features used by Docker (`http://blog.docker.com/wp-content/uploads/2014/03/docker-execdriver-diagram.png`)

Namespaces

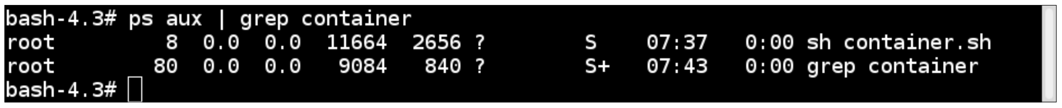
Namespaces are the building blocks of a container. There are different types of namespaces and each one of them isolates applications from each other. They are created using the clone system call. One can also attach to existing namespaces. Some of the namespaces used by Docker have been explained in 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.

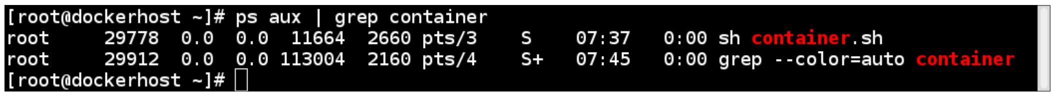
If there are two levels of hierarchy, then at the top level, we would see a process running inside the child namespace with a different PID. So, a process running in a child namespace would have two PIDs: one in the child namespace and the other in the parent namespace. For example, if we run a program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

On the container:



```
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:



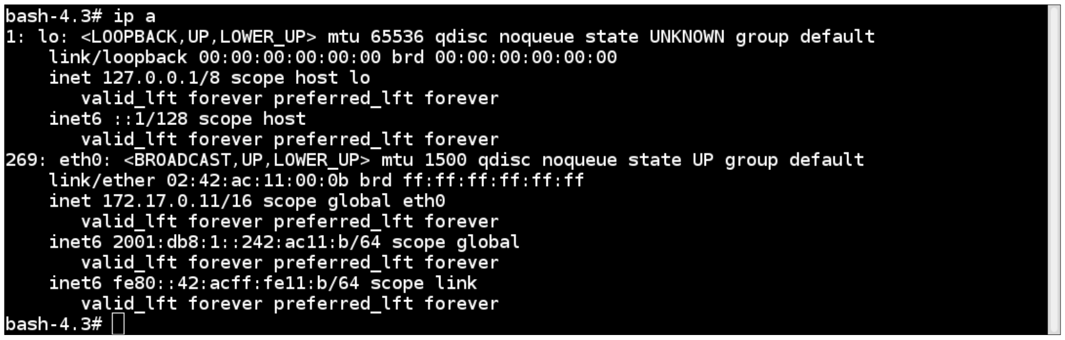
```
[root@dockerhost ~]# ps aux | grep container
root     29778  0.0  0.0  11664  2660 pts/3    S    07:37   0:00 sh container.sh
root     29912  0.0  0.0 113004  2160 pts/4    S+   07:45   0:00 grep --color=auto container
[root@dockerhost ~]# 
```

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.

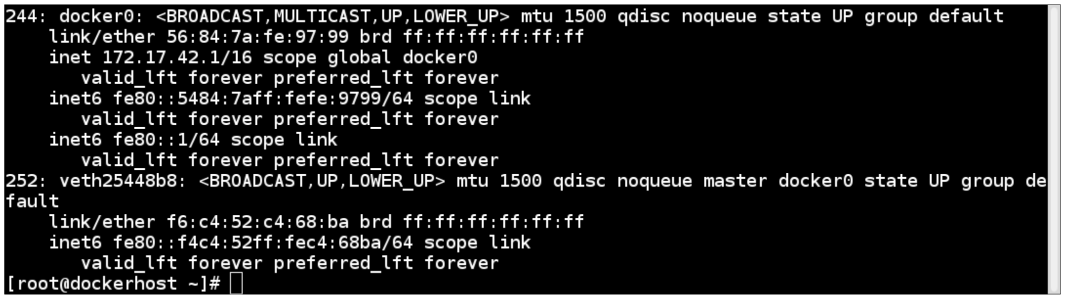
To enable networking in containers, we can create pairs of special interfaces in two different `net` namespaces and allow them to talk to each other. One end of the special interface resides inside the container and the other in the host system. Generally, the interface inside the container is named `eth0`, and in the host system, it is given a random name such as `vethcf1a`. These special interfaces are then linked through a bridge (`docker0`) on the host to enable communication between containers and route packets.

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



```
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# 
```

And in the host, it would look like the following:



```
244: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::5484:7aff:fefe:9799/64 scope link
       valid_lft forever preferred_lft forever
    inet6 fe80::1/64 scope link
       valid_lft forever preferred_lft forever
252: veth25448b8: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group de
fault
    link/ether f6:c4:52:c4:68:ba brd ff:ff:ff:ff:ff:ff
    inet6 fe80::f4c4:52ff:fec4:68ba/64 scope link
       valid_lft forever preferred_lft forever
[root@dockerhost ~]#
```

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

The ipc namespace

Inter Process Communication (**ipc**) 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

With just a chroot, one can inspect the relative paths of the system from a chrooted directory/namespace. The `mnt` namespace takes the idea of a chroot 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 can have a zero ID inside the container. This is because the `user` namespace allows per namespace mappings of users and groups IDs.

There are ways to share namespaces between the host and container and container and container. We'll see how to do that in subsequent chapters.

Cgroups

Control Groups (**cgroups**) provide resource limitations and accounting for containers. 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 the limiting of 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:

- **Resource limiting**: For example, one cgroup can be bound to specific CPUs, so all processes in that group would run off 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**: Freezing and restarting groups

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

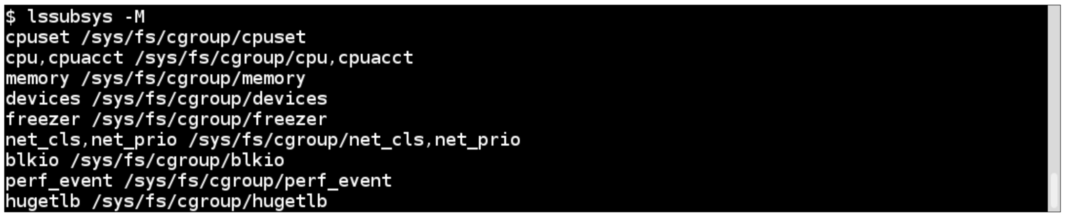
- **blkio**: It sets I/O access to and from block devices such as disk, SSD, and so on
- **Cpu**: It limits access to CPU
- **Cpuacct**: It generates CPU resource utilization
- **Cpuset**: It assigns the CPUs on a multicore system to tasks in a cgroup
- **Devices**: It devises access to a set of tasks in a cgroup
- **Freezer**: It suspends or resumes tasks in a cgroup
- **Memory**: It 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` in fedora, run the following command to install the required packages:

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

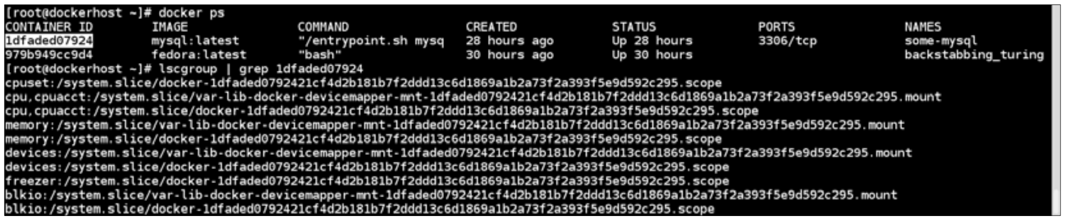
Once installed, you can get the list of subsystems and their mount point in the pseudo filesystem with the following command:

```
$ lssubsys -M
```



```
$ lssubsys -M
cpuset /sys/fs/cgroup/cpuset
cpu,cpuacct /sys/fs/cgroup/cpu,cpuacct
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
blkio /sys/fs/cgroup/blkio
perf_event /sys/fs/cgroup/perf_event
hugetlb /sys/fs/cgroup/hugetlb
```

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 from the following command:



```
[root@dockerhost ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                CREATED             STATUS              PORTS               NAMES
1dfaded07924        mysql:latest        "/entrypoint.sh mysq   28 hours ago        Up 28 hours         3306/tcp            some-mysql
979b949cc9d4        fedora:latest       "bash"                 30 hours ago        Up 30 hours                             backstabbing_turing
[root@dockerhost ~]# lscgroup | grep 1dfaded07924
cpuset:/system.slice/docker-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.scope
cpu,cpuacct:/system.slice/var-lib-docker-devicemapper-mnt-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.mount
cpu,cpuacct:/system.slice/docker-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.scope
memory:/system.slice/var-lib-docker-devicemapper-mnt-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.mount
memory:/system.slice/docker-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.scope
devices:/system.slice/var-lib-docker-devicemapper-mnt-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.mount
devices:/system.slice/docker-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.scope
freezer:/system.slice/docker-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.scope
blkio:/system.slice/var-lib-docker-devicemapper-mnt-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.mount
blkio:/system.slice/docker-1dfaded0792421cf4d2b181b7f2ddd13c6d1869a1b2a73f2a393f5e9d592c295.scope
```

For more details, visit `https://docs.docker.com/articles/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 which is used by the container's runtime environment. Look at 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, vfs, and DeviceMapper.

Docker can work with different execution drivers, such as `libcontainer`, `lxc`, and `libvirt` to manage containers. The default execution driver is `libcontainer`, which comes with Docker out of the box. It can manipulate namespaces, control groups, capabilities, and so on for Docker.

Verifying the requirements for Docker installation

Docker is supported on many Linux platforms, such as RHEL, Ubuntu, Fedora, CentOS, Debian, Arch Linux, and so on. It is also supported on many cloud platforms, such as Amazon EC2, Rackspace Cloud, and Google Compute Engine. With the help of a virtual environment, Boot2Docker, it can also run on OS X and Microsoft Windows. A while back, Microsoft announced that it would add native support to Docker on its next Microsoft Windows release.

In this recipe, let's verify the requirements for Docker installation. We will check on the system with Fedora 21 installation, though the same steps should work on Ubuntu as well.

Getting ready

Log in as root on the system with Fedora 21 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
   3.18.7-200.fc21.x86_64
   ```

3. Running kernel should support an appropriate storage backend. Some of these are VFS, DeviceMapper, AUFS, Btrfs, and OverlayFS.

   Mostly, the default storage backend or driver is devicemapper, which uses the device-mapper thin provisioning module to implement layers. It should be installed by default on the majority of Linux platforms. To check for device-mapper, you can run the following command:

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

   In most distributions, AUFS would require a modified kernel.

4. Support for cgroups and namespaces are in 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 Fedora, I can do something like the following:

   ```
   $ grep -i namespaces /boot/config-3.18.7-200.fc21.x86_64
   CONFIG_NAMESPACES=y
   $ grep -i cgroups /boot/config-3.18.7-200.fc21.x86_64
   CONFIG_CGROUPS=y
   ```

How it works...

With the preceding commands, we verified the requirements for Docker installation.

See also

- Installation document on the Docker website at `https://docs.docker.com/installation/`

Installing Docker

As there are many distributions which support Docker, we'll just look at the installation steps on Fedora 21 in this recipe. For others, you can refer to the installation instructions mentioned in the *See also* section of this recipe. Using Docker Machine, we can set up Docker hosts on local systems, on cloud providers, and other environments very easily. We'll cover that in a different recipe.

Getting ready

Check for the prerequisites mentioned in the previous recipe.

How to do it...

1. Install Docker using yum:

   ```
   $  yum -y install docker
   ```

How it works...

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

There's more...

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

- To start the service:
  ```
  $ systemctl start docker
  ```
- To verify the installation:
  ```
  $ docker info
  ```
- To update the package:
  ```
  $ yum -y update docker
  ```
- To enable the service start at boot time:
  ```
  $ systemctl enable docker
  ```
- To stop the service:
  ```
  $ systemctl stop docker
  ```

See also

- The installation document is on the Docker website at `https://docs.docker.com/installation/`

Pulling an image and running a container

I am borrowing this recipe from the next chapter to introduce some concepts. Don't worry if you don't find all the explanation in this recipe. We'll cover all 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

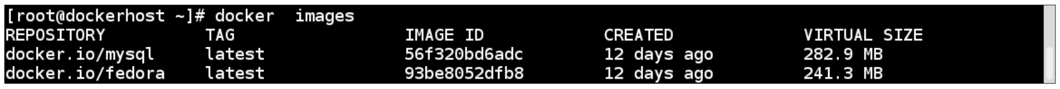
Get access to a system with Docker installed.

How to do it...

1. To pull an image, run the following command:
   ```
   $ docker pull fedora
   ```

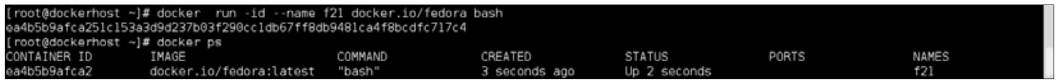
2. List the existing images by using the following command:

```
$ docker images
```



```
[root@dockerhost ~]# docker  images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
docker.io/mysql     latest              56f320bd6adc        12 days ago         282.9 MB
docker.io/fedora    latest              93be8052dfb8        12 days ago         241.3 MB
```

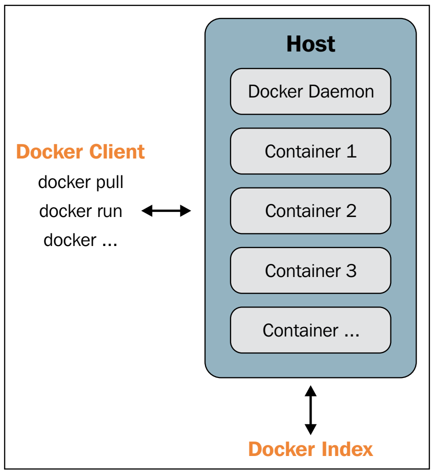
3. Create a container using the pulled image and list the containers as:



```
[root@dockerhost ~]# docker  run -id --name f21 docker.io/fedora bash
ea4b5b9afca251c153a3d9d237b03f290cc1db67ff8db9481ca4f8bcdfc717c4
[root@dockerhost ~]# docker ps
CONTAINER ID        IMAGE                    COMMAND             CREATED             STATUS              PORTS               NAMES
ea4b5b9afca2        docker.io/fedora:latest  "bash"              3 seconds ago       Up 2 seconds                            f21
```

How it works...

Docker has client-server architecture. Its binary consists of the Docker client and server daemon, and it can reside in the same host. The client can communicate via sockets or the 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 the public or local index to get the images requested by the client:



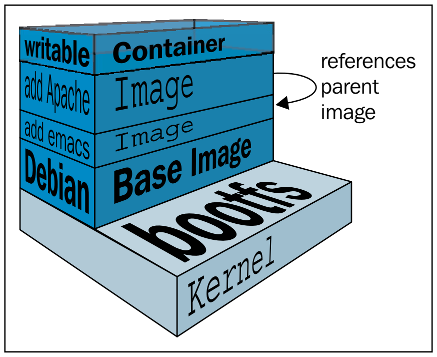
Docker client-server architecture (`https://docs.docker.com/introduction/understanding-docker/`)

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 Index and downloads the image. Once downloaded, we can run it.

There's more...

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

- **Images**: Docker images are read-only templates and they give us containers during runtime. There is the notion of a base image and layers on top of it. For example, we can have a base image of Fedora 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 following figure, **Debian** is the base image and **emacs** and **Apache** are the two layers added on top of it. They are highly portable and can be shared easily:



Docker Image layers (`http://docs.docker.com/terms/images/docker-filesystems-multilayer.png`)

  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 from where 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, search, tagging, and all that nice stuff that's in the public web interface of the Docker registry.
- **Containers**: Containers are running images that are created by combining the base image and the layers on top of it. They contain everything needed to run an application. As shown in preceding diagram, a temporary layer is also added while starting the container, which would get discarded if not committed after the container is stopped and deleted. If 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

- The documentation on the Docker website at `http://docs.docker.com/introduction/understanding-docker/`
- With Docker 1.6, the Docker community and Microsoft Windows released a Docker native client for Windows `http://azure.microsoft.com/blog/2015/04/16/docker-client-for-windows-is-now-available`

Adding a nonroot user to administer Docker

For ease of use, we can allow a nonroot user to administer Docker by adding them to a Docker group.

Getting ready

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

   ```
   $ sudo group add docker
   ```

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

   ```
   $ useradd dockertest
   ```

How to do it…

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

```
$ sudo  gpasswd -a dockertest docker
```

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. This can be the security risk. Visit *Chapter 9*, *Docker Security* for more details.

Setting up the Docker host with Docker Machine

Earlier this year, Docker released Orchestration tools (`https://blog.docker.com/2015/02/orchestrating-docker-with-machine-swarm-and-compose/`) and Machine, Swarm, and Compose deploy containers seamlessly. In this recipe, we'll cover Docker Machine and look at the others in later chapters. Using the Docker Machine tool (`https://github.com/docker/machine/`), you can set up Docker hosts locally on cloud with one command. It is currently in beta mode and not recommended for production use. It supports environments such as VirtualBox, OpenStack, Google, Digital Ocean, and others. For a complete list, you can visit `https://github.com/docker/machine/tree/master/drivers`. Let's use this tool and set up a host in Google Cloud.

> We will be using Docker Machine just for this recipe. Recipes mentioned in this or other chapters may or may not work on the host set up by Docker Machine.

Getting ready

Docker Machine does not appear with the default installation. You need to download it from its GitHub releases link (`https://github.com/docker/machine/releases`). Please check the latest version and distribution before downloading. As a root user, download the binary and make it executable:

```
$ curl -L
https://github.com/docker/machine/releases/download/v0.2.0/docker-machine_linux-amd64 > /usr/local/bin/docker-machine
$ chmod a+x  /usr/local/bin/docker-machine
```

If you don't have an account on **Google Compute Engine** (**GCE**), then you can sign up for a free trial (`https://cloud.google.com/compute/docs/signup`) to try this recipe. I am assuming that you have a project on GCE and have the Google Cloud SDK installed on the system on which you downloaded Docker Machine binary. If not, then you can follow these steps:

1. Set up the Google Cloud SDK on your local system:
   ```
   $ curl https://sdk.cloud.google.com | bash
   ```
2. Create a project on GCE (`https://console.developers.google.com/project`) and get its project ID. Please note that the project name and its ID are different.
3. Go to the project home page and under the **APIs & auth** section, select **APIs**, and enable Google **Compute Engine API**.

How to do it...

1. Assign the project ID we collected to a variable, `GCE_PROJECT`:

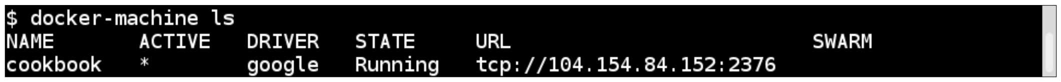
   ```
   $ export  GCE_PROJECT="<Your Project ID>"
   ```

2. Run the following command and enter the code which is provided on the popped up web browser:

   ```
   $ docker-machine  create -d google --google-
   project=$GCE_PROJECT  --google-machine-type=n1-standard-2 --
   google-disk-size=50 cookbook
   INFO[0000] Opening auth URL in browser.
   .......
   ......
   INFO[0015] Saving token in
   /home/nkhare/.docker/machine/machines/cookbook/gce_token


   INFO[0015] Creating host...
   INFO[0015] Generating SSH Key
   INFO[0015] Creating instance.
   INFO[0016] Creating firewall rule.
   INFO[0020] Waiting for Instance...
   INFO[0066] Waiting for SSH...
   INFO[0066] Uploading SSH Key
   INFO[0067] Waiting for SSH Key
   INFO[0224] "cookbook" has been created and is now the active
   machine.
   INFO[0224] To point your Docker client at it, run this in your
   shell: eval "$(docker-machine_linux-amd64 env cookbook)"
   ```

3. List the existing hosts managed by Docker Machine:

   ```
   $ ./docker-machine_linux-amd64 ls
   ```

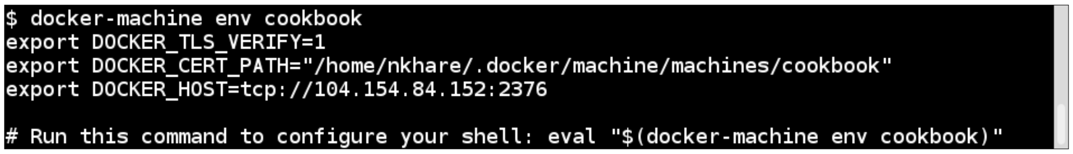


```
$ docker-machine ls
NAME       ACTIVE   DRIVER   STATE     URL                         SWARM
cookbook   *        google   Running   tcp://104.154.84.152:2376
```

You can manage multiple hosts with Docker Machine. The `*` indicates the active one.

4. To display the commands to set up the environment for the Docker client:

   ```
   $  ./docker-machine_linux-amd64 env cookbook
   ```



```
$ docker-machine env cookbook
export DOCKER_TLS_VERIFY=1
export DOCKER_CERT_PATH="/home/nkhare/.docker/machine/machines/cookbook"
export DOCKER_HOST=tcp://104.154.84.152:2376

# Run this command to configure your shell: eval "$(docker-machine env cookbook)"
```

   So, if you point the Docker client with the preceding environment variables, we would connect to the Docker daemon running on the GCE.

5. And to point the Docker client to use our newly created machine, run the following command:

   ```
   $ eval "$(./docker-machine_linux-amd64 env  cookbook)"
   ```

From now on, all the Docker commands will run on the machine we provisioned on GCE, until the preceding environment variables are set.

How it works...

Docker Machine connects to the cloud provider and sets up a Linux VM with Docker Engine. It creates a `.docker/machine/` directory under the current user's home directory to save the configuration.

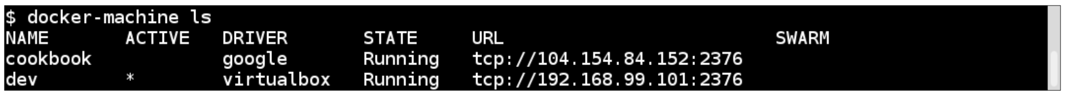
There's more...

Docker Machine provides management commands, such as `create`, `start`, `stop`, `restart`, `kill`, `remove`, `ssh`, and others to manage machines. For detailed options, look for the help option of Docker Machine:

```
$ docker-machine  -h
```

You can use the `--driver/-d` option to create choosing one of the many endpoints available for deployment. For example, to set up the environment with VirtualBox, run the following command:

```
$ docker-machine create --driver virtualbox dev
```



```
$ docker-machine ls
NAME       ACTIVE   DRIVER       STATE     URL                         SWARM
cookbook            google       Running   tcp://104.154.84.152:2376
dev        *        virtualbox   Running   tcp://192.168.99.101:2376
```

Here, `dev` is the machine name. By default, the latest deployed machine becomes primary.

See also

- Documentation on the Docker website at `https://docs.docker.com/machine/`
- Guide to setting up Docker on Google Compute Engine at `https://docs.docker.com/installation/google/`

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

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 any 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 show help.

See also

- Documentation on the Docker website at `http://docs.docker.com/reference/commandline/cli/`

2
Working with Docker Containers

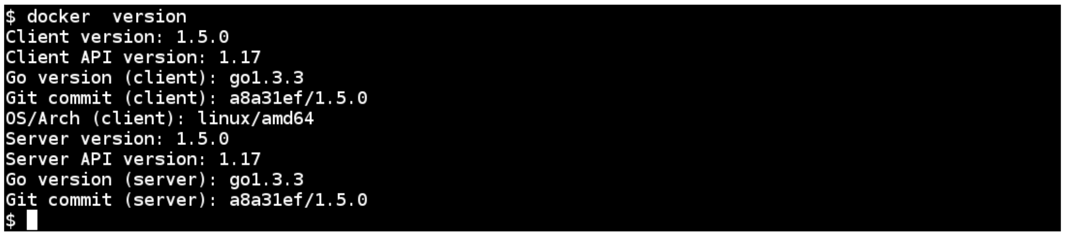
In this chapter, we will cover the following recipes:

- Listing/searching for an image
- Pulling an image
- Listing images
- Starting a container
- Listing containers
- Stopping a container
- Looking at the logs of containers
- Deleting a container
- Setting the restart policy on a container
- Getting privileged access inside a container
- Exposing a port while starting a container
- Accessing the host device inside the container
- Injecting a new process to a running container
- Returning low-level information about a container
- Labeling and filtering containers

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 do with containers such as starting, stopping, listing, deleting, and so on. This will help us to 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: 1.5.0
Client API version: 1.17
Go version (client): go1.3.3
Git commit (client): a8a31ef/1.5.0
OS/Arch (client): linux/amd64
Server version: 1.5.0
Server API version: 1.17
Go version (server): go1.3.3
Git commit (server): a8a31ef/1.5.0
$ 
```

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

I am using Fedora 20/21 as my primary environment to run the recipes. They should also work with the other environment.

Listing/searching for an image

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

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

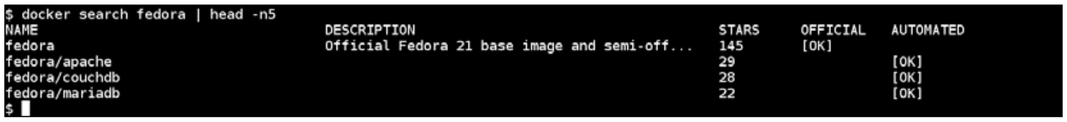
How to do it...

1. To search an image on a Docker registry, run the following command:

   ```
   docker search TERM
   ```

The following is an example to search a Fedora image:

```
$ docker search fedora |  head -n5
```



```
$ docker search fedora | head -n5
NAME                      DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
fedora                    Official Fedora 21 base image and semi-off...   145       [OK]
fedora/apache                                                             29                   [OK]
fedora/couchdb                                                            28                   [OK]
fedora/mariadb                                                            22                   [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` signifies 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 to tell whether an image is built automatically with push in GitHub or Bitbucket repositories. More details about `AUTOMATED` can be found in the next chapter.

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

How it works...

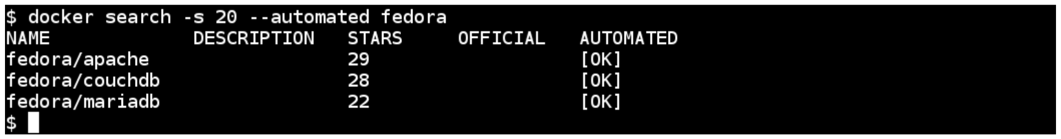
Docker searches for images on the Docker public registry, which has a repository for images at `https://registry.hub.docker.com/`.

We can configure our private index as well, which it can search for.

There's more...

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

  ```
  $ docker search -s 20 --automated fedora
  ```



```
$ docker search -s 20 --automated fedora
NAME              DESCRIPTION   STARS     OFFICIAL   AUTOMATED
fedora/apache                   29                   [OK]
fedora/couchdb                  28                   [OK]
fedora/mariadb                  22                   [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 to 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/reference/commandline/cli/#insecure-registries`.

- The Docker package on RHEL 7 and Fedora provides options to add and block the registry with the `--add-registry` and `--block-registry` options respectively, to have better control over the image search path. For more details, look at the following links:
    - `http://rhelblog.redhat.com/2015/04/15/understanding-the-changes-to-docker-search-and-docker-pull-in-red-hat-enterprise-linux-7-1/`
    - `https://github.com/docker/docker/pull/10411`

See also

- For help with the Docker search, run the following command:

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

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#search`

Pulling an image

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

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

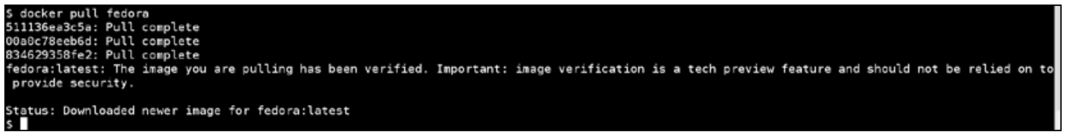
How to do it...

1. To pull an image on the Docker registry, run the following command:

   ```
   docker pull NAME[:TAG]
   ```

The following is an example to pull the Fedora image:

```
$ docker pull fedora
```



```
$ docker pull fedora
511136ea3c5a: Pull complete
00a0c78eeb6d: Pull complete
834629358fe2: Pull complete
fedora:latest: The image you are pulling has been verified. Important: image verification is a tech preview feature and should not be relied on to
 provide security.

Status: Downloaded newer image for fedora:latest
$ 
```

How it works...

The `pull` command downloads all layers from the Docker registry, which 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 `centos5`, `centos6`, and so on. For example, to pull an image with the specific tag, run the following command:

  ```
  $ docker 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 pull --all-tags centos
  ```

- With 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 a specific image, rather than tags. To pull an image with a specific digest, we can consider the following syntax:

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

  Here is an example of a command:

  ```
  $ docker pull debian@sha256:cbbf2f9a99b47fc460d422812b6a5adff7dfee
  951d8fa2e4a98caa0382cfbdbf
  ```

  Digest is supported only with the Docker registry v2.

- Once an image gets pulled, it resides on 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 of Docker `pull`:

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

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#pull`

Listing images

We can list the images available on the system running the Docker daemon. These images might have been pulled from the registry, imported through the `docker` command, or created through Docker files.

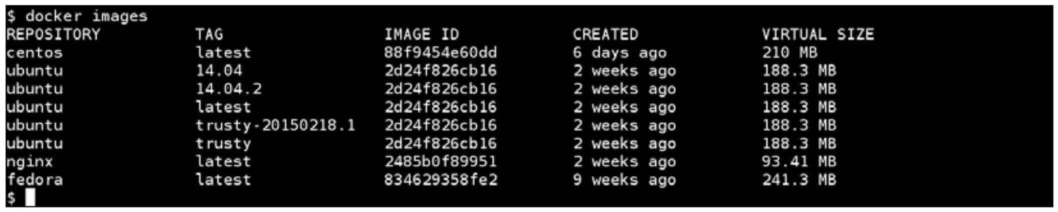
Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it...

1. Run the following command to list the images:

   `$ docker images`



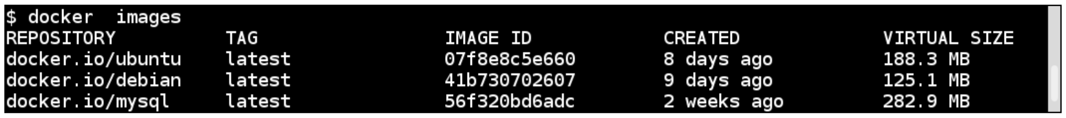
```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
centos              latest              88f9454e60dd        6 days ago          210 MB
ubuntu              14.04               2d24f826cb16        2 weeks ago         188.3 MB
ubuntu              14.04.2             2d24f826cb16        2 weeks ago         188.3 MB
ubuntu              latest              2d24f826cb16        2 weeks ago         188.3 MB
ubuntu              trusty-20150218.1   2d24f826cb16        2 weeks ago         188.3 MB
ubuntu              trusty              2d24f826cb16        2 weeks ago         188.3 MB
nginx               latest              2485b0f89951        2 weeks ago         93.41 MB
fedora              latest              834629358fe2        9 weeks ago         241.3 MB
$ 
```

How it works...

The Docker client talks to the Docker server and gets the list of images at the server end.

There's more...

- All the images with the same name but different tags get 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 `2d24f826cb16`.
- You might see a different output for `REPOSITORY`, as shown in the following screenshot, with the latest Docker packages.



```
$ docker  images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
docker.io/ubuntu    latest              07f8e8c5e660        8 days ago          188.3 MB
docker.io/debian    latest              41b730702607        9 days ago          125.1 MB
docker.io/mysql     latest              56f320bd6adc        2 weeks ago         282.9 MB
```

This is because the image listing prints the Docker registry hostname as well. As shown in the preceding screenshot, `docker.io` is the registry hostname.

See also

- Look at the `help` option of `docker images`:

  ```
  $ docker images --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#images`

Starting a container

Once we have images, we can use them to start the containers. In this recipe, we will start a container with the `fedora:latest` image and see what all things happen behind the scene.

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it…

1. The syntax used to start a container is as follows:

   ```
   docker run [ OPTIONS ]  IMAGE[:TAG]  [COMMAND]  [ARG...]
   ```

Here is an example of a command:

```
$ docker run -i -t --name=f21 fedora /bin/bash
```

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

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

So, with the preceding command, we start a container from the `fedora:latest` image, attach `pseudo-tty`, name it `f21`, 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 get downloaded from the registry first and then run. Docker will run the `search` and `pull` commands before running the `run` command.

How it works...

Under the hood, Docker:

- Will merge all the layers that make that image using UnionFS.
- Allocates a unique ID to a container, which is referred to as Container ID.
- Allocates a filesystem and mounts a read/write layer for the container. Any changes on this layer will be temporary and will be discarded if they are not committed.
- Allocates a network/bridge interface.
- Assigns an IP address to the container.
- Executes 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.
- 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 create -t -i fedora bash)
  $ docker 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 run -d -i -t fedora /bin/bash
  0df95cc49e258b74be713c31d5a28b9d590906ed9d6e1a2dc756 72aa48f28c4f
  ```

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

  ```
  $ ID='docker run -d -t -i fedora /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 noninteractive process and run it in the background to make a daemonized container like this:

```
$ docker run -d  fedora /bin/bash -c  "while [ 1 ]; do echo
hello docker ; sleep 1; done"
```

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

  ```
  $ docker run --rm fedora 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 the `read-only` mode:

  ```
  $ docker run --read-only -d -i -t fedora /bin/bash
  ```

  Remember that this option just makes sure that we cannot modify anything on the root filesystem, but we are writing on volumes, which we'll cover later in the book. This option is very useful when we don't want users to accidentally write content inside the container, which gets lost if the container is not committed or copied out on non-ephemeral storage such as volumes.

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

A container can be referred in three ways: by name, by container ID (0df95cc49e258b74be713c31d5a28b9d590906ed9d6e1a2dc75672 aa48f28c4f), and by short container ID (0df95cc49e25)

See also

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

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

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#run`
- The Docker 1.3 release announcement `http://blog.docker.com/2014/10/docker-1-3-signed-images-process-injection-security-options-mac-shared-directories/`

Listing containers

We can list both running and stopped containers.

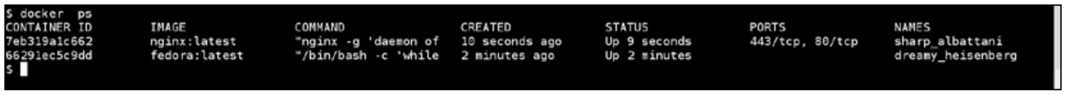
Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client. You will also need a few running and/or stopped containers.

How to do it...

1. To list the containers, run the following command:

```
docker ps [ OPTIONS ]
```



```
$ docker  ps
CONTAINER ID        IMAGE               COMMAND                CREATED             STATUS              PORTS               NAMES
7eb319a1c662        nginx:latest        "nginx -g 'daemon of   10 seconds ago      Up 9 seconds        443/tcp, 80/tcp     sharp_albattani
66291ec5c9dd        fedora:latest       "/bin/bash -c 'while   2 minutes ago       Up 2 minutes                            dreamy_heisenberg
$ 
```

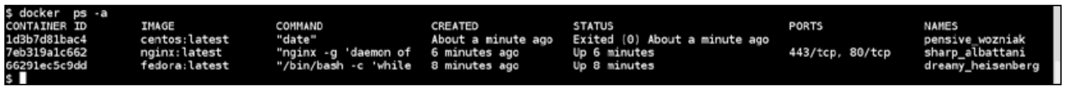
How it works...

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

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

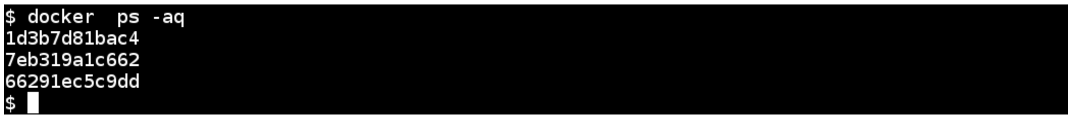
There's more...

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



```
$ docker  ps -a
CONTAINER ID        IMAGE               COMMAND                CREATED              STATUS                          PORTS               NAMES
1d3b7d81bac4        centos:latest       "date"                 About a minute ago   Exited (0) About a minute ago                       pensive_wozniak
7eb319a1c662        nginx:latest        "nginx -g 'daemon of   6 minutes ago        Up 6 minutes                    443/tcp, 80/tcp     sharp_albattani
66291ec5c9dd        fedora:latest       "/bin/bash -c 'while   8 minutes ago        Up 8 minutes                                        dreamy_heisenberg
$ 
```

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



```
$ docker  ps -aq
1d3b7d81bac4
7eb319a1c662
66291ec5c9dd
$ 
```

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

  `$ docker ps -l`

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

See also

Look at the `man` page of `docker ps` to see more options:

- Look at the `help` option of `docker ps`:

  `$ docker ps --help`

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#ps`

Looking at the logs of containers

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

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client. You will also need a running container, which emits logs/output on `STDOUT`.

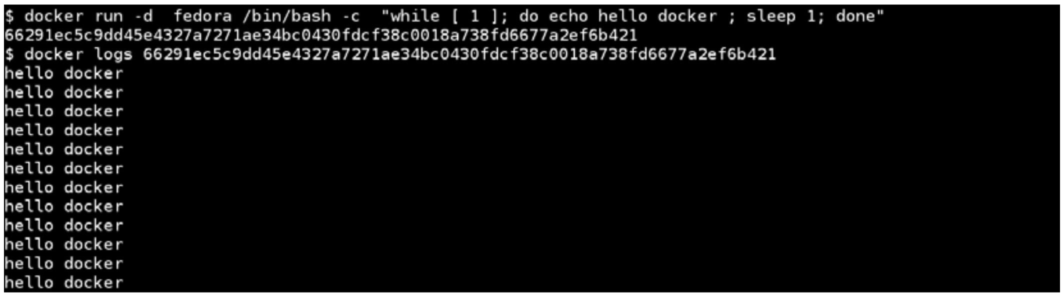
How to do it...

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

   ```
   docker logs [-f|--follow[=false]][-t|--timestamps[=false]]
   CONTAINER
   ```

2. Let's take the example from the earlier section of running a daemonized container and look at the logs:

   ```
   $ docker run -d  fedora /bin/bash -c  "while [ 1 ]; do echo
   hello docker ; sleep 1; done"
   ```



```
$ docker run -d  fedora /bin/bash -c  "while [ 1 ]; do echo hello docker ; sleep 1; done"
66291ec5c9dd45e4327a7271ae34bc0430fdcf38c0018a738fd6677a2ef6b421
$ docker logs 66291ec5c9dd45e4327a7271ae34bc0430fdcf38c0018a738fd6677a2ef6b421
hello docker
hello docker
hello docker
hello docker
hello docker
hello docker
hello docker
hello docker
hello docker
hello docker
hello docker
hello docker
```

How it works...

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

There's more...

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

See also

- Look at `help` option of `docker logs`:

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

- Documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#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

Make sure the Docker daemon is running on the host and you can connect through the Docker client. You will also need one or more running containers.

How to do it...

1. To stop the container, run the following command:

   ```
   docker stop [-t|--time[=10]] CONTAINER [CONTAINER...]
   ```

2. If you already have a running container, then you can go ahead and stop it; if not, we can create one and then stop it as follows:

   ```
   $ ID='docker run -d -i fedora /bin/bash'
   $ docker stop $ID
   ```

How it works...

This will save the state of the container and stop it. It can 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 the running containers run the following command:

  ```
  $ docker stop 'docker ps -q'
  ```

See also

- Look at `help` option of `docker stop`:

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

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#stop`

Deleting a container

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

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client. You will also need some containers in a stopped or running state to delete them.

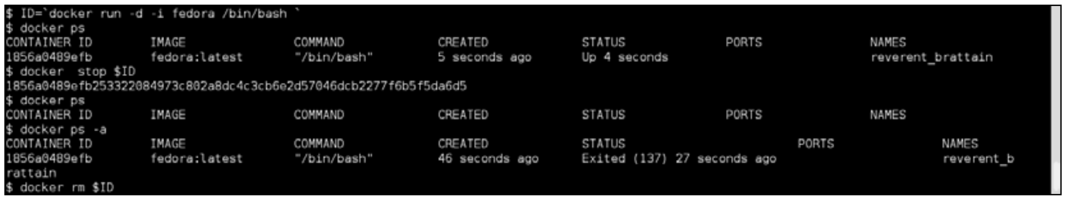
How to do it...

1. Use the following command:

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

2. Let's first start a container, stop it, and then delete it using the following commands:

```
$ ID='docker run -d -i fedora /bin/bash '
$ docker stop $ID
$ docker rm $ID
```



```
$ ID=`docker run -d -i fedora /bin/bash `
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
1856a0489efb        fedora:latest       "/bin/bash"         5 seconds ago       Up 4 seconds                            reverent_brattain
$ docker  stop $ID
1856a0489efb253322084973c802a8dc4c3cb6e2d57046dcb2277f6b5f5da6d5
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                        PORTS               NAMES
1856a0489efb        fedora:latest       "/bin/bash"         46 seconds ago      Exited (137) 27 seconds ago                       reverent_b
rattain
$ docker rm $ID
```

As we can see from the preceding screenshot, the container did not show up, which just entered the `docker ps` command after stopping it. We had to provide the `-a` option to list it. After the container is stopped, we can delete it.

There's more...

- To forcefully delete a container without an intermediate stop, use the `-f` option.
- To delete all the containers, we first need to stop all the running containers and then remove them. Be careful before running the commands as these will delete both the running and the stopped containers:

```
$ docker stop 'docker ps -q'
$ docker rm 'docker ps -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 rm`

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

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#rm`

Setting the restart policy on a container

Before Docker 1.2, there used to be an option to restart the container. With the release of Docker 1.2, it has been added with the `run` command with flags to specify the restart policy. With this policy, we can configure containers to start at boot time. This option is also very useful when a container dies accidentally.

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it...

You can set the restart policy using the following syntax:

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

Here is an example of a command:

```
$ docker run --restart=always -d -i -t fedora /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 nonzero exit code
- `always`: This always restarts the container without worrying about the return code

There's more...

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

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

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

See also

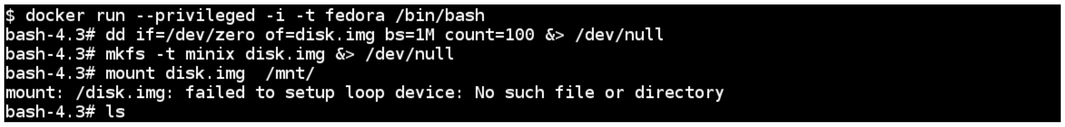
- Look at the `help` option of `docker run`:

  `$ docker run --help`

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#run`.
- If a restart does not suit your requirements, then use `systemd` (`http://freedesktop.org/wiki/Software/systemd/`) for solutions to automatically restart the container on failure. For more information, visit `https://docs.docker.com/articles/host_integration/`.

Getting privileged access inside a container

Linux divides the privileges traditionally associated with superuser 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 give more capabilities to perform operations normally done by root. For example, let's try to create a loopback device while mounting a disk image.



```
$ docker run --privileged -i -t fedora /bin/bash
bash-4.3# dd if=/dev/zero of=disk.img bs=1M count=100 &> /dev/null
bash-4.3# mkfs -t minix disk.img &> /dev/null
bash-4.3# mount disk.img  /mnt/
mount: /disk.img: failed to setup loop device: No such file or directory
bash-4.3# ls
```

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

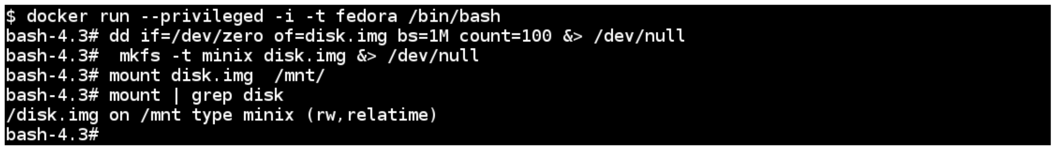
How to do it…

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

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

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

   ```
   $ docker run  --privileged  -i -t fedora /bin/bash
   ```



```
$ docker run --privileged -i -t fedora /bin/bash
bash-4.3# dd if=/dev/zero of=disk.img bs=1M count=100 &> /dev/null
bash-4.3#  mkfs -t minix disk.img &> /dev/null
bash-4.3# mount disk.img  /mnt/
bash-4.3# mount | grep disk
/disk.img on /mnt type minix (rw,relatime)
bash-4.3#
```

How it works...

By providing 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 new, two new flags `--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 run --cap-drop=CHOWN [ OPTIONS ]  IMAGE[:TAG]  [COMMAND]
[ARG...]
```

Look at *Chapter 9*, *Docker Security*, for more details.

See also

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

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

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#run`
- The Docker 1.2 release announcement `http://blog.docker.com/2014/08/announcing-docker-1-2-0/`

Exposing a port while starting a container

There are a number of ways by which ports on the container can be exposed. One of them is through the `run` command, which we will cover in this chapter. The other ways are through the Docker file and the `--link` command. We will explore them in the other chapters.

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it...

1. The syntax to expose a port is as follows:

   ```
   $ docker run --expose=PORT [ OPTIONS ]  IMAGE[:TAG]  [COMMAND]
   [ARG...]
   ```

For example, to expose port 22 while starting a container, run the following command:

```
$ docker run --expose=22 -i -t fedora /bin/bash
```

There's more...

There are multiple ways to expose the ports for a container. For now, we will see how we can expose the port while starting the container. We'll look other options to expose the ports in later chapters.

See also

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

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

- Documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#run`

Accessing the host device inside the container

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

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client. You will also need a device to pass to the container.

How to do it...

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

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

Here is an example of a command:

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

How it works...

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

See also

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

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

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#run`

Injecting a new process to a running container

While doing development and debugging, we might want to look inside the already running container. There are a few utilities, such as `nsenter` (`https://github.com/jpetazzo/nsenter`), that allow us to enter into the namespace of the container to inspect it. 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 the Docker daemon is running on the host and you can connect through the Docker client. You might also need a running container to inject a process in.

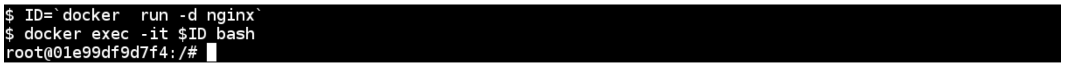
How to do it...

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

```
 $ docker exec [-d|--detach[=false]] [--help] [-i|--
interactive[=false]] [-t|--tty[=false]] CONTAINER COMMAND
[ARG...]
```

2. Let's start an `nginx` container and then inject `bash` into that:

```
$ ID='docker run -d nginx'
$ docker run -it $ID bash
```



```
$ ID=`docker  run -d nginx`
$ docker exec -it $ID bash
root@01e99df9d7f4:/#
```

How it works...

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

See also

- Look at `help` option of Docker inspect:

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

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#exec`

Returning low-level information about a container

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

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it...

1. To inspect a container/image, run the following command:

   ```
   $ docker inspect [-f|--format="" CONTAINER|IMAGE
   [CONTAINER|IMAGE...]
   ```

2. We'll start a container and then inspect it:

   ```
   $ ID='docker run -d -i fedora /bin/bash'
   $ docker inspect $ID
   [{
       "Args": [],
       "Config": {
           "AttachStderr": false,
           "AttachStdin": false,
           "AttachStdout": false,
           "Cmd": [
               "/bin/bash"
           ],
       .........
       .........
   }]
   ```

How it works...

Docker will look into the metadata and configuration for the given image or container and present it.

There's more...

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

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

See also

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

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

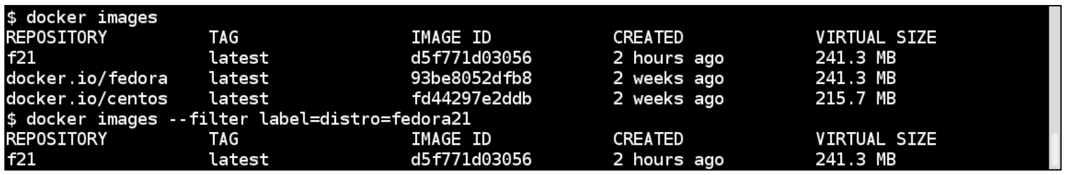
- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#inspect`

Labeling and filtering containers

With Docker 1.6, a feature has been added to label containers and images, through which we can attach arbitrary key-value metadata to them. You can think of them as environment variables, which are not available to running applications inside containers but they are available to programs (Docker CLI) that are managing images and containers. Labels attached to images also get applied to containers started via them. We can also attach labels to containers while starting them.

Docker also provides filters to containers, images, and events (`https://docs.docker.com/reference/commandline/cli/#filtering`), which we can use in conjunction with labels to narrow down our searches.

For this recipe, let's assume that we have an image with the label, `distro=fedora21`. In the next chapter, we will see how to assign a label to an image.



```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
f21                 latest              d5f771d03056        2 hours ago         241.3 MB
docker.io/fedora    latest              93be8052dfb8        2 weeks ago         241.3 MB
docker.io/centos    latest              fd44297e2ddb        2 weeks ago         215.7 MB
$ docker images --filter label=distro=fedora21
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
f21                 latest              d5f771d03056        2 hours ago         241.3 MB
```

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

Getting ready

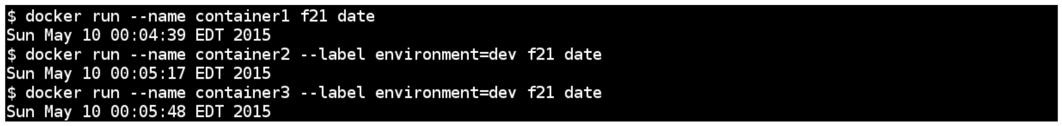
Make sure that the Docker daemon 1.6 and above is running on the host and you can connect through the Docker client.

How to do it...

1. To start the container with the `--label/-l` option, run the following command:

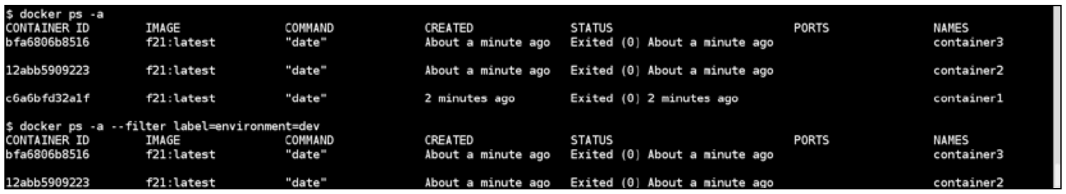
   ```
   $ docker run --label environment=dev f21 date
   ```

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



```
$ docker run --name container1 f21 date
Sun May 10 00:04:39 EDT 2015
$ docker run --name container2 --label environment=dev f21 date
Sun May 10 00:05:17 EDT 2015
$ docker run --name container3 --label environment=dev f21 date
Sun May 10 00:05:48 EDT 2015
```

If we list all the containers without a label, we will see all the containers, but if we use label, then we get only containers, which matches the label.



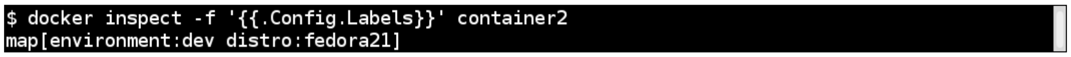
```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                         PORTS               NAMES
bfa6806b8516        f21:latest          "date"              About a minute ago  Exited (0) About a minute ago                      container3

12abb5909223        f21:latest          "date"              About a minute ago  Exited (0) About a minute ago                      container2

c6a6bfd32a1f        f21:latest          "date"              2 minutes ago       Exited (0) 2 minutes ago                           container1

$ docker ps -a --filter label=environment=dev
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                         PORTS               NAMES
bfa6806b8516        f21:latest          "date"              About a minute ago  Exited (0) About a minute ago                      container3

12abb5909223        f21:latest          "date"              About a minute ago  Exited (0) About a minute ago                      container2
```

How it works...

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

There's more...

- We can list all the labels attached to a container through the `inspect` command, which we saw in an earlier recipe. As we can see, the `inspect` command returns both the image and the container labels.



```
$ docker inspect -f '{{.Config.Labels}}' container2
map[environment:dev distro:fedora21]
```

- You can apply labels from a file (with the `--from-file` option) that has a list of labels, separated by a new EOL.
- These labels are different from the Kubernetes label, which we will see in *Chapter 8*, *Docker Orchestration and Hosting Platforms*.

See also

- The documentation on the Docker website `https://docs.docker.com/reference/builder/#label`
- `http://rancher.com/docker-labels/`

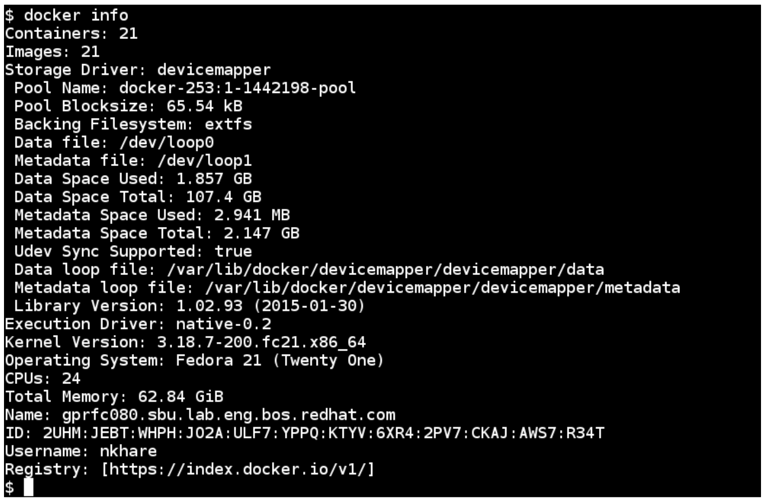
3
Working with Docker Images

In this chapter, we will cover the following recipes:

- Creating an account with Docker Hub
- Creating an image from the container
- Publishing an image to the registry
- Looking at the history of an image
- Deleting an image
- Exporting an image
- Importing an image
- Building images using Dockerfiles
- Building an Apache image – a Dockerfile example
- Accessing Firefox from a container – a Dockerfile example
- Building a WordPress image – a Dockerfile example
- Setting up a private index/registry
- Automated Builds – with GitHub and Bitbucket
- Creating the base image – using supermin
- Creating the base image – using Debootstrap
- Visualizing dependencies between layers

Introduction

In this chapter, we will focus on operations relating to images. As we know, images are required to run containers. You can either use existing images or create new custom images. You will need to create custom images to suit your development and deployment environment. Once you create an image, you can share it through the public or private registry. Before we explore more about Docker images, let's look at the output of the `docker info` command:



```
$ docker info
Containers: 21
Images: 21
Storage Driver: devicemapper
 Pool Name: docker-253:1-1442198-pool
 Pool Blocksize: 65.54 kB
 Backing Filesystem: extfs
 Data file: /dev/loop0
 Metadata file: /dev/loop1
 Data Space Used: 1.857 GB
 Data Space Total: 107.4 GB
 Metadata Space Used: 2.941 MB
 Metadata Space Total: 2.147 GB
 Udev Sync Supported: true
 Data loop file: /var/lib/docker/devicemapper/devicemapper/data
 Metadata loop file: /var/lib/docker/devicemapper/devicemapper/metadata
 Library Version: 1.02.93 (2015-01-30)
Execution Driver: native-0.2
Kernel Version: 3.18.7-200.fc21.x86_64
Operating System: Fedora 21 (Twenty One)
CPUs: 24
Total Memory: 62.84 GiB
Name: gprfc080.sbu.lab.eng.bos.redhat.com
ID: 2UHM:JEBT:WHPH:JO2A:ULF7:YPPQ:KTYV:6XR4:2PV7:CKAJ:AWS7:R34T
Username: nkhare
Registry: [https://index.docker.io/v1/]
$ 
```

The preceding command gives the current system-wide info as follows:

- It has 21 containers and 21 images.
- The current storage driver, `devicemapper`, and its related information, such as thin pool name, data, metadata file, and so on. Other types of storage drivers are aufs, btrfs, overlayfs, vfs, and so on. Devicemapper, btrfs, and overlayfs have native support in the Linux kernel. AUFS support needs a patched kernel. We talked about the Union filesystem in *Chapter 1*, *Introduction and Installation*.
- To leverage the kernel features that enable containerization, the Docker daemon has to talk to the Linux kernel. This is done through the execution driver. `libconatiner` or `native` is one of that type. The others are `libvirt`, `lxc`, and so on, which we saw in *Chapter 1*, *Introduction and Installation*.
- The kernel version on the host operating system.
- The user account that is registered on the registry mentioned in the next section to pull/push images.

I am using Fedora 20/21 as my primary environment to run the recipes. They should also work with other environments.

Creating an account with Docker Hub

Docker Hub is like GitHub for images. It is a public registry on which you can host images both public and private, share them and collaborate with others. It has integration with GitHub, Bitbucket, and can trigger automated builds.

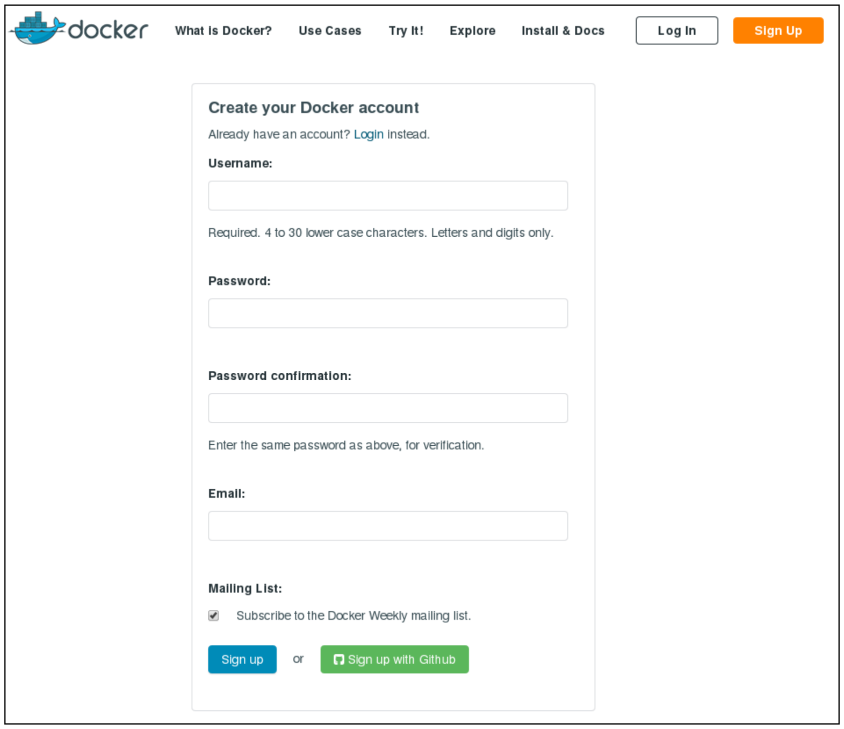
As of now, the creation of an account on Docker Hub is free. A repository can hold different versions of an image. You can create any number of public repositories for your images. By default, you will have one private repository, which will not be accessible to the public. You can buy more private repositories. You can create an account either through a web browser or from the command line.

Getting ready

To sign up from the command line, you will need to have Docker installed on your system.

How to do it...

1. To create an account through a web browser on Docker Hub, visit `https://hub.docker.com/account/signup/` and create an account:



Create your Docker account

Already have an account? Login instead.

Username:

Required. 4 to 30 lower case characters. Letters and digits only.

Password:

Password confirmation:

Enter the same password as above, for verification.

Email:

Mailing List:

☑ Subscribe to the Docker Weekly mailing list.

2. To create an account using the command line, run the following command and submit the required details:

   ```
   $ docker login
   ```

How it works...

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

See also

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

Creating an image from the container

There are a couple of ways to create images, one is by manually committing layers and the other way is through Dockerfiles. In this recipe, we'll see the former 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 see how to save that layer and make a new image from the running or stopped container using the `docker commit` command.

Getting ready

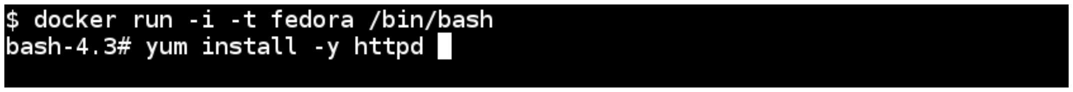
To get a Docker image, start a container with it.

How to do it...

1. To do the commit, run the following command:

   ```
   docker commit -a|--author[=""] -m|--message[=""] CONTAINER
   [REPOSITORY[:TAG]]
   ```

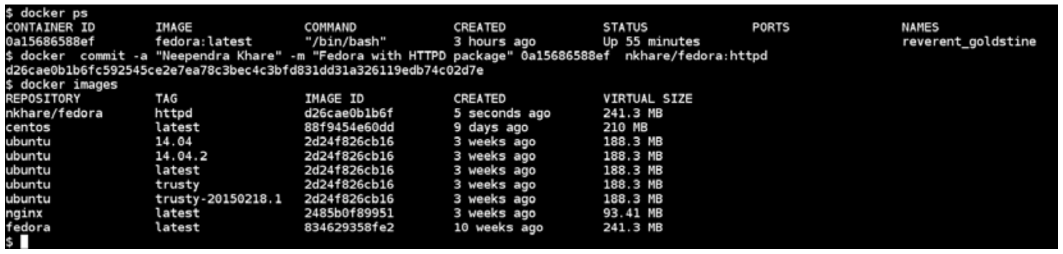
2. Let's start a container and create/modify some files using the `install httpd` package:



```
$ docker run -i -t fedora /bin/bash
bash-4.3# yum install -y httpd
```

3. Then, open a new terminal and create a new image by doing the commit:

   ```
   $ docker commit -a "Neependra Khare" -m "Fedora with HTTPD
   package" 0a15686588ef nkhare/fedora:httpd
   ```



```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
0a15686588ef        fedora:latest       "/bin/bash"         3 hours ago         Up 55 minutes                           reverent_goldstine
$ docker  commit -a "Neependra Khare" -m "Fedora with HTTPD package" 0a15686588ef  nkhare/fedora:httpd
d26cae0b1b6fc592545ce2e7ea78c3bec4c3bfd831dd31a326119edb74c02d7e
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
nkhare/fedora       httpd               d26cae0b1b6f        5 seconds ago       241.3 MB
centos              latest              88f9454e60dd        9 days ago          210 MB
ubuntu              14.04               2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              14.04.2             2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              latest              2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              trusty              2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              trusty-20150218.1   2d24f826cb16        3 weeks ago         188.3 MB
nginx               latest              2485b0f89951        3 weeks ago         93.41 MB
fedora              latest              834629358fe2        10 weeks ago        241.3 MB
$ 
```

As you can see, the new image is now being committed to the local repository with `nkhare/fedora` as a name and `httpd` as a tag.

How it works...

In *Chapter 1*, *Introduction and Installation*, we saw that while starting a container, a read/write filesystem layer will be created on top of the existing image layers from which the container started, and with the installation of a package, some files would have been added/modified in that layer. All of those changes are currently in the ephemeral read/write filesystem layer, which is assigned to the container. If we stop and delete the container, then all of the earlier mentioned modifications would be lost.

Using commit, we create a new layer with the changes that have occurred since the container started, which get saved in the backend storage driver.

There's more...

- To look for files, which are changed since the container started:

  ```
  $ docker diff CONTAINER
  ```

  In our case, we will see something like the following code:

  ```
  $ docker diff 0a15686588ef
  .....
  C /var/log
  A /var/log/httpd
  C /var/log/lastlog
  .....
  ```

We can see 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

- Look at the `help` option of `docker commit`:

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

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#commit`

Publishing an image to the registry

Let's say you have created an image that suits the development environment in your organization. You can either share it using tar ball, which we'll see later in this chapter, or put in a central location from where the user can pull it. This central location can be either a public or a private registry. In this recipe, we'll see how to push the image to the registry using the `docker push` command. Later in this chapter, we'll cover how to set up the private registry.

Getting ready

You will need a valid account on Docker Hub to push images/repositories.

A local registry must be set up if you are pushing images/repositories locally.

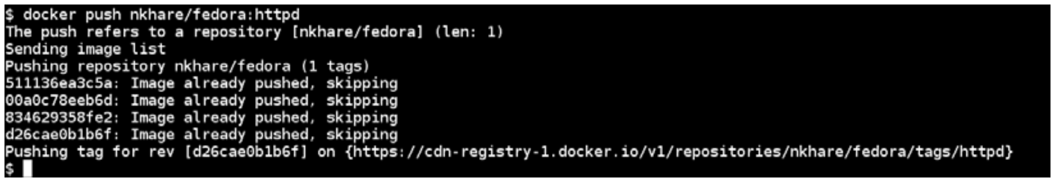
How to do it...

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

By default, the preceding command will use the username and registry shown in the `docker info` command to push the images. As shown in the preceding screenshot, the command will use `nkhare` as the username and `https://index.docker.io/v1/` as the registry.

To push the image that we created in the previous section, run the following command:

```
$ docker push nkhare/fedora:httpd
```



```
$ docker push nkhare/fedora:httpd
The push refers to a repository [nkhare/fedora] (len: 1)
Sending image list
Pushing repository nkhare/fedora (1 tags)
511136ea3c5a: Image already pushed, skipping
00a0c78eeb6d: Image already pushed, skipping
834629358fe2: Image already pushed, skipping
d26cae0b1b6f: Image already pushed, skipping
Pushing tag for rev [d26cae0b1b6f] on {https://cdn-registry-1.docker.io/v1/repositories/nkhare/fedora/tags/httpd}
$
```

Let's say you want to push the image to the local registry, which is hosted on a host called `local-registry`. 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.

```
$ docker tag [-f|--force[=false] IMAGE
[REGISTRYHOST/][USERNAME/]NAME[:TAG]
$ docker push [REGISTRYHOST/][USERNAME/]NAME[:TAG]
```

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

```
$ docker tag nkhare/fedora:httpd
shadowfax.example.com:5000/nkhare/fedora:httpd
```

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

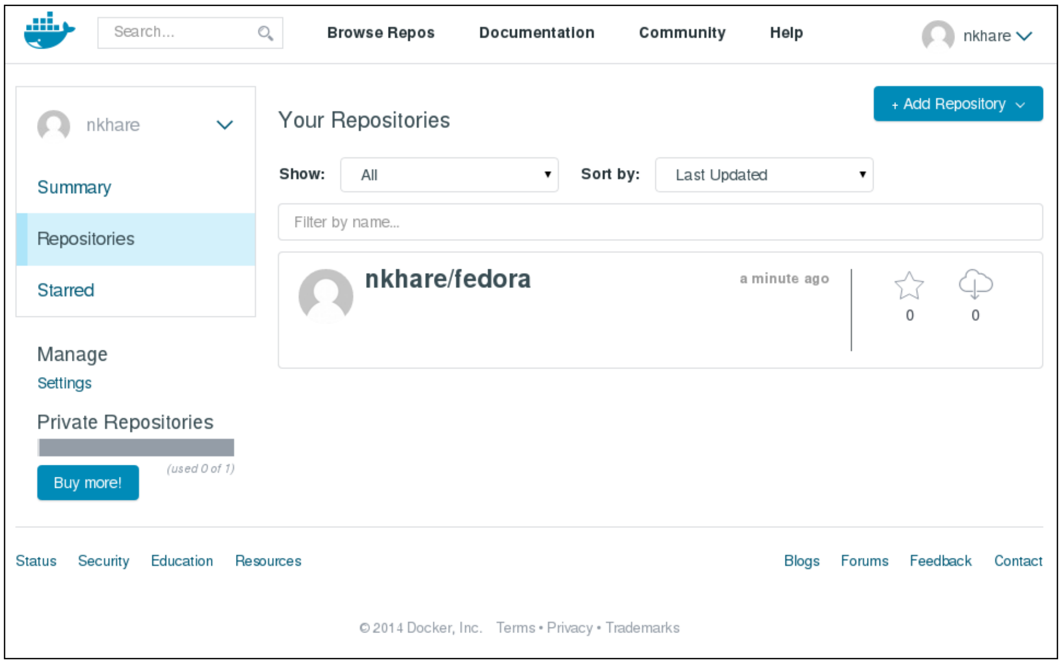
```
$ docker push shadowfax.example.com:5000/nkhare/fedora:httpd
```

How it works...

It will first list down all the intermediate layers that are required to make that specific image. It will then check to see, out of those layers, how many are already present inside the registry. At last, it will copy all the layers, which are not present in the registry with the metadata required to build the image.

There's more...

As we pushed our image to the public registry, we can log in to Docker Hub and look for the image:



See also

- Look at the `help` option of `docker push`:

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

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#push`

Looking at the history of an image

It is handy to know how the image that we are using has been created. The `docker history` command helps us find all the intermediate layers.

Getting ready

Pull or import any Docker image.

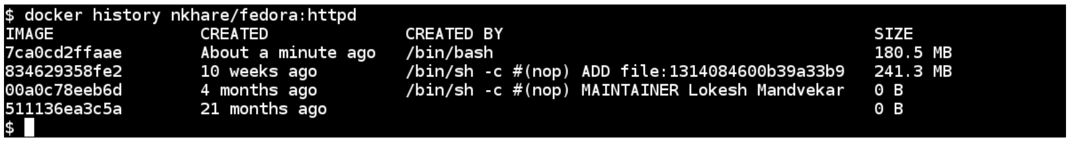
How to do it...

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

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

   Here's an example using the preceding syntax:

   ```
   $ docker history nkhare/fedora:httpd
   ```



```
$ docker history nkhare/fedora:httpd
IMAGE               CREATED             CREATED BY                                      SIZE
7ca0cd2ffaae        About a minute ago  /bin/bash                                       180.5 MB
834629358fe2        10 weeks ago        /bin/sh -c #(nop) ADD file:1314084600b39a33b9   241.3 MB
00a0c78eeb6d        4 months ago        /bin/sh -c #(nop) MAINTAINER Lokesh Mandvekar   0 B
511136ea3c5a        21 months ago                                                       0 B
$ 
```

How it works...

From the metadata of an image, Docker can know how an image is being created. With the `history` command, it will look at the metadata recursively to get to the origin.

There's more...

Look at the commit message of a layer that got committed:

```
$ docker inspect --format='{{.Comment}}' nkhare/fedora:httpd
Fedora with HTTPD package
```

Currently, there is no direct way to look at the commit message for each layer using one single command, but we can use the `inspect` command, which we saw earlier, for each layer.

See also

- Look at the `help` option of `docker history`:

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

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#history`

Deleting an image

To remove the image from the host, we can use the `docker rmi` command. However, this does not remove images from the registry.

Getting ready

Make sure one or more Docker images are locally available.

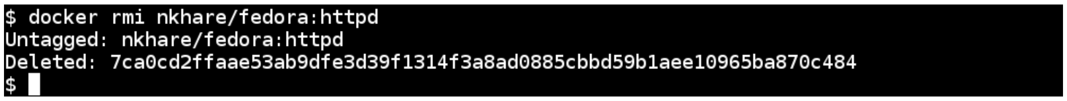
How to do it...

1. To remove the image, consider the following syntax:

   ```
   $ docker rmi [ OPTIONS ] IMAGE [IMAGE...]
   ```

   In our case, here's an example using the preceding syntax:

   ```
   $ docker rmi nkhare/fedora:httpd
   ```



```
$ docker rmi nkhare/fedora:httpd
Untagged: nkhare/fedora:httpd
Deleted: 7ca0cd2ffaae53ab9dfe3d39f1314f3a8ad0885cbbd59b1aee10965ba870c484
$ 
```

There's more...

If you want to remove all containers and images, then do following; however, be sure about what you are doing, as this is very destructive:

- To stop all containers, use the following command:

  ```
  $ docker stop 'docker ps -q'
  ```

- To delete all containers, use the following command:

  ```
  $ docker rm 'docker ps -a -q'
  ```

- To delete all images, use the following command:

  ```
  $ docker rmi 'docker images -q'
  ```

See also

- Look at the `help` option of `docker rmi`:

  ```
  $ docker rmi --help
  ```

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#rmi`

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 images through tarballs, which later can be imported on another system. In this recipe, we will see how to do that using the `docker save` command.

Getting ready

Pull or import one or more Docker images on the Docker host.

How to do it...

1. Use the following syntax to save the image in the tar file:

   ```
   $ docker save [-o|--output=""] IMAGE [:TAG]
   ```

   For example, to create a tar archive for Fedora, run the following command:

   ```
   $ docker save --output=fedora.tar fedora
   ```

If the tag name is specified with the image name we want to export, such as `fedora:latest`, then only the layers related to that tag will get exported.

There's more...

If `--output` or `-o` is not used, then the output will be streamed to `STDOUT`:

```
$ docker save fedora:latest > fedora-latest.tar
```

Similarly, the contents of the container's filesystem can be exported using the following command:

```
$ docker export CONTAINER  > containerXYZ.tar
```

See also

- Look at the `help` option of `docker save` and `docker export`:

  ```
  $ docker save -help
  $ docker export --help
  ```

- The documentation on the Docker website:
  - `https://docs.docker.com/reference/commandline/cli/#save`
  - `https://docs.docker.com/reference/commandline/cli/#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 the earlier recipe. Using the `docker import` command, we import an exported image.

Getting ready

You need an accessible exported Docker image.

How to do it…

1. To import an image, we can use following syntax:

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

   Here's an example using the preceding syntax:

   ```
   $ cat fedora-latest.tar | docker import - fedora:latest
   ```

   Alternatively, you can consider the following example:

   ```
   $ docker import http://example.com/example.tar example/image
   ```

The preceding example will first create an empty filesystem and then import the contents.

See also

- Look at the `help` option of `docker import`:

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

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#import`

Building images using Dockerfiles

Dockerfiles help us in automating image creation and getting precisely the same image every time we want it. The Docker builder reads instructions from a text file (a Dockerfile) and executes them one after the other in order. It can be compared as Vagrant files, which allows you to configure VMs in a predictable manner.

Getting ready

A Dockerfile with build instructions.

- Create an empty directory:

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

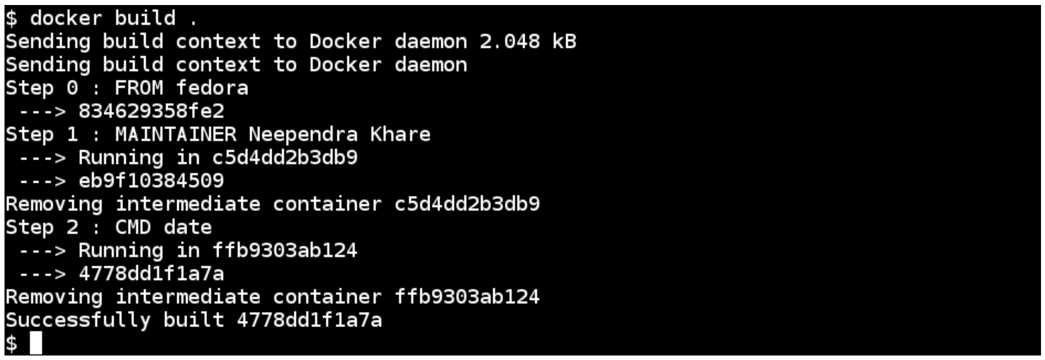
- Create a file named `Dockerfile` with the following content:

  ```
  $ cat Dockerfile
  # Pick up the base image
  FROM fedora
  # Add author name
  MAINTAINER Neependra Khare
  # Add the command to run at the start of container
  CMD date
  ```

How to do it...

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

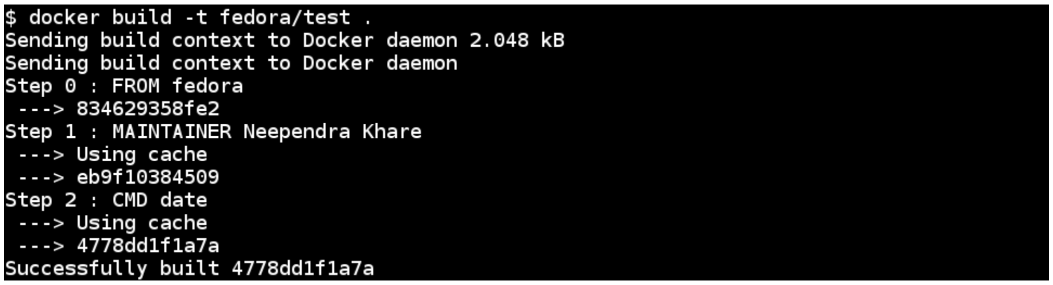
   ```
   $ docker build .
   ```



```
$ docker build .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM fedora
 ---> 834629358fe2
Step 1 : MAINTAINER Neependra Khare
 ---> Running in c5d4dd2b3db9
 ---> eb9f10384509
Removing intermediate container c5d4dd2b3db9
Step 2 : CMD date
 ---> Running in ffb9303ab124
 ---> 4778dd1f1a7a
Removing intermediate container ffb9303ab124
Successfully built 4778dd1f1a7a
$ 
```

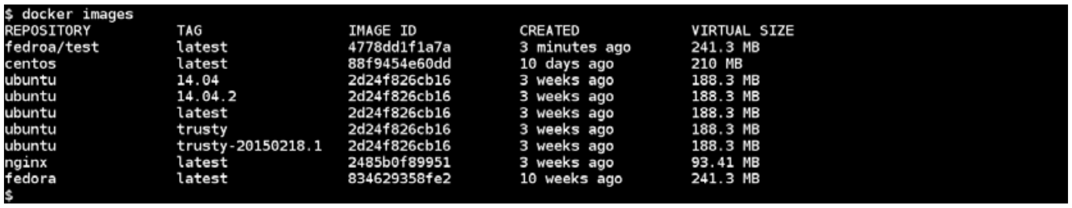
We did not specify any repository or tag name while building the image. We can give those with the `-t` option as follows:

```
$ docker build -t fedora/test .
```



```
$ docker build -t fedora/test .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM fedora
 ---> 834629358fe2
Step 1 : MAINTAINER Neependra Khare
 ---> Using cache
 ---> eb9f10384509
Step 2 : CMD date
 ---> Using cache
 ---> 4778dd1f1a7a
Successfully built 4778dd1f1a7a
```

The preceding output is different from what we did earlier. However, here we are using a cache after each instruction. Docker tries to save the intermediate images as we saw earlier and tries to use them in subsequent builds to accelerate the build process. If you don't want to cache the intermediate images, then add the `--no-cache` option with the build. Let's take a look at the available images now:



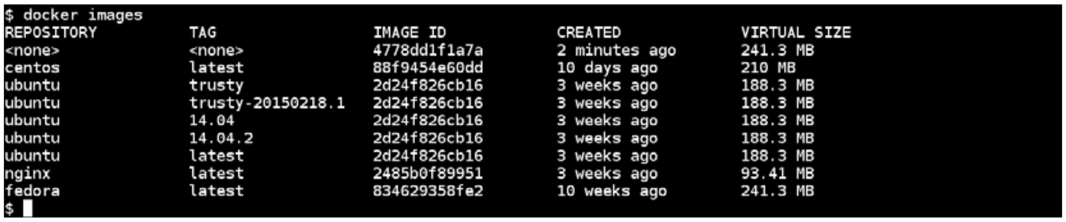
```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
fedroa/test         latest              4778dd1f1a7a        3 minutes ago       241.3 MB
centos              latest              88f9454e60dd        10 days ago         210 MB
ubuntu              14.04               2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              14.04.2             2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              latest              2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              trusty              2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              trusty-20150218.1   2d24f826cb16        3 weeks ago         188.3 MB
nginx               latest              2485b0f89951        3 weeks ago         93.41 MB
fedora              latest              834629358fe2        10 weeks ago        241.3 MB
$
```

How it works…

A context defines the files used to build the Docker image. In the preceding command, we define the context to the build. The build is done by the Docker daemon and the entire context is transferred to the daemon. This is why we see the `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 by the build context. More details about `.dockerignore` can be found at `https://docs.docker.com/reference/builder/#the-dockerignore-file`.

After executing each instruction, Docker commits the intermediate image and runs a container with it for the next instruction. After the next instruction has run, Docker will again commit the container to create the intermediate image and remove the intermediate container created in the previous step.

For example, in the preceding screenshot, `eb9f10384509` is an intermediate image and `c5d4dd2b3db9` and `ffb9303ab124` are the intermediate containers. After the last instruction is executed, the final image will be created. In this case, the final image is `4778dd1f1a7a`:



```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
<none>              <none>              4778dd1f1a7a        2 minutes ago       241.3 MB
centos              latest              88f9454e60dd        10 days ago         210 MB
ubuntu              trusty              2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              trusty-20150218.1   2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              14.04               2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              14.04.2             2d24f826cb16        3 weeks ago         188.3 MB
ubuntu              latest              2d24f826cb16        3 weeks ago         188.3 MB
nginx               latest              2485b0f89951        3 weeks ago         93.41 MB
fedora              latest              834629358fe2        10 weeks ago        241.3 MB
$ █
```

The `-a` option can be specified with the `docker images` command to look for intermediate layers:

```
$ docker images -a
```

There's more…

The format of the Dockerfile is:

```
INSTRUCTION arguments
```

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

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

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

  ```
  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 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 mention this as follows:

```
            [registry_hostname[:port]/][user_name/](repository_
name:version_tag)
```

Here is an example using the preceding syntax:

```
FROM registry-host:5000/nkhare/f20:httpd
```

- `MAINTAINER`: This sets the author for the generated image, `MAINTAINER <name>`.
- `RUN`: We can execute the `RUN` instruction in two ways—first, run in the shell (`sh -c`):

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

  Second, 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 it 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 to 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 as `LABEL distro=fedora21`.
- `CMD`: The `CMD` instruction provides a default executable while starting a container. If the `CMD` instruction does not have an 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 a 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 at max one instruction for `ENTRYPOINT`; if more than 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 to `ENTRYPOINT`.

- `EXPOSE`: This exposes the 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 all the future 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 one is for the 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 the files/directories from the source will be copied.

- `COPY`: This is similar to `ADD.COPY <src> <dest>`:

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

- `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 the `RUN`, `CMD`, and `ENTRYPOINT` instructions that follow it. It can have multiple entries in the same Dockerfile. A relative path can be given which 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 of another image. This trigger will run as part of the `FROM` instruction in downstream Dockerfile using the following syntax:

  ```
  ONBUILD [INSTRUCTION]
  ```

See also

- Look at the `help` option of `docker build`:

  ```
  $ docker build -help
  ```

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

Building an Apache image – a Dockerfile example

I am going to refer Dockerfiles from the Fedora-Dockerfiles GitHub repo (`https://github.com/fedora-cloud/Fedora-Dockerfiles`) after forking it. If you are using Fedora, then you can also install the `fedora-dockerfiles` package to get the sample Dockerfiles in `/usr/share/fedora-dockerfiles`. In each of the subdirectories, you will put a Dockerfile, the supporting files and a README file.

The Fedora-Dockerfiles GitHub repo would have the latest examples and I highly recommend that you try out latest bits.

Getting ready

Clone the Fedora-Dockerfiles Git repo using the following command:

```
$ git clone https://github.com/nkhare/Fedora-Dockerfiles.git
```

Now, go to the `apache` subdirectory:

```
$ cd Fedora-Dockerfiles/apache/
$ cat Dockerfile
FROM fedora:20
```

```
MAINTAINER "Scott Collier" <scollier@redhat.com>

RUN yum -y update && yum clean all
RUN yum -y install httpd && yum clean all
RUN echo "Apache" >> /var/www/html/index.html

EXPOSE 80

# Simple startup script to avoid some issues observed with container
restart
ADD run-apache.sh /run-apache.sh
RUN chmod -v +x /run-apache.sh

CMD ["/run-apache.sh"]
```

The other supporting files are:

- `README.md`: This is the README file
- `run-apache.sh`: This is the script to run `HTTPD` in the foreground
- `LICENSE`: This is the GPL license

How to do it…

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

```
$ docker build -t fedora/apache .
Sending build context to Docker daemon 23.55 kB
Sending build context to Docker daemon
Step 0 : FROM fedora:20
 ---> 6cece30db4f9
Step 1 : MAINTAINER "Scott Collier" <scollier@redhat.com>
 ---> Running in 2048200e6338
 ---> ae8e3c258061
Removing intermediate container 2048200e6338
Step 2 : RUN yum -y update && yum clean all
 ---> Running in df8bc8ee3117
.... Installing/Update packages ...
Cleaning up everything
```

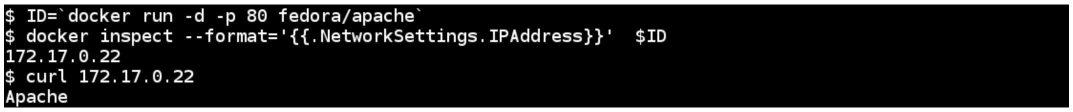
```
 ---> 5a6d449e59f6
Removing intermediate container df8bc8ee3117
Step 3 : RUN yum -y install httpd && yum clean all
 ---> Running in 24449e520f18
.... Installing HTTPD ...
Cleaning up everything
 ---> ae1625544ef6
Removing intermediate container 24449e520f18
Step 4 : RUN echo "Apache" >> /var/www/html/index.html
 ---> Running in a35cbcd8d97a
 ---> 251eea31b3ce
Removing intermediate container a35cbcd8d97a
Step 5 : EXPOSE 80
 ---> Running in 734e54f4bf58
 ---> 19503ae2a8cf
Removing intermediate container 734e54f4bf58
Step 6 : ADD run-apache.sh /run-apache.sh
 ---> de35d746f43b
Removing intermediate container 3eec9a46da64
Step 7 : RUN chmod -v +x /run-apache.sh
 ---> Running in 3664efba393f
mode of '/run-apache.sh' changed from 0644 (rw-r--r--) to 0755 (rwxr-
xr-x)
 ---> 1cb729521c3f
Removing intermediate container 3664efba393f
Step 8 : CMD /run-apache.sh
 ---> Running in cd5e7534e815
 ---> 5f8041b6002c
Removing intermediate container cd5e7534e815
Successfully built 5f8041b6002c
```

How it works...

The build process takes a base image, installs the required `HTTPD` package and creates an HTML page. Then, it exposes port `80` to serve the web page and sets instructions to start Apache at the start of the container.

There's more...

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



```
$ ID=`docker run -d -p 80 fedora/apache`
$ docker inspect --format='{{.NetworkSettings.IPAddress}}'  $ID
172.17.0.22
$ curl 172.17.0.22
Apache
```

See also

- Look at the `help` option of `docker build`:

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

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

Accessing Firefox from a container – a Dockerfile example

We can do something more interesting through a Dockerfile, such as creating a container that just runs Firefox. This kind of use case can help run multiple browsers of different versions on the same machine, which can be very helpful in doing multibrowser testing.

Getting ready

Clone the Fedora-Dockerfiles Git repo using the following command:

```
$ git clone  https://github.com/nkhare/Fedora-Dockerfiles.git
```

Then, go to the `firefox` subdirectory.

```
$ cd Fedora-Dockerfiles/firefox
$ cat Dockerfile
FROM fedora
MAINTAINER scollier <emailscottcollier@gmail.com>

# Install the appropriate software
RUN yum -y update && yum clean all
RUN yum -y install x11vnc \
```

```
firefox xorg-x11-server-Xvfb \
xorg-x11-twm tigervnc-server \
xterm xorg-x11-font \
xulrunner-26.0-2.fc20.x86_64 \
dejavu-sans-fonts \
dejavu-serif-fonts \
xdotool && yum clean all

# Add the xstartup file into the image
ADD ./xstartup /

RUN mkdir /.vnc
RUN x11vnc -storepasswd 123456 /.vnc/passwd
RUN  \cp -f ./xstartup /.vnc/.
RUN chmod -v +x /.vnc/xstartup
RUN sed -i '/\/etc\/X11\/xinit\/xinitrc-common/a [ -x
/usr/bin/firefox ] && /usr/bin/firefox &' /etc/X11/xinit/xinitrc

EXPOSE 5901


CMD    ["vncserver", "-fg" ]
# ENTRYPOINT ["vncserver", "-fg" ]
```

Supporting files:

- `README.md`: This is a README file
- `LICENSE`: This is the GPL license
- `xstartup`: This is the script to set up the X11 environment

How to do it...

Run the following command to build the image:

```
$ docker build  -t fedora/firefox .
Sending build context to Docker daemon 24.58 kB
Sending build context to Docker daemon
Step 0 : FROM fedora
 ---> 834629358fe2
Step 1 : MAINTAINER scollier <emailscottcollier@gmail.com>
```

```
 ---> Running in ae0fd3c2cb2e
 ---> 7ffc6c9af827
Removing intermediate container ae0fd3c2cb2e
Step 2 : RUN yum -y update && yum clean all
 ---> Running in 1c67b8772718
..... Installing/Update packages ...
 ---> 075d6ceef3d0
Removing intermediate container 1c67b8772718
Step 3 : RUN yum -y install x11vnc firefox xorg-x11-server-Xvfb xorg-
x11-twm tigervnc-server xterm xorg-x11-font xulrunner-26.0-
2.fc20.x86_64 dejavu-sans-fonts dejavu-serif-fonts xdotool && yum
clean all
..... Installing required packages packages ...
Cleaning up everything
 ---> 986be48760a6
Removing intermediate container c338a1ad6caf
Step 4 : ADD ./xstartup /
 ---> 24fa081dcea5
Removing intermediate container fe98d86ba67f
Step 5 : RUN mkdir /.vnc
 ---> Running in fdb8fe7e697a
 ---> 18f266ace765
Removing intermediate container fdb8fe7e697a
Step 6 : RUN x11vnc -storepasswd 123456 /.vnc/passwd
 ---> Running in c5b7cdba157f
stored passwd in file: /.vnc/passwd
 ---> e4fcf9b17aa9
Removing intermediate container c5b7cdba157f
Step 7 : RUN \cp -f ./xstartup /.vnc/.
 ---> Running in 21d0dc4edb4e
 ---> 4c53914323cb
Removing intermediate container 21d0dc4edb4e
Step 8 : RUN chmod -v +x /.vnc/xstartup
 ---> Running in 38f18f07c996
mode of '/.vnc/xstartup' changed from 0644 (rw-r--r--) to 0755 (rwxr-
xr-x)
 ---> caa278024354
```

```
Removing intermediate container 38f18f07c996
Step 9 : RUN sed -i '/\/etc\/X11\/xinit\/xinitrc-common/a [ -x /usr/bin/
firefox ] && /usr/bin/firefox &' /etc/X11/xinit/xinitrc
 ---> Running in 233e99cab02c
 ---> 421e944ac8b7
Removing intermediate container 233e99cab02c
Step 10 : EXPOSE 5901
 ---> Running in 530cd361cb3c
 ---> 5de01995c156
Removing intermediate container 530cd361cb3c
Step 11 : CMD vncserver -fg
 ---> Running in db89498ae8ce
 ---> 899be39b7feb
Removing intermediate container db89498ae8ce
Successfully built 899be39b7feb
```

How it works...

We start with the base Fedora image, install X Windows System, Firefox, a VNC server, and other packages. We then set up the VNC server to start X Windows System, which will start Firefox.

There's more...

- To start the container, run the following command:

  ```
  $ docker run -it -p 5901:5901 fedora/firefox
  ```

  And give `123456` as the password.

- While running the container, we mapped the `5901` port of the host to `5901` port of the container. In order to connect to the VNC server inside the container, just run the following command from another terminal:

  ```
  $ vncviewer localhost:1
  ```

  Alternatively, from another machine in the network, replace `localhost` with the Docker host's IP address or FQDN.

See also

- Look at the `help` option of `docker build`:

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

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

Building a WordPress image – a Dockerfile example

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

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

Again, we will use a Dockerfile from the Fedora-Dockerfiles repository.

Getting ready

Clone the Fedora-Dockerfiles Git repo using the following command:

```
$ git clone  https://github.com/nkhare/Fedora-Dockerfiles.git
```

Then, go to the `wordpress_single_container` subdirectory:

```
$ cd Fedora-Dockerfiles/systemd/wordpress_single_container
$ cat Dockerfile
FROM fedora
MAINTAINER scollier <scollier@redhat.com>
RUN yum -y update && yum clean all
RUN yum -y install httpd php php-mysql php-gd pwgen supervisor bash-
completion openssh-server psmisc tar && yum clean all
ADD ./start.sh /start.sh
ADD ./foreground.sh /etc/apache2/foreground.sh
ADD ./supervisord.conf /etc/supervisord.conf
RUN echo %sudo  ALL=NOPASSWD: ALL >> /etc/sudoers
ADD http://wordpress.org/latest.tar.gz /wordpress.tar.gz
RUN tar xvzf /wordpress.tar.gz
RUN mv /wordpress/* /var/www/html/.
RUN chown -R apache:apache /var/www/
RUN chmod 755 /start.sh
RUN chmod 755 /etc/apache2/foreground.sh
RUN mkdir /var/run/sshd
EXPOSE 80
EXPOSE 22
CMD ["/bin/bash", "/start.sh"]
```

The supporting files used in the preceding code are explained as follows:

- `foreground.sh`: This is a script to run HTTPS in the foreground.
- `LICENSE`, `LICENSE.txt`, and `UNLICENSE.txt`: These files contain the license information.
- `README.md`: This is a README file.
- `supervisord.conf`: This is a resulting container which will have to run `SSHD`, `MySQL`, and `HTTPD` at the same time. In this particular case, the supervisor is used to manage them. It is a configuration file of the supervisor. More information about this can be found at `http://supervisord.org/`.
- `start.sh`: This is a script to set up MySQL, HTTPD, and to start the supervisor daemon.

How to do it...

```
$ docker build -t fedora/wordpress  .
Sending build context to Docker daemon 41.98 kB
Sending build context to Docker daemon
Step 0 : FROM fedora
 ---> 834629358fe2
Step 1 : MAINTAINER scollier <scollier@redhat.com>
 ---> Using cache
 ---> f21eaf47c9fc
Step 2 : RUN yum -y update && yum clean all
 ---> Using cache
 ---> a8f497a6e57c
Step 3 : RUN yum -y install httpd php php-mysql php-gd pwgen supervisor
bash-completion openssh-server psmisc tar && yum clean all
 ---> Running in 303234ebf1e1
.... updating/installing packages ....
Cleaning up everything
 ---> cc19a5f5c4aa
```

```
Removing intermediate container 303234ebf1e1
Step 4 : ADD ./start.sh /start.sh
 ---> 3f911077da44
Removing intermediate container c2bd643236ef
Step 5 : ADD ./foreground.sh /etc/apache2/foreground.sh
 ---> 3799902a60c5
Removing intermediate container c99b8e910009
Step 6 : ADD ./supervisord.conf /etc/supervisord.conf
 ---> f232433b8925
Removing intermediate container 0584b945f6f7
Step 7 : RUN echo %sudo  ALL=NOPASSWD: ALL >> /etc/sudoers
 ---> Running in 581db01d7350
 ---> ec686e945dfd
Removing intermediate container 581db01d7350
Step 8 : ADD http://wordpress.org/latest.tar.gz /wordpress.tar.gz
Downloading [==================================================>] 6.186
MB/6.186 MB
 ---> e4e902c389a4
Removing intermediate container 6bfecfbe798d
Step 9 : RUN tar xvzf /wordpress.tar.gz
 ---> Running in cd772500a776
.......... untarring wordpress .........
---> d2c5176228e5
Removing intermediate container cd772500a776
Step 10 : RUN mv /wordpress/* /var/www/html/.
 ---> Running in 7b19abeb509c
 ---> 09400817c55f
Removing intermediate container 7b19abeb509c
Step 11 : RUN chown -R apache:apache /var/www/
 ---> Running in f6b9b6d83b5c
 ---> b35a901735d9
Removing intermediate container f6b9b6d83b5c
Step 12 : RUN chmod 755 /start.sh
 ---> Running in 81718f8d52fa
 ---> 87470a002e12
Removing intermediate container 81718f8d52fa
```

```
Step 13 : RUN chmod 755 /etc/apache2/foreground.sh
 ---> Running in 040c09148e1c
 ---> 1c76f1511685
Removing intermediate container 040c09148e1c
Step 14 : RUN mkdir /var/run/sshd
 ---> Running in 77177a33aee0
 ---> f339dd1f3e6b
Removing intermediate container 77177a33aee0
Step 15 : EXPOSE 80
 ---> Running in f27c0b96d17f
 ---> 6078f0d7b70b
Removing intermediate container f27c0b96d17f
Step 16 : EXPOSE 22
 ---> Running in eb7c7d90b860
 ---> 38f36e5c7cab
Removing intermediate container eb7c7d90b860
Step 17 : CMD /bin/bash /start.sh
 ---> Running in 5635fe4783da
 ---> c1a327532355
Removing intermediate container 5635fe4783da
Successfully built c1a327532355
```

How it works…

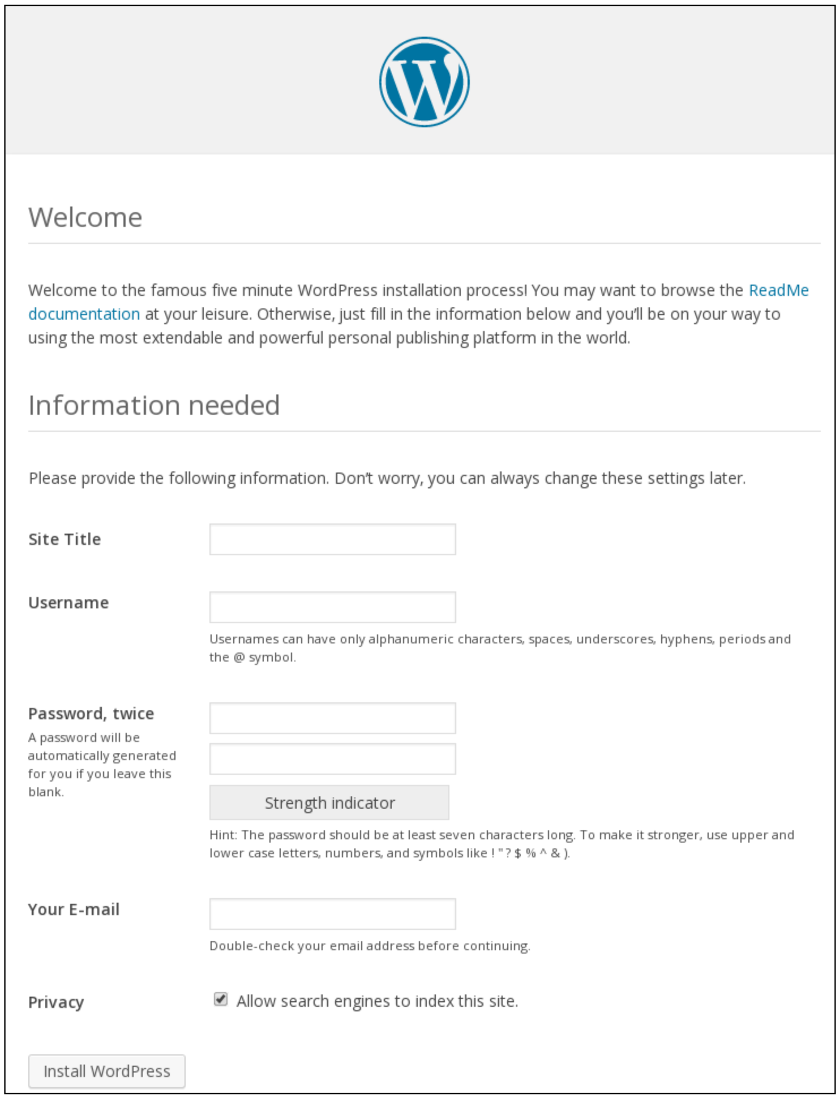
As with the other recipes, we start with the base image, install the required packages, and copy the supporting files. We will then set up `sudo`, `download`, and `untar` WordPress inside the HTTPD document root. After this, we expose the ports and run the start.sh scripts, which sets up MySQL, WordPress, HTTPS permissions and gives control to supervisord. In the `supervisord.conf`, you will see entries, such as the following services that supervisord manages:

```
[program:mysqld]
command=/usr/bin/mysqld_safe
[program:httpd]
command=/etc/apache2/foreground.sh
stopsignal=6
[program:sshd]
command=/usr/sbin/sshd -D
```

```
stdout_logfile=/var/log/supervisor/%(program_name)s.log
stderr_logfile=/var/log/supervisor/%(program_name)s.log
autorestart=true
```

There's more...

- Start the container, get its IP address and open it through a web browser. You should see the Welcome screen, as shown in the following screenshot, after doing the language selection:



Welcome

Welcome to the famous five minute WordPress installation process! You may want to browse the ReadMe documentation at your leisure. Otherwise, just fill in the information below and you'll be on your way to using the most extendable and powerful personal publishing platform in the world.

Information needed

Please provide the following information. Don't worry, you can always change these settings later.

Site Title

Username

Usernames can have only alphanumeric characters, spaces, underscores, hyphens, periods and the @ symbol.

Password, twice

A password will be automatically generated for you if you leave this blank.

Strength indicator

Hint: The password should be at least seven characters long. To make it stronger, use upper and lower case letters, numbers, and symbols like ! " ? $ % ^ &).

Your E-mail

Double-check your email address before continuing.

Privacy ✓ Allow search engines to index this site.

Install WordPress

- It is now possible to run systemd inside the container, which is a more preferred way. Systemd can manage more than one service .You can look at the example of systemd at `https://github.com/fedora-cloud/Fedora-Dockerfiles/tree/master/systemd`.

See also

- Look at the `help` option of `docker build`:

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

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

Setting up a private index/registry

As we saw earlier, the public Docker registry is the available Docker Hub (`https://registry.hub.docker.com/`) through which users can push/pull images. We can also host a private registry either on a local environment or on the cloud. There are a few ways to set up the local registry:

- Use the Docker registry from Docker Hub
- Build an image from Dockerfile and run a registry container:

  `https://github.com/fedora-cloud/Fedora-Dockerfiles/tree/master/registry`

- Configure the distribution-specific package such as Fedora, which provides the docker-registry package that you can install and configure.

The easiest way to set it up is through the registry container itself.

Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it...

1. To run the registry on the container, run the following command:

   ```
   $ docker run -p 5000:5000 registry
   ```

2. To test the newly created registry, perform the following steps:

1. Start a container and its ID by using the following command:

   ```
   $ ID='docker run -d -i fedora /bin/bash'
   ```

2. If needed, attach to the newly created container and make some changes. Then, commit those changes to the local repository:

   ```
   $ docker commit $ID fedora-20
   ```

3. To push the image to the local registry, we need to tag the image with the hostname or IP address of the registry host. Let's say our registry host is `registry-host`; then, to tag it, use the following command:

   ```
   $ docker tag fedora-20 registry-host:5000/nkhare/f20
   ```

4. As we have not configured HTTPS correctly while starting the registry, we will get an error such as the `ping attempt failed with error: Get https://dockerhost:5000/v1/_ping`, which is expected. For our example to work, we need to add the `--insecure-registry registry-host:5000` option to the daemon. If you have started the Docker daemon manually, then we have to run the command as follows to allow insecure registry:

   ```
   $ docker -d   --insecure-registry registry-host:5000
   ```

5. To push the image, use the following command:

   ```
   $ docker push registry-host:5000/nkhare/f20
   ```

6. To pull the image from the local registry, run the following command:

   ```
   $ docker pull registry-host:5000/nkhare/f20
   ```

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 look at the details about port publishing in the next chapter.

The registry can also be configured on any existing servers using the docker-registry app. The steps to do this are available at the docker-registry GitHub page:

`https://github.com/docker/docker-registry`

There's more...

Let's look at Dockerfile of docker-registry to understand how the registry image is being created and how to set different configuration options:

```
# VERSION 0.1
# DOCKER-VERSION  0.7.3
```

```
# AUTHOR:          Sam Alba <sam@docker.com>
# DESCRIPTION:     Image with docker-registry project and dependencies
# TO_BUILD:        docker build -rm -t registry .
# TO_RUN:          docker run -p 5000:5000 registry

# Latest Ubuntu LTS
FROM ubuntu:14.04

# Update
RUN apt-get update \
# Install pip
    && apt-get install -y \
        swig \
        python-pip \
# Install deps for backports.lzma (python2 requires it)
        python-dev \
        python-mysqldb \
        python-rsa \
        libssl-dev \
        liblzma-dev \
        libevent1-dev \
    && rm -rf /var/lib/apt/lists/*

COPY . /docker-registry
COPY ./config/boto.cfg /etc/boto.cfg

# Install core
RUN pip install /docker-registry/depends/docker-registry-core

# Install registry
RUN pip install file:///docker-registry#egg=docker-
registry[bugsnag,newrelic,cors]

RUN patch \
 $(python -c 'import boto; import os; print
os.path.dirname(boto.__file__)')/connection.py \
 < /docker-registry/contrib/boto_header_patch.diff

ENV DOCKER_REGISTRY_CONFIG /docker-registry/config/config_sample.yml
ENV SETTINGS_FLAVOR dev
EXPOSE 5000
CMD ["docker-registry"]
```

With the preceding Dockerfile, we will:

- Take Ubuntu's base image install/update packages
- Copy the docker-registry source code inside the image
- Use the `pip install` docker-registry
- Set up the configuration file to use while running the registry using the environment variable
- Set up the flavor to use while running the registry using the environment variable
- Expose port `5000`
- Run the registry executable

Flavors in the configuration file (`/docker-registry/config/config_sample.yml`) provide different ways to configure the registry. With the preceding Dockerfile, we will set the `dev` flavor using the environment variables. The different types of flavors are:

- `common`: This is used by all the other flavors as base settings
- `local`: This stores data on the local filesystem
- `s3`: This stores data in an AWS S3 bucket
- `dev`: This is the basic configuration using the local flavors
- `test`: This is used by unit tests
- `prod`: This is the production configuration (basically a synonym for the S3 flavor)
- `gcs`: This stores data in Google cloud storage
- `swift`: This stores data in OpenStack Swift
- `glance`: This stores data in OpenStack Glance, with a fallback to the local storage
- `glance-swift`: This stores data in OpenStack Glance, with a fallback to Swift
- `elliptics`: This stores data in Elliptics key-value storage

For each of preceding flavors, different configuration options such as loglevel, authentication, and so on are available. The documentation for all of the options are available on the GitHub page of docker-registry, which I mentioned earlier.

See also

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

Automated builds – with GitHub and Bitbucket

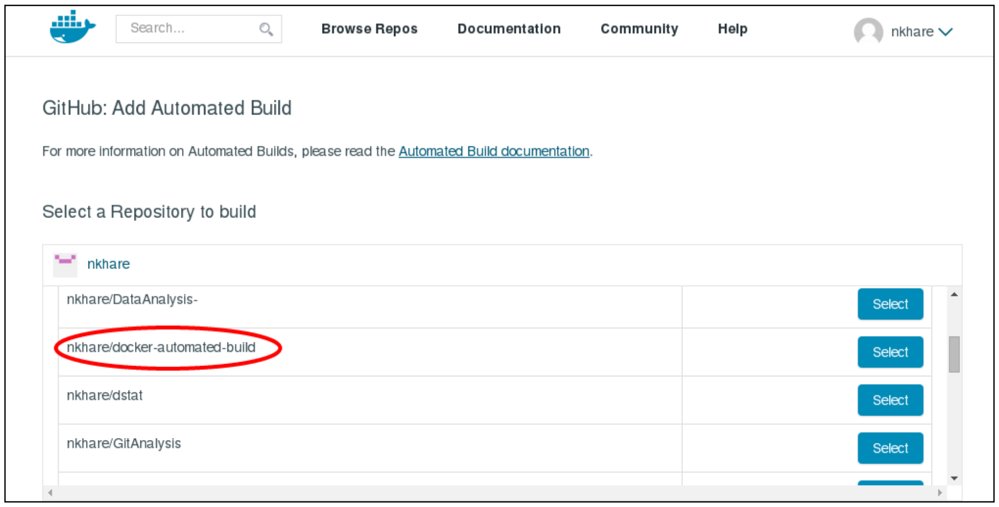
We have seen earlier how to push the Docker images to Docker Hub. Docker Hub allows us to create automated images from a GitHub/Bitbucket repository using its build clusters. The GitHub/Bitbucket repository should contain the Dockerfile and the content required to copy/add inside the image. Let's look at a GitHub example in the upcoming sections.

Getting ready

You will need an account on Docker Hub and GitHub. You will also need a GitHub repository with a corresponding Dockerfile at the top level.

How to do it…

1. Log in to Docker Hub (`https://hub.docker.com/`) and click on the green plus sign. Add the Repository icon on the top right-hand side corner and click on **Automated Build**. Select GitHub as a source to use for automated build. Then, select the **Public and Private (recommended)** option to connect to GitHub. Provide the GitHub username/password when prompted. Select the GitHub repository to perform automated build.



GitHub: Add Automated Build

For more information on Automated Builds, please read the Automated Build documentation.

Select a Repository to build

nkhare

nkhare/DataAnalysis-

nkhare/docker-automated-build

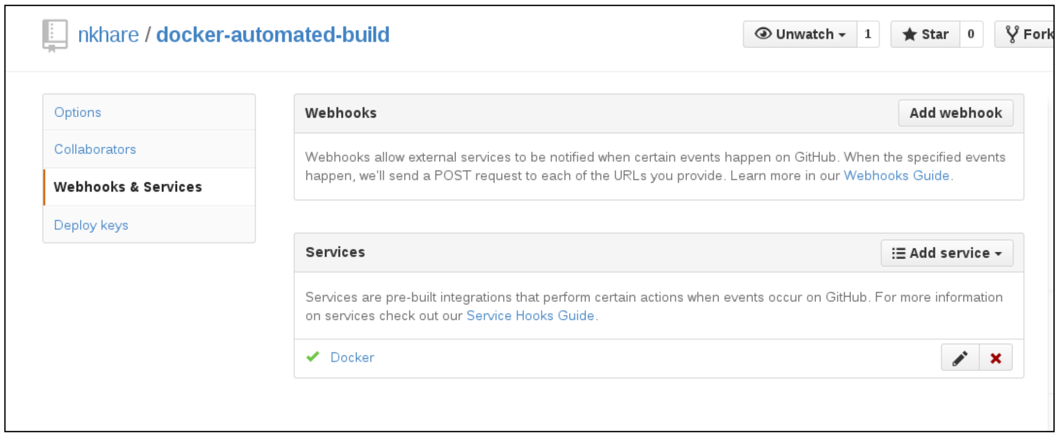
nkhare/dstat

nkhare/GitAnalysis

2. After selecting the GitHub repository, it will ask you to pick its branch to use for automated build. It will also ask for a tag name to use after the image it automatically built. By default, the latest tag name will be used. Then, click on the **Save and trigger build** button to start the automated build process. That's it!! Your build is now submitted. You can click on the build status to check the status of the build.

How it works...

When we select a GitHub repository for automated build, GitHub enables the Docker service for that repository. You can look at the **Settings** section of the GitHub repository for more configuration. Whenever we make any changes to this GitHub repository, such as commits, an automated build gets triggered using the Dockerfile that resides in the GitHub repository.

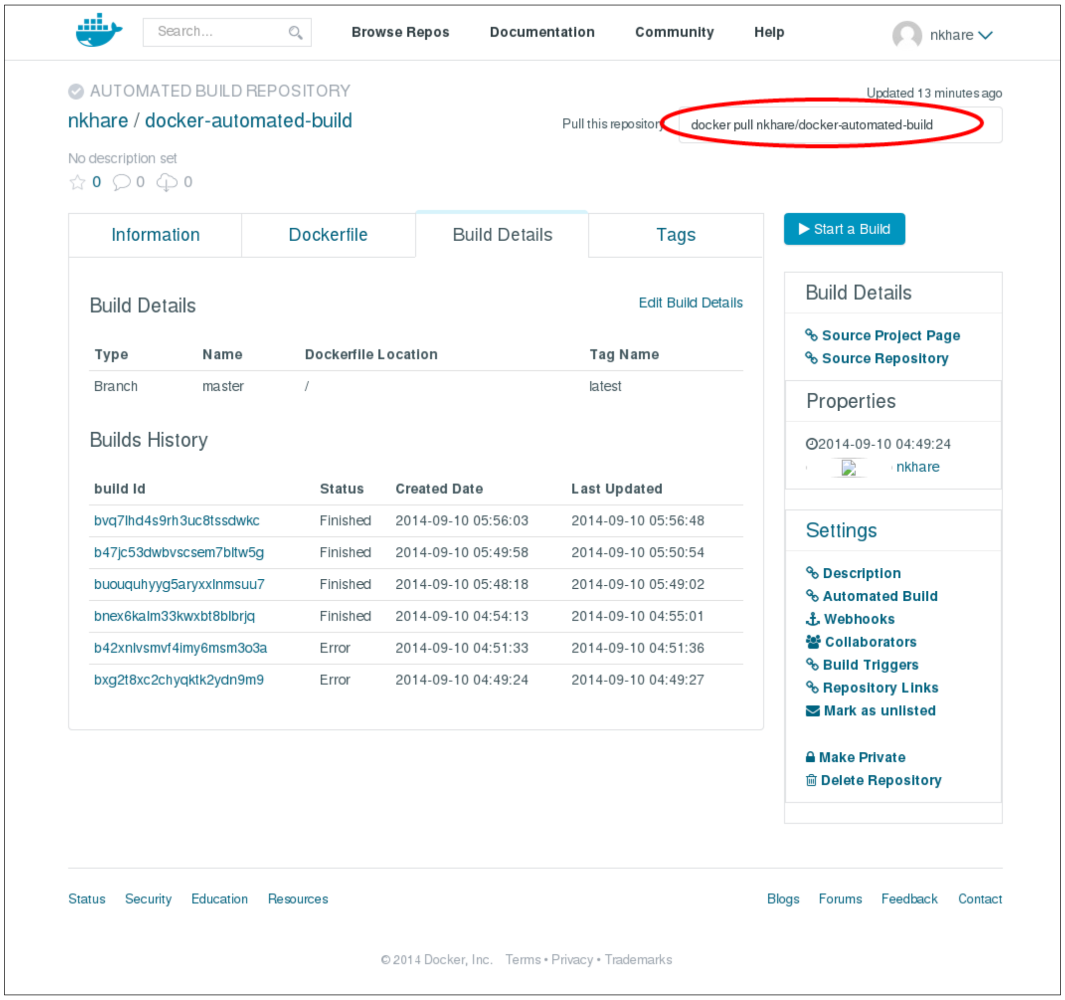


Webhooks allow external services to be notified when certain events happen on GitHub. When the specified events happen, we'll send a POST request to each of the URLs you provide. Learn more in our Webhooks Guide.

Services are pre-built integrations that perform certain actions when events occur on GitHub. For more information on services check out our Service Hooks Guide.

There's more...

You can get the details such as the Dockerfile, build details tags, and other information, by going to the **Your Repositories** section. It also has the details of how to pull your image:



AUTOMATED BUILD REPOSITORY

nkhare / docker-automated-build

Updated 13 minutes ago

Pull this repository `docker pull nkhare/docker-automated-build`

No description set

☆ 0 0 0

Build Details

Type	Name	Dockerfile Location	Tag Name
Branch	master	/	latest

Builds History

build Id	Status	Created Date	Last Updated
bvq7lhd4s9rh3uc8tssdwkc	Finished	2014-09-10 05:56:03	2014-09-10 05:56:48
b47jc53dwbvscsem7bltw5g	Finished	2014-09-10 05:49:58	2014-09-10 05:50:54
buouquhyyg5aryxxlnmsuu7	Finished	2014-09-10 05:48:18	2014-09-10 05:49:02
bnex6kalm33kwxbt8blbrjq	Finished	2014-09-10 04:54:13	2014-09-10 04:55:01
b42xnlvsmvf4imy6msm3o3a	Error	2014-09-10 04:51:33	2014-09-10 04:51:36
bxg2t8xc2chyqktk2ydn9m9	Error	2014-09-10 04:49:24	2014-09-10 04:49:27

Build Details

Properties

2014-09-10 04:49:24
nkhare

Settings

© 2014 Docker, Inc. Terms • Privacy • Trademarks

The images that get created using the automated build process cannot be pushed through the `docker push` command.

You can change the settings in the **Webhooks & Services** section of the repository on GitHub to unregister the Docker service. This will stop doing the automated builds.

See also

- The steps for setting up automated build with Bitbucket are almost identical. The hook for automated build gets configured under the **Hooks** section of Bitbucket repository's **Settings** section.
- The documentation on the Docker website `https://docs.docker.com/docker-hub/builds/`

Creating the base image – using supermin

Earlier in this chapter, we used the `FROM` instruction to pick the base image to start with. The image we create can become the base image to containerize another application and so on. From the very beginning to this chain, we will have a base image from the underlying Linux distribution that we want to use such as Fedora, Ubuntu, CentOS, and so on.

To build such a base image, we will need to have a distribution-specific base system installed into a directory, which can then be imported as an image to Docker. With chroot utility, we can fake a directory as the root filesystem and then put all the necessary files inside it before importing it as a Docker image. Supermin and Debootstrap are the kind of tools that can help us make the preceding process easier.

Supermin is a tool to build supermin appliances. These are tiny appliances, which get fully instantiated on the fly. Earlier this program was called febootstrap.

Getting ready

Install supermin on the system where you want to build the base image. You can install supermin on Fedora with the following command:

```
$ yum install supermin
```

How to do it...

1. Using the `prepare` mode install `bash`, `coreutils`, and the related dependencies inside a directory.

   ```
   $ supermin --prepare -o OUTPUTDIR PACKAGE [PACKAGE ...]
   ```

   Here's an example using the preceding syntax:

   ```
   $ supermin --prepare bash coreutils -o f21_base
   ```

2. Now, with the `build` mode, create a chrooted environment for the base image:

   ```
   $ supermin --build -o OUTPUTDIR -f chroot|ext2 INPUT [INPUT ...]
   ```

Here's an example using the preceding syntax :

```
$ supermin --build --format chroot f21_base -o f21_image
```

3. If we do `ls` on the output directory, we will see a directory tree similar to any Linux root filesystem:

```
$ ls f21_image/
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root
run  sbin  srv  sys  tmp  usr  var
```

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

```
$ tar -C f21_image/ -c . | docker import - nkhare/f21_base
d6db8b798dee30ad9c84480ef7497222f063936a398ecf639e60599eed7f6560
```

5. Now, look at the `docker images` output. You should have a new image with `nkhare/f21_base` as the name.

How it works...

Supermins has two modes, `prepare` and `build`. With the `prepare` mode, it just puts all the requested packages with their dependencies inside a directory without copying the host OS specific files.

With the `build` mode, the previously created supermin appliance from the `prepare` mode gets converted into a full blown bootable appliance with all the necessary files. This step will copy the required files/binaries from the host machine to the appliance directory, so the packages must be installed on the host machines that you want to use in the appliance.

The `build` mode has two output formats, chroot, and ext2. With the chroot format, the directory tree gets written into the directory, and with the ext2 format, a disk image gets created. We exported the directory created through the chroot format to create the Docker image.

There's more...

Supermin is not specific to Fedora and should work on any Linux distribution.

See also

- Look at the `man` page of supermin for more information using the following command:

```
$ man supermin
```

- The online documentation `http://people.redhat.com/~rjones/supermin/`
- The GitHub repository `https://github.com/libguestfs/supermin`

Creating the base image – using Debootstrap

Debootstrap is a tool to install a Debian-based system into a directory of an already installed system.

Getting ready

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

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

How to do it...

The following command can be used to create the base image using Debootstrap:

```
$ debootstrap [OPTION...]  SUITE TARGET [MIRROR [SCRIPT]]
```

`SUITE` refers to the release code name and `MIRROR` is the respective repository. If you wanted to create the base image of Ubuntu 14.04.1 LTS (Trusty Tahr), then do the following:

1. Create a directory on which you want to install the OS. Debootstrap also creates the chroot environment to install a package, as we saw earlier with supermin.

   ```
   $ mkdir trusty_chroot
   ```

2. Now, using `debootstrap`, install Trusty Tahr inside the directory we created earlier:

   ```
   $ debootstrap trusty ./trusty_chroot
   http://in.archive.ubuntu.com/ubuntu/
   ```

3. You will see the directory tree similar to any Linux root filesystem, inside the directory in which Trusty Tahr is installed.

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

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

   ```
   $ tar -C trusty_chroot/ -c . |  docker import -
   nkhare/trusty_base
   ```

5. Now, look at the `docker images` output. You should have a new image with `nkhare/trusty_base` as the name.

See also

- The Debootstrap wiki page `https://wiki.debian.org/Debootstrap`.
- There are a few other ways to create base images. You can find links to them at `https://docs.docker.com/articles/baseimages/`.

Visualizing dependencies between layers

As the number of images grows, it becomes difficult to find relation between them. There are a few utilities for which you can find the relation between images.

Getting ready

One or more Docker images on the host running the Docker daemon.

How to do it...

1. Run the following command to get a tree-like view of the images:

   ```
   $ docker images -t
   ```

How it works...

The dependencies between layers will be fetched from the metadata of the Docker images.

There's more...

From `--viz` to `docker images`, we can see dependencies graphically; to do this, you will need to have the `graphviz` package installed:

```
$ docker images --viz | dot -Tpng -o /tmp/docker.png
$ display /tmp/docker.png
```

As it states in the warning that appears when running the preceding commands, the `-t` and `--viz` options might get deprecated soon.

See also

- The following project tries to visualize Docker data as well by using raw JSON output from Docker `https://github.com/justone/dockviz`

4
Network and Data Management for Containers

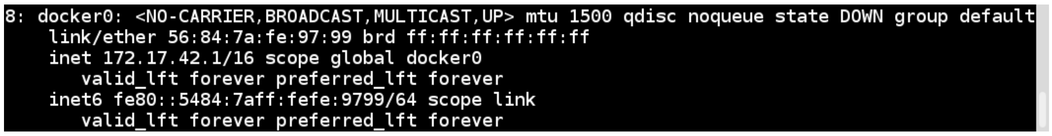
In this chapter, we will cover the following recipes:

- Accessing containers from outside
- Managing data in containers
- Linking two or more containers
- Developing a LAMP application by linking containers
- Networking of multihost container with Flannel
- Assigning IPv6 addresses to containers

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 see how to fulfill some of those requirements. Let's start by understanding Docker's default networking setup and then go to advanced use cases.

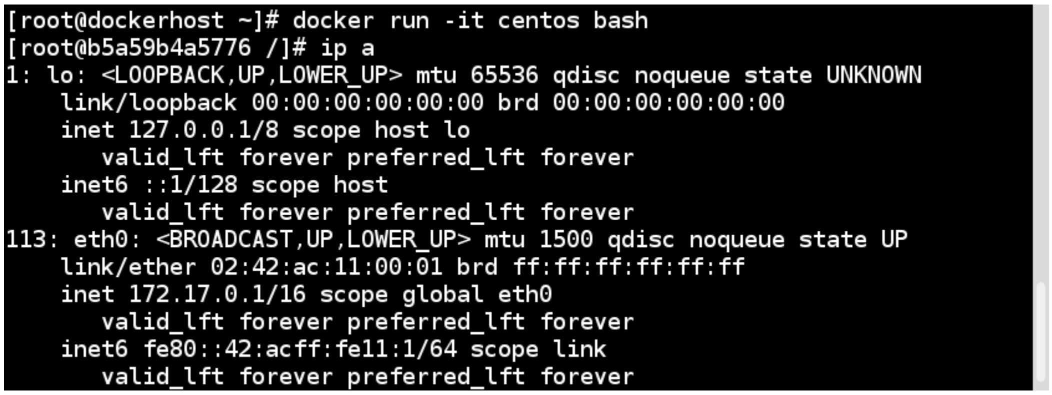
When the Docker daemon starts, it creates a virtual Ethernet bridge with the name `docker0`. For example, we will see the following with the `ip addr` command on the system that runs the Docker daemon:



```
8: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::5484:7aff:fefe:9799/64 scope link
       valid_lft forever preferred_lft forever
```

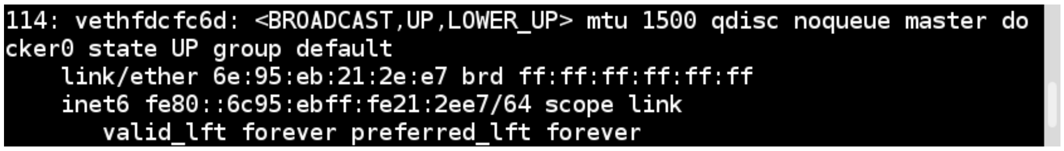
As we can see, `docker0` has the IP address 172.17.42.1/16. Docker randomly chooses an address and subnet from a private range defined in RFC 1918 (`https://tools.ietf.org/html/rfc1918`). Using this bridged interface, containers can communicate with each other and with the host system.

By default, every time Docker starts a container, it creates a pair of virtual interfaces, one end of which is attached to the host system and other end to the created container. Let's start a container and see what happens:



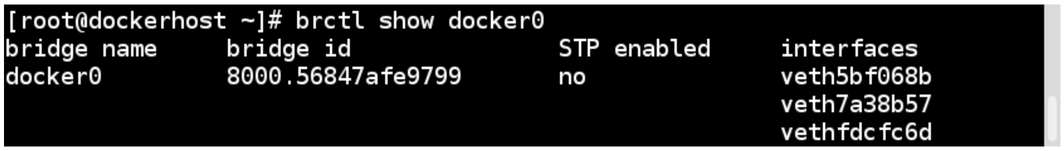
```
[root@dockerhost ~]# docker run -it centos bash
[root@b5a59b4a5776 /]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
113: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:01 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:1/64 scope link
       valid_lft forever preferred_lft forever
```

The end that is attached to the `eth0` interface of the container gets the 172.17.0.1/16 IP address. We also see the following entry for the other end of the interface on the host system:



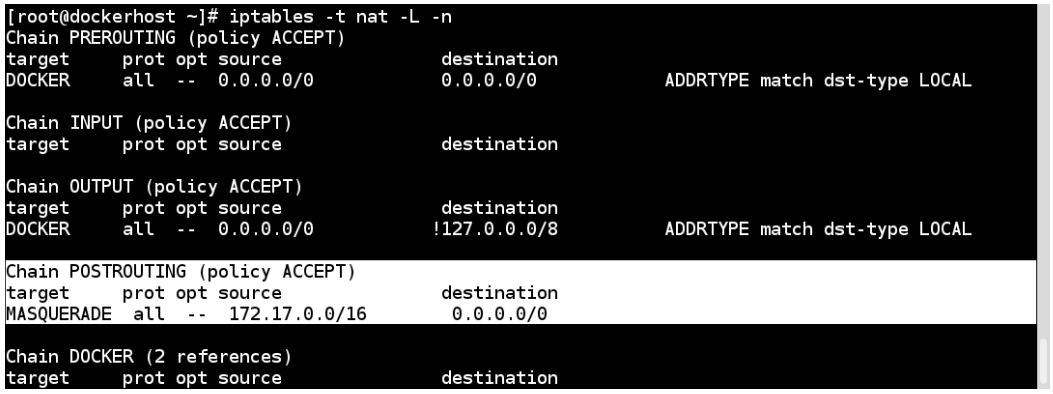
```
114: vethfdcfc6d: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master do
cker0 state UP group default
    link/ether 6e:95:eb:21:2e:e7 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::6c95:ebff:fe21:2ee7/64 scope link
       valid_lft forever preferred_lft forever
```

Now, let's create a few more containers and look at the `docker0` bridge with the `brctl` command, which manages Ethernet bridges:



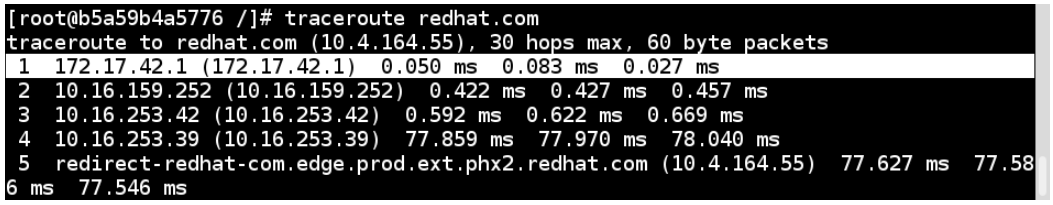
```
[root@dockerhost ~]# brctl show docker0
bridge name     bridge id               STP enabled     interfaces
docker0         8000.56847afe9799       no              veth5bf068b
                                                        veth7a38b57
                                                        vethfdcfc6d
```

Every veth* binds to the `docker0` bridge, which creates a virtual subnet shared between the host and every Docker container. Apart from setting up the `docker0` bridge, Docker creates IPtables NAT rules, such that all containers can talk to the external world by default but not the other way around. Let's look at the NAT rules on the Docker host:



```
[root@dockerhost ~]# 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
```

If we try to connect to the external world from a container, we will have to go through the Docker bridge that was created by default:



```
[root@b5a59b4a5776 /]# traceroute redhat.com
traceroute to redhat.com (10.4.164.55), 30 hops max, 60 byte packets
 1  172.17.42.1 (172.17.42.1)  0.050 ms  0.083 ms  0.027 ms
 2  10.16.159.252 (10.16.159.252)  0.422 ms  0.427 ms  0.457 ms
 3  10.16.253.42 (10.16.253.42)  0.592 ms  0.622 ms  0.669 ms
 4  10.16.253.39 (10.16.253.39)  77.859 ms  77.970 ms  78.040 ms
 5  redirect-redhat-com.edge.prod.ext.phx2.redhat.com (10.4.164.55)  77.627 ms  77.58
6 ms  77.546 ms
```

Later in this chapter, we will see how the external world can connect to a container.

When starting a container, we have a few modes to select its networking:

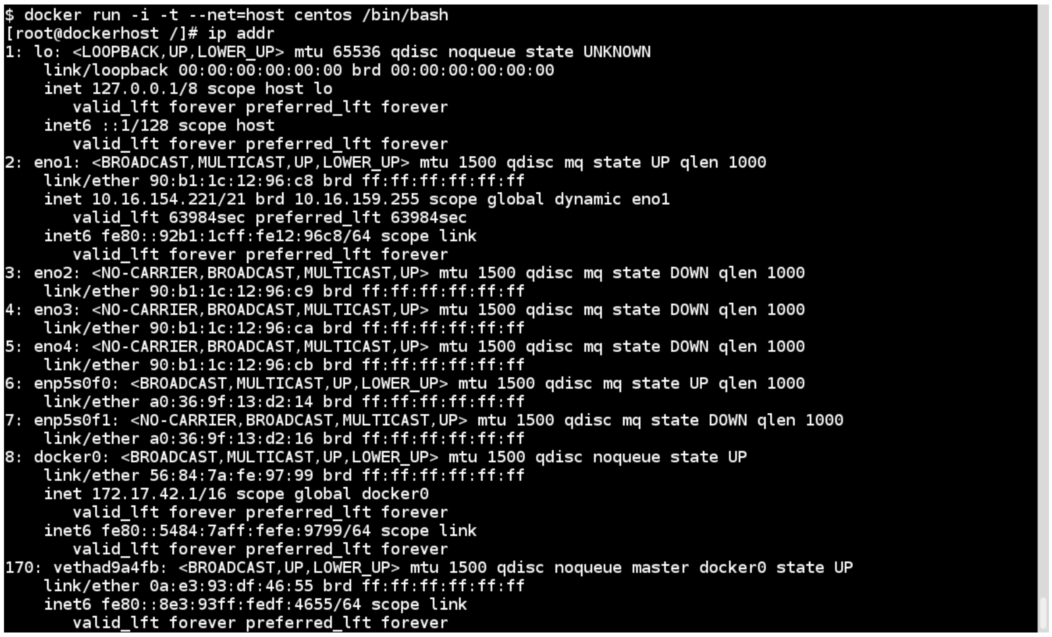
- `--net=bridge`: This is the default mode that we just saw. So, the preceding command that we used to start the container can be written as follows:

  `$ docker run -i -t --net=bridge centos /bin/bash`

- `--net=host`: With this option, Docker does not create a network namespace for the container; instead, the container will network stack with the host. So, we can start the container with this option as follows:

  ```
  $ docker run -i -t  --net=host centos bash
  ```

  We can then run the `ip addr` command within the container as seen here:

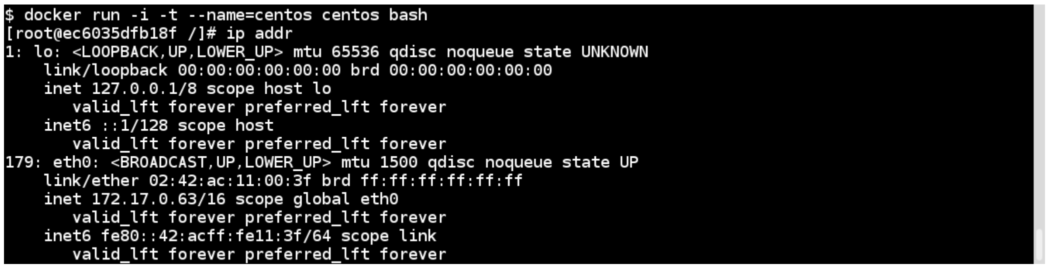


```
$ docker run -i -t --net=host centos /bin/bash
[root@dockerhost /]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    link/ether 90:b1:1c:12:96:c8 brd ff:ff:ff:ff:ff:ff
    inet 10.16.154.221/21 brd 10.16.159.255 scope global dynamic eno1
       valid_lft 63984sec preferred_lft 63984sec
    inet6 fe80::92b1:1cff:fe12:96c8/64 scope link
       valid_lft forever preferred_lft forever
3: eno2: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN qlen 1000
    link/ether 90:b1:1c:12:96:c9 brd ff:ff:ff:ff:ff:ff
4: eno3: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN qlen 1000
    link/ether 90:b1:1c:12:96:ca brd ff:ff:ff:ff:ff:ff
5: eno4: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN qlen 1000
    link/ether 90:b1:1c:12:96:cb brd ff:ff:ff:ff:ff:ff
6: enp5s0f0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    link/ether a0:36:9f:13:d2:14 brd ff:ff:ff:ff:ff:ff
7: enp5s0f1: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN qlen 1000
    link/ether a0:36:9f:13:d2:16 brd ff:ff:ff:ff:ff:ff
8: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::5484:7aff:fefe:9799/64 scope link
       valid_lft forever preferred_lft forever
170: vethad9a4fb: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP
    link/ether 0a:e3:93:df:46:55 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::8e3:93ff:fedf:4655/64 scope link
       valid_lft forever preferred_lft forever
```

  We can see all the network devices attached to the host. An example of using such a configuration is to run the `nginx` reverse proxy within a container to serve the web applications running on the host.

- `--net=container:NAME_or_ID`: With this option, Docker does not create a new network namespace while starting the container but shares it from another container. Let's start the first container and look for its IP address:

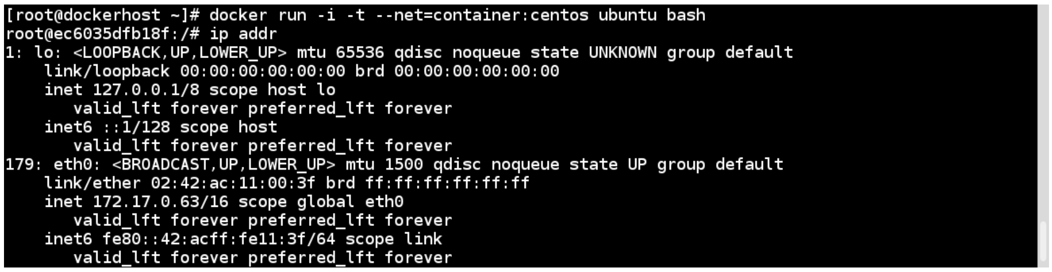
  ```
  $ docker run -i -t --name=centos centos bash
  ```



```
$ docker run -i -t --name=centos centos bash
[root@ec6035dfb18f /]# ip addr
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
179: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:3f brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.63/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:3f/64 scope link
       valid_lft forever preferred_lft forever
```

Now start another as follows:

```
$ docker run -i -t --net=container:centos ubuntu bash
```



```
[root@dockerhost ~]# docker run -i -t --net=container:centos ubuntu bash
root@ec6035dfb18f:/# ip addr
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
179: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:3f brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.63/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:3f/64 scope link
       valid_lft forever preferred_lft forever
```

As we can see, both containers contain the same IP address.

Containers in a Kubernetes (`http://kubernetes.io/`) Pod use this trick to connect with each other. We will revisit this in *Chapter 8*, *Docker Orchestration and Hosting Platforms*.

- `--net=none`: With this option, Docker creates the network namespace inside the container but does not configure networking.

For more information about the different networking we discussed in the preceding section, visit `https://docs.docker.com/articles/networking/#how-docker-networks-a-container`.

From Docker 1.2 onwards, it is also possible to change `/etc/host`, `/etc/hostname`, and `/etc/resolv.conf` on a running container. However, note that these are just used to run a container. If it restarts, we will have to make the changes again.

So far, we have looked at networking on a single host, but in the real world, we would like to connect multiple hosts and have a container from one host to talk to a container from another host. Flannel (`https://github.com/coreos/flannel`), Weave (`https://github.com/weaveworks/weave`), Calio (`http://www.projectcalico.org/getting-started/docker/`), and Socketplane (`http://socketplane.io/`) are some solutions that offer this functionality. Later in this chapter, we will see how to configure Flannel to multihost networking. Socketplane joined Docker Inc in March '15.

Community and Docker are building a **Container Network Model** (**CNM**) with libnetwork (`https://github.com/docker/libnetwork`), which provides a native Go implementation to connect containers. More information on this development can be found at `http://blog.docker.com/2015/04/docker-networking-takes-a-step-in-the-right-direction-2/`.

Accessing containers from outside

Once the container is up, we would like to access it from outside. If you have started the container with the `--net=host` option, then it can be accessed through the Docker host IP. With `--net=none`, you can attach the network interface from the public end or through other complex settings. Let's see what happens in by default—where packets are forwarded from the host network interface to the container.

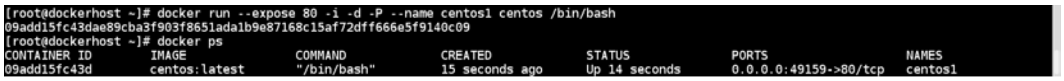
Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it…

1. Let's start a container with the `-P` option:

   ```
   $ docker run --expose 80 -i -d -P --name f20 fedora /bin/bash
   ```



```
[root@dockerhost ~]# docker run --expose 80 -i -d -P --name centos1 centos /bin/bash
09add15fc43dae89cba3f903f8651ada1b9e87168c15af72dff666e5f9140c09
[root@dockerhost ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                   NAMES
09add15fc43d        centos:latest       "/bin/bash"         15 seconds ago      Up 14 seconds       0.0.0.0:49159->80/tcp   centos1
```

   This automatically maps any network port of the container to a random high port of the Docker host between 49000 to 49900.

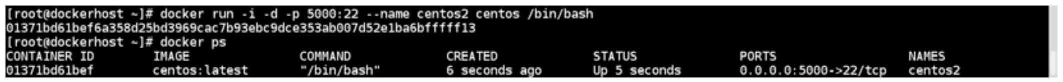
   In the `PORTS` section, we see `0.0.0.0:49159->80/tcp`, which is of the following form:

   ```
   <Host Interface>:<Host Port> -> <Container
   Interface>/<protocol>
   ```

   So, in case any request comes on port `49159` from any interface on the Docker host, the request will be forwarded to port `80` of the `centos1` container.

   We can also map a specific port of the container to the specific port of the host using the `-p` option:

   ```
   $  docker run -i -d -p 5000:22 --name centos2 centos /bin/bash
   ```

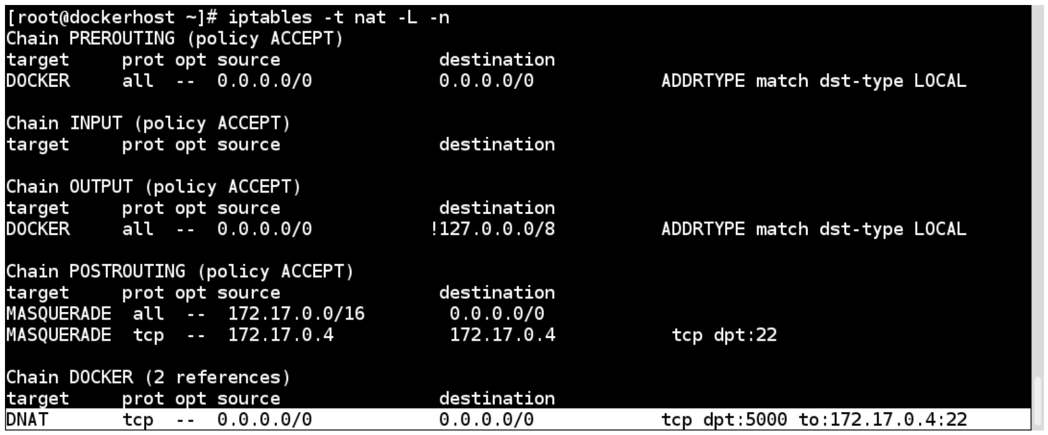


```
[root@dockerhost ~]# docker run -i -d -p 5000:22 --name centos2 centos /bin/bash
01371bd61bef6a358d25bd3969cac7b93ebc9dce353ab007d52e1ba6bfffff13
[root@dockerhost ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                   NAMES
01371bd61bef        centos:latest       "/bin/bash"         6 seconds ago       Up 5 seconds        0.0.0.0:5000->22/tcp    centos2
```

In this case, all requests coming on port `5000` from any interface on the Docker host will be forwarded to port `22` of the `centos2` container.

How it works...

With the default configuration, Docker sets up the firewall rule to forward the connection from the host to the container and enables IP forwarding on the Docker host:



```
[root@dockerhost ~]# 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  --  172.17.0.4           172.17.0.4           tcp dpt:22

Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tcp dpt:5000 to:172.17.0.4:22
```

As we can see from the preceding example, a `DNAT` rule has been set up to forward all traffic on port `5000` of the host to port `22` of the container.

There's more...

By default, with the `-p` option, Docker will forward all the requests coming to any interface to the host. To bind to a specific interface, we can specify something like the following:

```
$ docker run -i -d -p 192.168.1.10:5000:22 --name f20 fedora /bin/bash
```

In this case, only requests coming to port `5000` on the interface that has the IP `192.168.1.10` on the Docker host will be forwarded to port `22` of the `f20` container. To map port `22` of the container to the dynamic port of the host, we can run following command:

```
$ docker run -i -d -p 192.168.1.10::22 --name f20 fedora /bin/bash
```

We can bind multiple ports on containers to ports on hosts as follows:

```
$  docker run -d -i -p 5000:22 -p 8080:80 --name f20 fedora /bin/bash
```

We can look up the public-facing port that is mapped to the container's port as follows:

```
$ docker port f20 80
0.0.0.0:8080
```

To look at all the network settings of a container, we can run the following command:

```
$ docker inspect   -f "{{ .NetworkSettings }}" f20
```

See also

- Networking documentation on the Docker website at `https://docs.docker.com/articles/networking/`.

Managing data in containers

Any uncommitted data or changes in containers get lost as soon as containers are deleted. For example, if you have configured the Docker registry in a container and pushed some images, as soon as the registry container is deleted, all of those images will get lost if you have not committed them. Even if you commit, it is not the best practice. We should try to keep containers as light as possible. The following are two primary ways to manage data with Docker:

- **Data volumes**: From the Docker documentation (`https://docs.docker.com/userguide/dockervolumes/`), a data volume is a specially-designated directory within one or more containers that bypasses the Union filesystem to provide several useful features for persistent or shared data:
  - Volumes are initialized when a container is created. If the container's base image contains data at the specified mount point, that data is copied into the new volume.
  - Data volumes can be shared and reused between containers.
  - Changes to a data volume are made directly.
  - Changes to a data volume will not be included when you update an image.
  - Volumes persist until no containers use them.
- **Data volume containers**: As a volume persists until no container uses it, we can use the volume to share persistent data between containers. So, we can create a named volume container and mount the data to another container.

Getting ready

Make sure that the Docker daemon is running on the host and you can connect through the Docker client.

How to do it...

1. Add a data volume. With the `-v` option with the `docker run` command, we add a data volume to the container:

   ```
   $ docker run -t -d -P -v /data --name f20 fedora /bin/bash
   ```

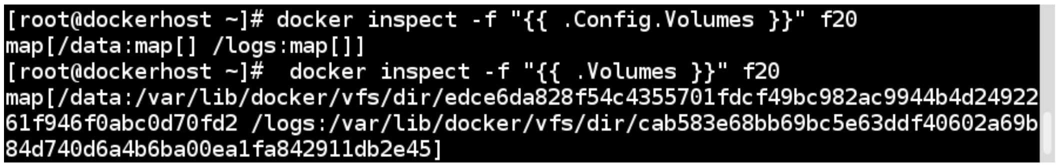
We can have multiple data volumes within a container, which can be created by adding `-v` multiple times:

```
$ docker run -t -d -P -v /data -v /logs --name f20 fedora
/bin/bash
```

The `VOLUME` instruction can be used in a Dockerfile to add data volume as well by adding something similar to `VOLUME ["/data"]`.

We can use the `inspect` command to look at the data volume details of a container:

```
$ docker inspect -f "{{ .Config.Volumes }}" f20
$ docker inspect -f "{{ .Volumes }}" f20
```



```
[root@dockerhost ~]# docker inspect -f "{{ .Config.Volumes }}" f20
map[/data:map[] /logs:map[]]
[root@dockerhost ~]#  docker inspect -f "{{ .Volumes }}" f20
map[/data:/var/lib/docker/vfs/dir/edce6da828f54c4355701fdcf49bc982ac9944b4d24922
61f946f0abc0d70fd2 /logs:/var/lib/docker/vfs/dir/cab583e68bb69bc5e63ddf40602a69b
84d740d6a4b6ba00ea1fa842911db2e45]
```

If the target directory is not there within the container, it will be created.

2. Next, we mount a host directory as a data volume. We can also map a host directory to a data volume with the `-v` option:

   ```
   $ docker run -i -t -v
   /source_on_host:/destination_on_container fedora /bin/bash
   ```

   Consider the following example:

   ```
   $ docker run -i -t -v /srv:/mnt/code fedora /bin/bash
   ```

   This can be very useful in cases such as testing code in different environments, collecting logs in central locations, and so on. We can also map the host directory in read-only mode as follows:

   ```
   $ docker run -i -t -v /srv:/mnt/code:ro fedora /bin/bash
   ```

   We can also mount the entire root filesystem of the host within the container with the following command:

   ```
   $ docker run -i -t -v /:/host:ro fedora /bin/bash
   ```

If the directory on the host (`/srv`) does not exist, then it will be created, given that you have permission to create one. Also, on the Docker host where SELinux is enabled and if the Docker daemon is configured to use SELinux (`docker -d --selinux-enabled`), you will see the `permission denied` error if you try to access files on mounted volumes until you relabel them. To relabel them, use either of the following commands:

```
$ docker run -i -t -v /srv:/mnt/code:z fedora /bin/bash
$ docker run -i -t -v /srv:/mnt/code:Z fedora /bin/bash
```

Please visit *Chapter 9*, *Docker Security*, for more detail.

3. Now, create a data volume container. While sharing the host directory to a container through volume, we are binding the container to a given host, which is not good. Also, the storage in this case is not controlled by Docker. So, in cases when we want data to be persisted even if we update the containers, we can get help from data volume containers. Data volume containers are used to create a volume and nothing else; they do not even run. As the created volume is attached to a container (not running), it cannot be deleted. For example, here's a named data container:

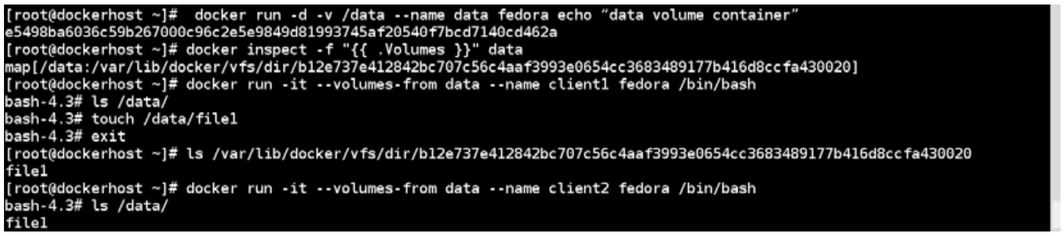
```
$ docker run -d -v /data --name data fedora echo "data volume
container"
```

This will just create a volume that will be mapped to a directory managed by Docker. Now, other containers can mount the volume from the data container using the `--volumes-from` option as follows:

```
$ docker run  -d -i -t --volumes-from data --name client1
fedora /bin/bash
```

We can mount a volume from the data volume container to multiple containers:

```
$ docker run  -d -i -t --volumes-from data --name client2
fedora /bin/bash
```



```
[root@dockerhost ~]#  docker run -d -v /data --name data fedora echo “data volume container”
e5498ba6036c59b267000c96c2e5e9849d81993745af20540f7bcd7140cd462a
[root@dockerhost ~]# docker inspect -f "{{ .Volumes }}" data
map[/data:/var/lib/docker/vfs/dir/b12e737e412842bc707c56c4aaf3993e0654cc3683489177b416d8ccfa430020]
[root@dockerhost ~]# docker run -it --volumes-from data --name client1 fedora /bin/bash
bash-4.3# ls /data/
bash-4.3# touch /data/file1
bash-4.3# exit
[root@dockerhost ~]# ls /var/lib/docker/vfs/dir/b12e737e412842bc707c56c4aaf3993e0654cc3683489177b416d8ccfa430020
file1
[root@dockerhost ~]# docker run -it --volumes-from data --name client2 fedora /bin/bash
bash-4.3# ls /data/
file1
```

We can also use `--volumes-from` multiple times to get the data volumes from multiple containers. We can also create a chain by mounting volumes from the container that mounts from some other container.

How it works...

In case of data volume, when the host directory is not shared, Docker creates a directory within `/var/lib/docker/` and then shares it with other containers.

There's more...

- Volumes are deleted with `-v` flag to `docker rm`, only if no other container is using it. If some other container is using the volume, then the container will be removed (with `docker rm`) but the volume will not be removed.
- In the previous chapter, we saw how to configure the Docker registry, which by default starts with the `dev` flavor. In this registry, uploaded images were saved in the `/tmp/registry` folder within the container we started. We can mount a directory from the host at `/tmp/registry` within the registry container, so whenever we upload an image, it will be saved on the host that is running the Docker registry. So, to start the container, we run following command:

  ```
  $ docker run -v /srv:/tmp/registry -p 5000:5000 registry
  ```

  To push an image, we run the following command:

  ```
  $ docker push registry-host:5000/nkhare/f20
  ```

  After the image is successfully pushed, we can look at the content of the directory that we mounted within the Docker registry. In our case, we should see a directory structure as follows:

  ```
  /srv/
  ├── images
  │   ├──
  3f2fed40e4b0941403cd928b6b94e0fd236dfc54656c00e456747093d10157ac
  │   │   ├── ancestry
  │   │   ├── _checksum
  │   │   ├── json
  │   │   └── layer
  │   ├──
  511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158
  │   │   ├── ancestry
  │   │   ├── _checksum
  │   │   ├── json
  │   │   └── layer
  │   ├──
  53263a18c28e1e54a8d7666cb835e9fa6a4b7b17385d46a7afe55bc5a7c1994c
  ```

```
│   │   ├── ancestry
│   │   ├── _checksum
│   │   ├── json
│   │   └── layer
│   └──
fd241224e9cf32f33a7332346a4f2ea39c4d5087b76392c1ac5490bf2ec55b68
│       ├── ancestry
│       ├── _checksum
│       ├── json
│       └── layer
├── repositories
│   └── nkhare
│       └── f20
│           ├── _index_images
│           ├── json
│           ├── tag_latest
│           └── taglatest_json
```

See also

- The documentation on the Docker website at `https://docs.docker.com/userguide/dockervolumes/`
- `http://container42.com/2013/12/16/persistent-volumes-with-docker-container-as-volume-pattern/`
- `http://container42.com/2014/11/03/docker-indepth-volumes/`

Linking two or more containers

With containerization, we would like to create our stack by running services on different containers and then linking them together. In the previous chapter, we created a WordPress container by putting both a web server and database in the same container. However, we can also put them in different containers and link them together. Container linking creates a parent-child relationship between them, in which the parent can see selected information of its children. Linking relies on the naming of containers.

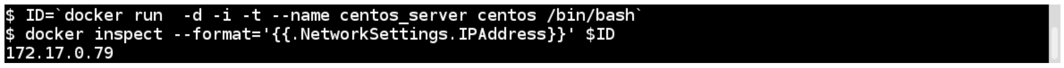
Getting ready

Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it...

1. Create a named container called `centos_server`:

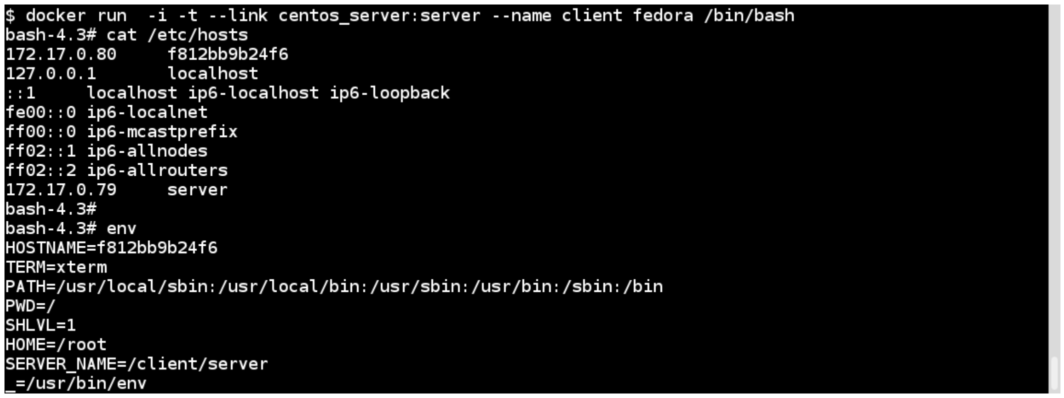
   ```
   $ docker run  -d -i -t --name centos_server centos /bin/bash
   ```



```
$ ID=`docker run  -d -i -t --name centos_server centos /bin/bash`
$ docker inspect --format='{{.NetworkSettings.IPAddress}}' $ID
172.17.0.79
```

2. Now, let's start another container with the `name` client and link it with the `centos_server` container using the `--link` option, which takes the `name:alias` argument. Then look at the `/etc/hosts` file:

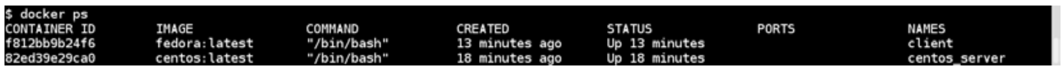
   ```
   $ docker run  -i -t --link centos_server:server --name client
   fedora /bin/bash
   ```



```
$ docker run  -i -t --link centos_server:server --name client fedora /bin/bash
bash-4.3# cat /etc/hosts
172.17.0.80     f812bb9b24f6
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.79     server
bash-4.3#
bash-4.3# env
HOSTNAME=f812bb9b24f6
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
SHLVL=1
HOME=/root
SERVER_NAME=/client/server
_=/usr/bin/env
```

How it works...

In the preceding example, we linked the `centos_server` container to the client container with an alias server. By linking the two containers, an entry of the first container, which is `centos_server` in this case, is added to the `/etc/hosts` file in the client container. Also, an environment variable called `SERVER_NAME` is set within the client to refer to the server.



```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
f812bb9b24f6        fedora:latest       "/bin/bash"         13 minutes ago      Up 13 minutes                           client
82ed39e29ca0        centos:latest       "/bin/bash"         18 minutes ago      Up 18 minutes                           centos_server
```

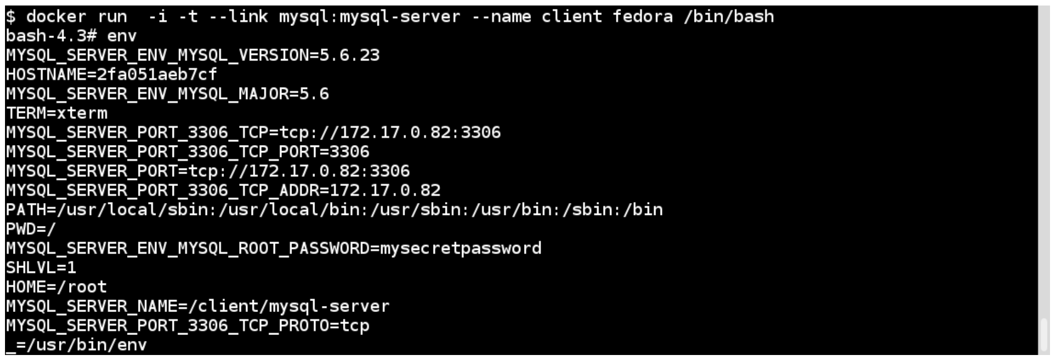
There's more...

Now, let's create a `mysql` container:

```
$ docker run --name mysql -e MYSQL_ROOT_PASSWORD=mysecretpassword -d
mysql
```

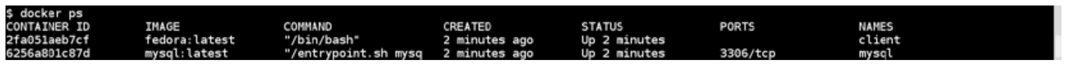
Then, let's link it from a client and check the environment variables:

```
$ docker run  -i -t --link mysql:mysql-server --name client fedora
/bin/bash
```



```
$ docker run  -i -t --link mysql:mysql-server --name client fedora /bin/bash
bash-4.3# env
MYSQL_SERVER_ENV_MYSQL_VERSION=5.6.23
HOSTNAME=2fa051aeb7cf
MYSQL_SERVER_ENV_MYSQL_MAJOR=5.6
TERM=xterm
MYSQL_SERVER_PORT_3306_TCP=tcp://172.17.0.82:3306
MYSQL_SERVER_PORT_3306_TCP_PORT=3306
MYSQL_SERVER_PORT=tcp://172.17.0.82:3306
MYSQL_SERVER_PORT_3306_TCP_ADDR=172.17.0.82
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
MYSQL_SERVER_ENV_MYSQL_ROOT_PASSWORD=mysecretpassword
SHLVL=1
HOME=/root
MYSQL_SERVER_NAME=/client/mysql-server
MYSQL_SERVER_PORT_3306_TCP_PROTO=tcp
_=/usr/bin/env
```

Also, let's look at the `docker ps` output:



```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                CREATED             STATUS              PORTS               NAMES
2fa051aeb7cf        fedora:latest       "/bin/bash"            2 minutes ago       Up 2 minutes                            client
6256a801c87d        mysql:latest        "/entrypoint.sh mysq   2 minutes ago       Up 2 minutes        3306/tcp            mysql
```

If you look closely, we did not specify the `-P` or `-p` options to map ports between two containers while starting the `client` container. Depending on the ports exposed by a container, Docker creates an internal secure tunnel in the containers that links to it. And, to do that, Docker sets environment variables within the linker container. In the preceding case, `mysql` is the linked container and client is the linker container. As the `mysql` container exposes port `3306`, we see corresponding environment variables (`MYSQL_SERVER_*`) within the client container.

> As linking depends on the name of the container, if you want to reuse a name, you must delete the old container.

See also

- Documentation on the Docker website at `https://docs.docker.com/userguide/dockerlinks/`

Developing a LAMP application by linking containers

Let's extend the previous recipe by creating a LAMP application (WordPress) by linking the containers.

Getting ready

To pull MySQL and WordPress images from the Docker registry:

- For MySQL:
    - For image, visit `https://registry.hub.docker.com/_/mysql/`
    - For Dockerfile, visit `https://github.com/docker-library/docker-mysql`
- For WordPress:
    - For image, visit `https://registry.hub.docker.com/_/wordpress/`
    - For Dockerfile, visit `https://github.com/docker-library/wordpress`

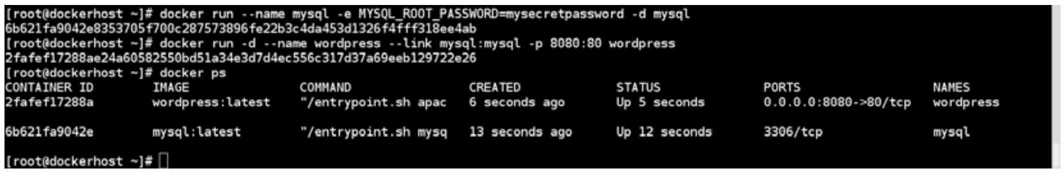
How to do it…

1. First, start a `mysql` container:

   ```
   $ docker run --name mysql -e
   MYSQL_ROOT_PASSWORD=mysecretpassword -d mysql
   ```

2. Then, start the `wordpress` container and link it with the `mysql` container:

   ```
   $ docker run -d --name wordpress --link mysql:mysql -p 8080:80
   wordpress
   ```



```
[root@dockerhost ~]# docker run --name mysql -e MYSQL_ROOT_PASSWORD=mysecretpassword -d mysql
6b621fa9042e8353705f700c287573896fe22b3c4da453d1326f4fff318ee4ab
[root@dockerhost ~]# docker run -d --name wordpress --link mysql:mysql -p 8080:80 wordpress
2fafef17288ae24a60582550bd51a34e3d7d4ec556c317d37a69eeb129722e26
[root@dockerhost ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                CREATED             STATUS              PORTS                  NAMES
2fafef17288a        wordpress:latest    "/entrypoint.sh apac   6 seconds ago       Up 5 seconds        0.0.0.0:8080->80/tcp   wordpress

6b621fa9042e        mysql:latest        "/entrypoint.sh mysq   13 seconds ago      Up 12 seconds       3306/tcp               mysql

[root@dockerhost ~]# 
```

We have the Docker host's `8080` port to container `80` port, so we can connect WordPress by accessing the `8080` port on the Docker host with the `http://<DockerHost>:8080` URL.

How it works…

A link is created between the `wordpress` and `mysql` containers. Whenever the `wordpress` container gets a DB request, it passes it on to the `mysql` container and gets the results. Look at the preceding recipe for more details.

Networking of multihost containers with Flannel

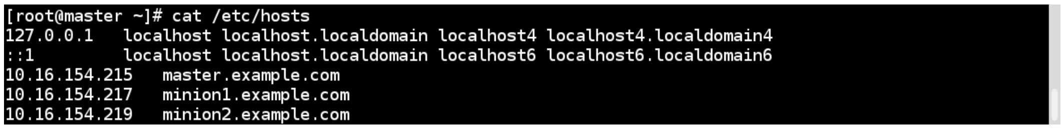
In this recipe, we'll use Flannel (`https://github.com/coreos/flannel`) to set up multihost container networking. Flannel is a generic overlay network that can be used as an alternative to **Software Defined Networking** (**SDN**). It is an IP-based solution that uses **Virtual Extensible LAN** (**VXLAN**), in which unique IP addresses are assigned to each container on a unique subnet given to the host that is running that container. So, in this kind of a solution, a different subnet and communication occurs within each host in the cluster, using the overlay network. Flannel uses the `etcd` service (`https://github.com/coreos/etcd`) for the key-value store.

Getting ready

For this recipe, we will require three VMs or physical machines with Fedora 21 installed.

How to do it…

1. Let's call one machine/VM `master` and other two `minion1` and `minion2`. According to your system's IP addresses, update the `/etc/hosts` file as follows:



```
[root@master ~]# cat /etc/hosts
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
10.16.154.215   master.example.com
10.16.154.217   minion1.example.com
10.16.154.219   minion2.example.com
```

2. Install `etcd`, `Flannel`, and `Docker` on all the systems we set up:

   `$ yum install -y etcd flannel docker`

3. Modify the value of the `ETCD_LISTEN_CLIENT_URLS` to `http://master.example.com:4001` in the `/etc/etcd/etcd.conf` file as follows:

   ```
   ETCD_LISTEN_CLIENT_URLS="http://master.example.com:4001"
   ```

4. In the master, start the `etcd` service and check its status:

   ```
   $ systemctl start etcd
   $ systemctl enable etcd
   $ systemctl status etcd
   ```

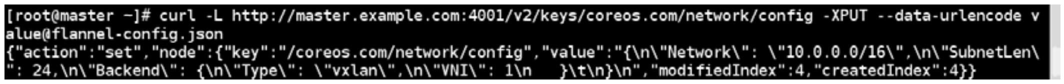
5. In the master, create a file called `flannel-config.json` with the following content:

   ```
   {
   "Network": "10.0.0.0/16",
   "SubnetLen": 24,
   ```

```
"Backend": {
"Type": "vxlan",
"VNI": 1
   }
}
```

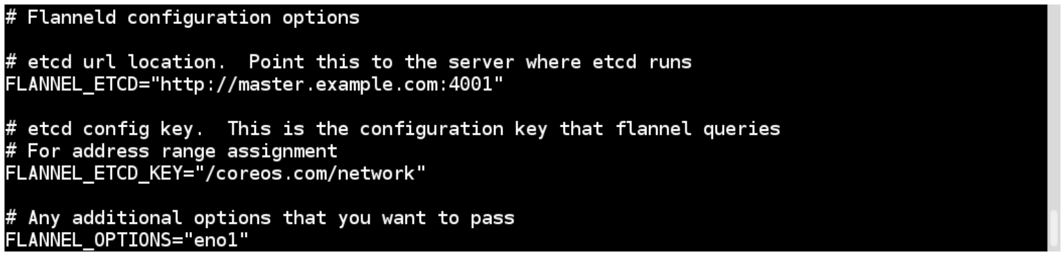
6. Upload the preceding configuration file to `etcd` using `config` as the key:

   ```
   $ curl -L
   http://master.example.com:4001/v2/keys/coreos.com/network/conf
   ig -XPUT --data-urlencode value@flannel-config.json
   ```



```
[root@master ~]# curl -L http://master.example.com:4001/v2/keys/coreos.com/network/config -XPUT --data-urlencode v
alue@flannel-config.json
{"action":"set","node":{"key":"/coreos.com/network/config","value":"{\n\"Network\": \"10.0.0.0/16\",\n\"SubnetLen\
": 24,\n\"Backend\": {\n\"Type\": \"vxlan\",\n\"VNI\": 1\n   }\t\n}\n","modifiedIndex":4,"createdIndex":4}}
```

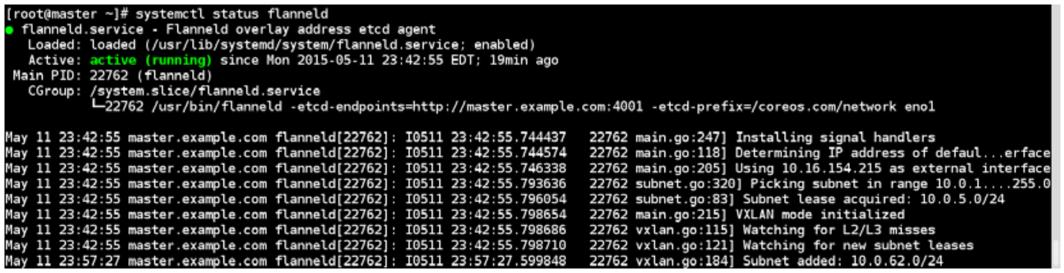
7. In master, update `FLANNEL_OPTIONS` in the `/etc/sysconfig/flanneld` file to reflect the interface of the system. Also, update `FLANNEL_ETCD` to use hostname instead of the 127.0.0.1:4001 address.



```
# Flanneld configuration options

# etcd url location.  Point this to the server where etcd runs
FLANNEL_ETCD="http://master.example.com:4001"

# etcd config key.  This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_KEY="/coreos.com/network"

# Any additional options that you want to pass
FLANNEL_OPTIONS="eno1"
```

8. To enable and start the `flanneld` service in master:

   ```
   $ systemctl enable flanneld
   $ systemctl start flanneld
   $ systemctl status flanneld
   ```

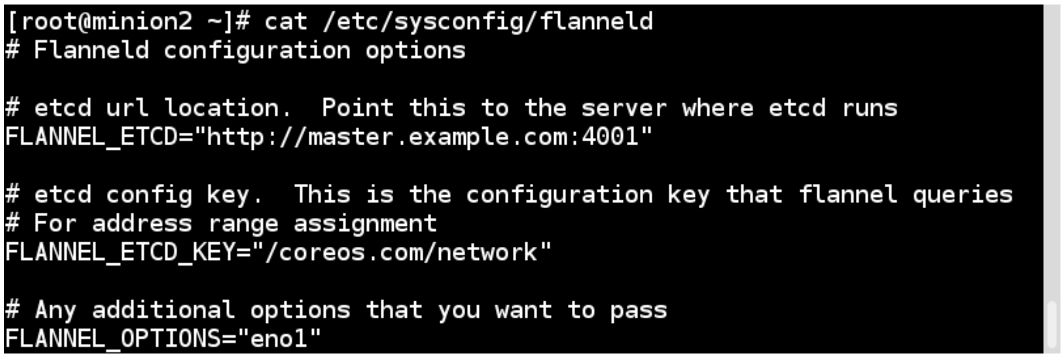


```
[root@master ~]# systemctl status flanneld
● flanneld.service - Flanneld overlay address etcd agent
   Loaded: loaded (/usr/lib/systemd/system/flanneld.service; enabled)
   Active: active (running) since Mon 2015-05-11 23:42:55 EDT; 19min ago
 Main PID: 22762 (flanneld)
   CGroup: /system.slice/flanneld.service
           └─22762 /usr/bin/flanneld -etcd-endpoints=http://master.example.com:4001 -etcd-prefix=/coreos.com/network eno1

May 11 23:42:55 master.example.com flanneld[22762]: I0511 23:42:55.744437   22762 main.go:247] Installing signal handlers
May 11 23:42:55 master.example.com flanneld[22762]: I0511 23:42:55.744574   22762 main.go:118] Determining IP address of defaul...erface
May 11 23:42:55 master.example.com flanneld[22762]: I0511 23:42:55.746338   22762 main.go:205] Using 10.16.154.215 as external interface
May 11 23:42:55 master.example.com flanneld[22762]: I0511 23:42:55.793636   22762 subnet.go:320] Picking subnet in range 10.0.1....255.0
May 11 23:42:55 master.example.com flanneld[22762]: I0511 23:42:55.796054   22762 subnet.go:83] Subnet lease acquired: 10.0.5.0/24
May 11 23:42:55 master.example.com flanneld[22762]: I0511 23:42:55.798654   22762 main.go:215] VXLAN mode initialized
May 11 23:42:55 master.example.com flanneld[22762]: I0511 23:42:55.798686   22762 vxlan.go:115] Watching for L2/L3 misses
May 11 23:42:55 master.example.com flanneld[22762]: I0511 23:42:55.798710   22762 vxlan.go:121] Watching for new subnet leases
May 11 23:57:27 master.example.com flanneld[22762]: I0511 23:57:27.599848   22762 vxlan.go:184] Subnet added: 10.0.62.0/24
```

9. From the minion systems, check the connectivity to master for `etcd`:

   ```
   [root@minion1 ~]#  curl -L
   http://master.example.com:4001/v2/keys/coreos.com/network/conf
   ig
   ```

10. Update the `/etc/sysconfig/flanneld` file in both minions to point to the `etcd` server running in master and update `FLANNEL_OPTIONS` to reflect the interface of the minion host:



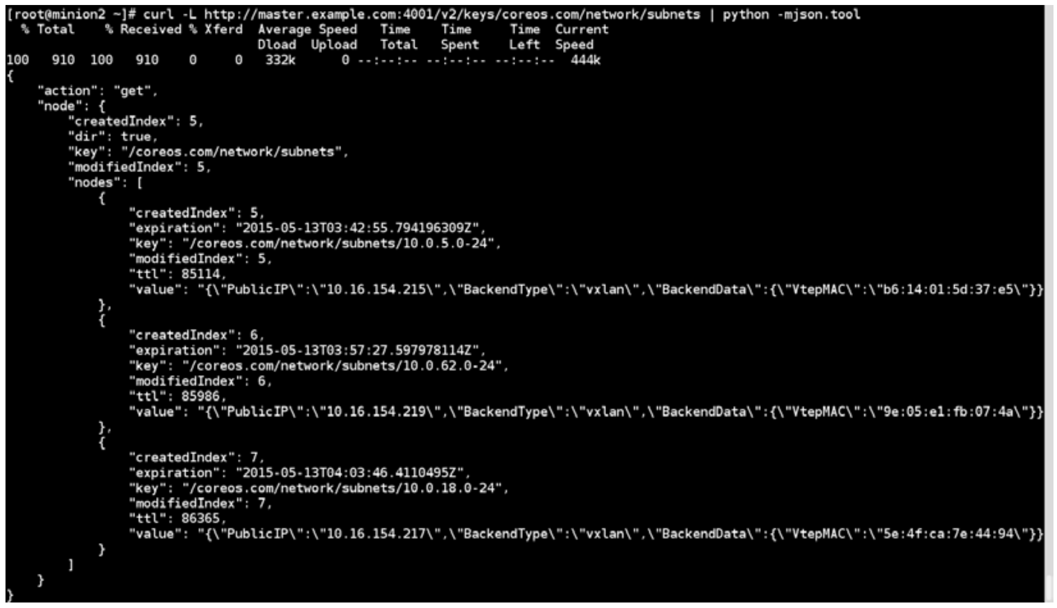
```
[root@minion2 ~]# cat /etc/sysconfig/flanneld
# Flanneld configuration options

# etcd url location.  Point this to the server where etcd runs
FLANNEL_ETCD="http://master.example.com:4001"

# etcd config key.  This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_KEY="/coreos.com/network"

# Any additional options that you want to pass
FLANNEL_OPTIONS="eno1"
```

11. To enable and start the `flanneld` service in both the minions:

    ```
    $ systemctl enable flanneld
    $ systemctl start flanneld
    $ systemctl status flanneld
    ```

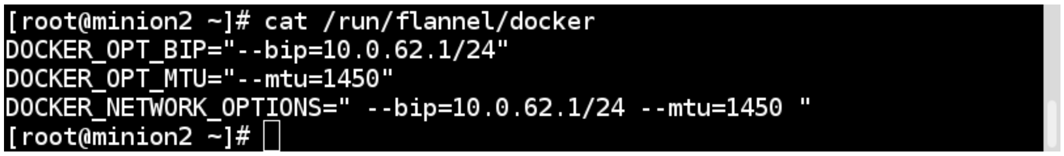
12. In any of the hosts in the cluster, run the following command:

    ```
    $ curl -L
    http://master.example.com:4001/v2/keys/coreos.com/network/subn
    ets | python -mjson.tool
    ```



```
[root@minion2 ~]# curl -L http://master.example.com:4001/v2/keys/coreos.com/network/subnets | python -mjson.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   910  100   910    0     0   332k      0 --:--:-- --:--:-- --:--:--  444k
{
    "action": "get",
    "node": {
        "createdIndex": 5,
        "dir": true,
        "key": "/coreos.com/network/subnets",
        "modifiedIndex": 5,
        "nodes": [
            {
                "createdIndex": 5,
                "expiration": "2015-05-13T03:42:55.794196309Z",
                "key": "/coreos.com/network/subnets/10.0.5.0-24",
                "modifiedIndex": 5,
                "ttl": 85114,
                "value": "{\"PublicIP\":\"10.16.154.215\",\"BackendType\":\"vxlan\",\"BackendData\":{\"VtepMAC\":\"b6:14:01:5d:37:e5\"}}
            },
            {
                "createdIndex": 6,
                "expiration": "2015-05-13T03:57:27.597978114Z",
                "key": "/coreos.com/network/subnets/10.0.62.0-24",
                "modifiedIndex": 6,
                "ttl": 85986,
                "value": "{\"PublicIP\":\"10.16.154.219\",\"BackendType\":\"vxlan\",\"BackendData\":{\"VtepMAC\":\"9e:05:e1:fb:07:4a\"}}
            },
            {
                "createdIndex": 7,
                "expiration": "2015-05-13T04:03:46.4110495Z",
                "key": "/coreos.com/network/subnets/10.0.18.0-24",
                "modifiedIndex": 7,
                "ttl": 86365,
                "value": "{\"PublicIP\":\"10.16.154.217\",\"BackendType\":\"vxlan\",\"BackendData\":{\"VtepMAC\":\"5e:4f:ca:7e:44:94\"}}
            }
        ]
    }
}
```

This tells us the number of hosts in the network and the subnets associated (look at the key for each node) with them. We can associate the subnet with the MAC address on the hosts. On each host, the `/run/flannel/docker` and `/run/flannel/subnet.env` files are populated with subnet information. For instance, in `minion2`, you would see something like the following:

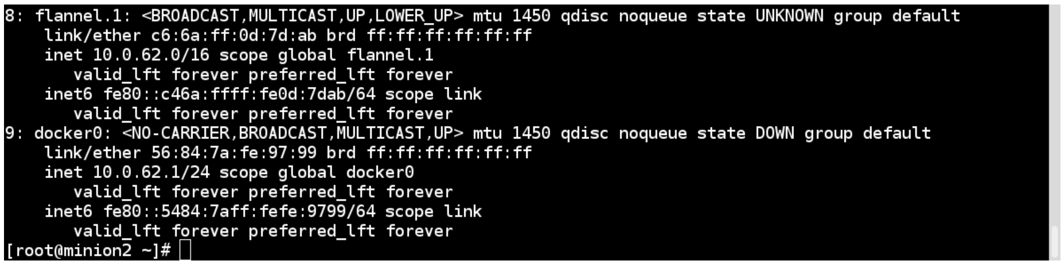


```
[root@minion2 ~]# cat /run/flannel/docker
DOCKER_OPT_BIP="--bip=10.0.62.1/24"
DOCKER_OPT_MTU="--mtu=1450"
DOCKER_NETWORK_OPTIONS=" --bip=10.0.62.1/24 --mtu=1450 "
[root@minion2 ~]# 
```

13. To restart the Docker daemon in all the hosts:

    `$ systemctl restart docker`

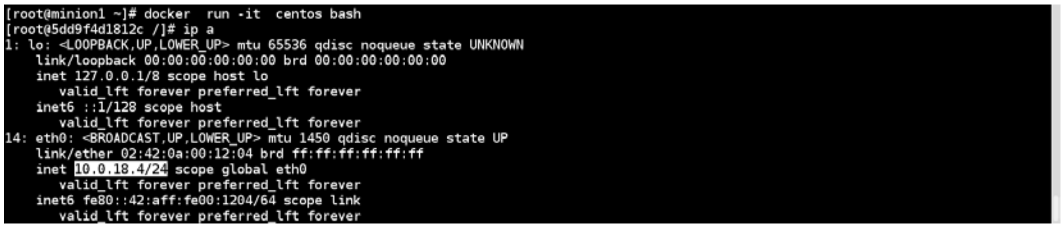
Then, look at the IP address of the `docker0` and `flannel.1` interfaces. In `minion2`, it looks like the following:



```
8: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN group default
    link/ether c6:6a:ff:0d:7d:ab brd ff:ff:ff:ff:ff:ff
    inet 10.0.62.0/16 scope global flannel.1
       valid_lft forever preferred_lft forever
    inet6 fe80::c46a:ffff:fe0d:7dab/64 scope link
       valid_lft forever preferred_lft forever
9: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1450 qdisc noqueue state DOWN group default
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 10.0.62.1/24 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::5484:7aff:fefe:9799/64 scope link
       valid_lft forever preferred_lft forever
[root@minion2 ~]#
```

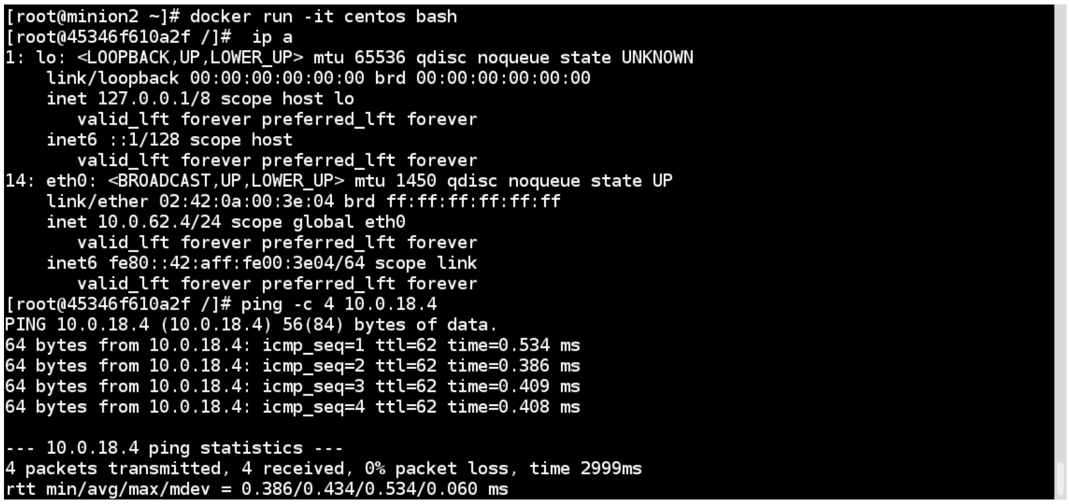
We can see that the `docker0` interface got the IP from the same subnet as the `flannel.1` interface, which is used to route all traffic.

14. We are all set to spawn two containers in any of the hosts and they should be able to communicate. Let's create one container in `minion1` and get its IP address:



```
[root@minion1 ~]# docker  run -it  centos bash
[root@5dd9f4d1812c /]# ip a
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
14: eth0: <BROADCAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP
    link/ether 02:42:0a:00:12:04 brd ff:ff:ff:ff:ff:ff
    inet 10.0.18.4/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:aff:fe00:1204/64 scope link
       valid_lft forever preferred_lft forever
```

15. Now create another container in `minion2` and ping the container running in `minion1` as follows:



```
[root@minion2 ~]# docker run -it centos bash
[root@45346f610a2f /]#  ip a
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
14: eth0: <BROADCAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP
    link/ether 02:42:0a:00:3e:04 brd ff:ff:ff:ff:ff:ff
    inet 10.0.62.4/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:aff:fe00:3e04/64 scope link
       valid_lft forever preferred_lft forever
[root@45346f610a2f /]# ping -c 4 10.0.18.4
PING 10.0.18.4 (10.0.18.4) 56(84) bytes of data.
64 bytes from 10.0.18.4: icmp_seq=1 ttl=62 time=0.534 ms
64 bytes from 10.0.18.4: icmp_seq=2 ttl=62 time=0.386 ms
64 bytes from 10.0.18.4: icmp_seq=3 ttl=62 time=0.409 ms
64 bytes from 10.0.18.4: icmp_seq=4 ttl=62 time=0.408 ms

--- 10.0.18.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2999ms
rtt min/avg/max/mdev = 0.386/0.434/0.534/0.060 ms
```

How it works...

With Flannel, we first configure the overlay with the `10.0.0.0/16` network. Then, each host picks up a random `/24` network; for instance, in our case, `minion2` gets the `10.0.62.0/24` subnet and so on. Once configured, a container in the host gets the IP address from that chosen subnet. Flannel encapsulates the packets and sends it to remote hosts using UDP.

Also, during installation, Flannel copies a configuration file (`flannel.conf`) within `/usr/lib/systemd/system/docker.service.d/`, which Docker uses to configure itself.

See also

- The diagram from Flannel GitHub to help you understand the theory of operations at `https://github.com/coreos/flannel/blob/master/packet-01.png`
- The documentation on the CoreOS website at `https://coreos.com/blog/introducing-rudder/`
- Scott Collier's blog post about setting Flannel on Fedora at `http://www.colliernotes.com/2015/01/flannel-and-docker-on-fedora-getting.html`

Assigning IPv6 addresses to containers

By default, Docker assigns IPv4 addresses to containers. With Docker 1.5, a feature has been added to support IPv6 addresses.

Getting ready

Make sure the Docker daemon (version 1.5 and above) is running on the host and you can connect through the Docker client.

How to do it...

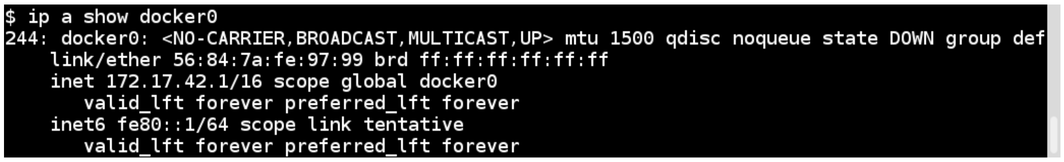
1. To start the Docker daemon with the `--ipv6` option, we can add this option in the daemon's configuration file (`/etc/sysconfig/docker` on Fedora) as follows:

   ```
   OPTIONS='--selinux-enabled --ipv6'
   ```

   Alternatively, if we start Docker in daemon mode, then we can start it as follows:

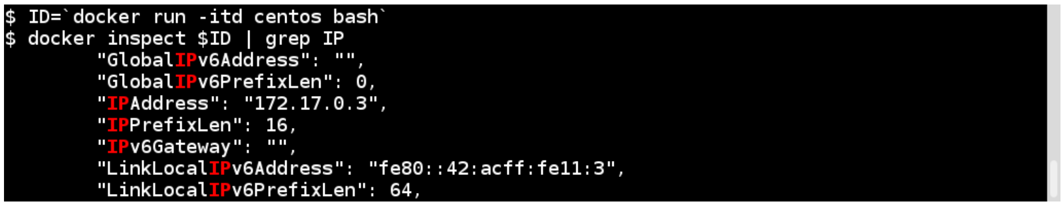
   ```
   $ docker -d --ipv6
   ```

By running either of these commands, Docker will set up the `docker0` bridge with the IPv6 local link address `fe80::1`.



```
$ ip a show docker0
244: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group def
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::1/64 scope link tentative
       valid_lft forever preferred_lft forever
```

2. Let's start the container and look for the IP addresses assigned to it:



```
$ ID=`docker run -itd centos bash`
$ docker inspect $ID | grep IP
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "IPAddress": "172.17.0.3",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "LinkLocalIPv6Address": "fe80::42:acff:fe11:3",
        "LinkLocalIPv6PrefixLen": 64,
```

As we can see, both the IPv4 and local link IPv6 addresses are available to the container. To ping on the IPv6 address of a container from the host machine, run the following command:

```
$ ping6 -I docker0 fe80::42:acff:fe11:3
```

To ping the `docker0` bridge from the container, run the following command:

```
[root@c7562c38bd0f /]# ping6 -I eth0 fe80::1
```

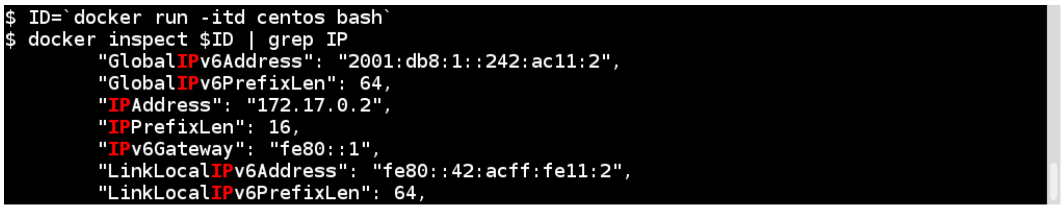
How it works...

Docker configures the `docker0` bridge to assign IPv6 addresses to containers, which enables us to use the IPv6 address of containers.

There's more...

By default, containers will get the link-local address. To assign them a globally routable address, you can pass the IPv6 subnet pick address with `--fixed-cidr-v6` as follows:

```
$ docker -d --ipv6 --fixed-cidr-v6="2001:db8:1::/64"
```



```
$ ID=`docker run -itd centos bash`
$ docker inspect $ID | grep IP
        "GlobalIPv6Address": "2001:db8:1::242:ac11:2",
        "GlobalIPv6PrefixLen": 64,
        "IPAddress": "172.17.0.2",
        "IPPrefixLen": 16,
        "IPv6Gateway": "fe80::1",
        "LinkLocalIPv6Address": "fe80::42:acff:fe11:2",
        "LinkLocalIPv6PrefixLen": 64,
```

From here, we can see that the globally routable address (GlobalIPv6Address) is now being set.

See also

- The Docker 1.5 release notes at `https://blog.docker.com/2015/02/docker-1-5-ipv6-support-read-only-containers-stats-named-dockerfiles-and-more/`.
- The documentation on the Docker website at `http://docs.docker.com/v1.5/articles/networking/#ipv6`.
- You might need to delete the exiting `docker0` bridge on the host before setting the IPv6 option. To understand how to do so, visit `http://docs.docker.com/v1.5/articles/networking/#customizing-docker0`.

5

Docker Use Cases

In this chapter, we will cover the following recipes:

- Testing with Docker
- Doing CI/CD with Shippable and Red Hat OpenShift
- Doing CI/CD with Drone
- Setting up PaaS with OpenShift Origin
- Building and deploying an app on OpenShift v3 from the source code
- Configuring Docker as a hypervisor driver for Openstack

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 other 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 servers and link them, as we saw in the previous chapter.
- **Collaboration and distribution**: GitHub is one of the best examples of collaborating and distributing the code. Similarly, Docker provides features such as Dockerfile, registry, and import/export to share and collaborate with others. We have covered all this in earlier chapters.

- **Continuous Integration** (**CI**): The following definition on Martin Fowler's 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. This article is a quick overview of Continuous Integration summarizing the technique and its current usage."

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 Drone. We'll see how Shippable and Drone can be used for CI work later in this chapter. Shippable is not a hosted solution but Drone is, which can give you better control. 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 customers, the cloud and other environments without any manual work. In this chapter, we'll see how we can automatically deploy an app on Red Hat OpenShift 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, Tsuru, and so on. Later in this chapter, we'll see how to set up PaaS using OpenShift Origin (`https://www.openshift.com/products/origin`).

Testing with Docker

While doing the development or QA, it will be helpful if we can check our code against different environments. For example, we may wish to check our Python code between different versions of Python or on different distributions such as Fedora, Ubuntu, CentOS, and so on. For this recipe, we will pick up sample code from Flask's GitHub repository, which is a microframework for Python (`http://flask.pocoo.org/`). 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 to have one container with Python 2.7 and other with Python 3.3. We'll then use a sample Python test code to run against each container.

Getting ready

- As we are going to use example code from Flask's GitHub repository, let's clone it:

  ```
  $ git clone https://github.com/mitsuhiko/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
  RUN pip install pytest
  WORKDIR /test
  CMD ["/usr/local/bin/py.test"]
  ```

- To build the `python2.7test` image, run the following command:

  ```
  $ docker build -t python2.7test - < /tmp/Dockerfile_2.7
  ```

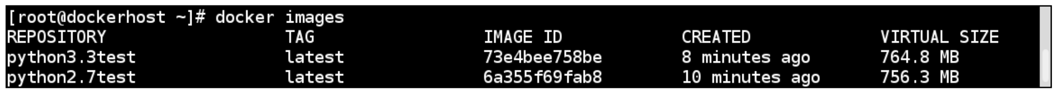
- Similarly, create a Dockerfile with `python:3.3` as the base image and build the `python3.3test` image:

  ```
  $ cat /tmp/Dockerfile_3.3
  FROM python:3.3
  RUN pip install flask
  RUN pip install pytest
  WORKDIR /test
  CMD ["/usr/local/bin/py.test"]
  ```

- To build the image, run the following command:

  ```
  $ docker build -t python3.3test  - < /tmp/Dockerfile_3.3
  ```

Make sure both the images are created.



```
[root@dockerhost ~]# docker images
REPOSITORY           TAG              IMAGE ID          CREATED           VIRTUAL SIZE
python3.3test        latest           73e4bee758be      8 minutes ago     764.8 MB
python2.7test        latest           6a355f69fab8      10 minutes ago    756.3 MB
```

How to do it...

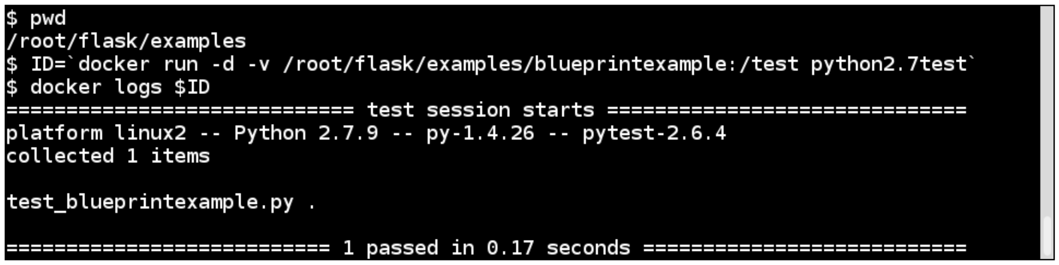
Now, using Docker's volume feature, we will mount the external directory that contains the source code and test cases. To test with Python 2.7, do the following:

1. Go to the directory that contains the Flask examples:

   ```
   $ cd /tmp/flask/examples/
   ```

2. Start a container with the `python2.7` test image and mount `blueprintexample` under `/test`:

   ```
   $ docker run -d -v `pwd`/blueprintexample:/test python2.7test
   ```



```
$ pwd
/root/flask/examples
$ ID=`docker run -d -v /root/flask/examples/blueprintexample:/test python2.7test`
$ docker logs $ID
=========================== test session starts ============================
platform linux2 -- Python 2.7.9 -- py-1.4.26 -- pytest-2.6.4
collected 1 items

test_blueprintexample.py .

========================== 1 passed in 0.17 seconds ==========================
```

3. Similarly, to test with Python 3.3, run the following command:

   ```
   $ docker run -d -v `pwd`/blueprintexample:/test python3.3test
   ```

4. 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 within the container as follows:

   ```
   $ docker run -d -v `pwd`/blueprintexample:/test:z
   python2.7test
   ```

[For more details on SELinux, please look at *Chapter 9*, *Docker Security*.]

How it works...

As you can see from the Dockerfile, before running CMD, which runs the `py.test` binary, we change our working directory to `/test`. And while starting the container, we mount our source code to `/test`. So, as soon as the container starts, it will run the `py.test` binary and run 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, 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://wiki.jenkins-ci.org/display/JENKINS/Docker+Plugin`.

Doing CI/CD with Shippable and Red Hat OpenShift

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 the CI/CD environment. In this recipe, we'll see how we can use Shippable (`http://www.shippable.com/`) to perform CI/CD and deploy it on Red Hat's OpenShift environment (`https://openshift.redhat.com`).

Shippable is a SaaS platform that lets you easily add Continuous Integration/Deployment to your GitHub and Bitbucket(Git) repositories, which 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 12.04 LTS and Ubuntu 14.04. 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 `yml` file called `shippable.yml`, which you have to provide in your source code repo. The `yml` file contains the following instructions:

- `build_image`: This is a Docker image to use to build
- `language`: This will show the programming language
    - `versions`: You can specify different versions of the language to get tested in a single build instruction.
- `before_install`: These are the instructions before running the build
- `script`: This is a binary/script to run the test
- `after_success`: These are instructions after the build succeeds; this is used to perform deployment on PaaS such as Heroku, Amazon Elastic Beanstalk, AWS OpsWorks, Google App Engine, Red Hat OpenShift, and others.

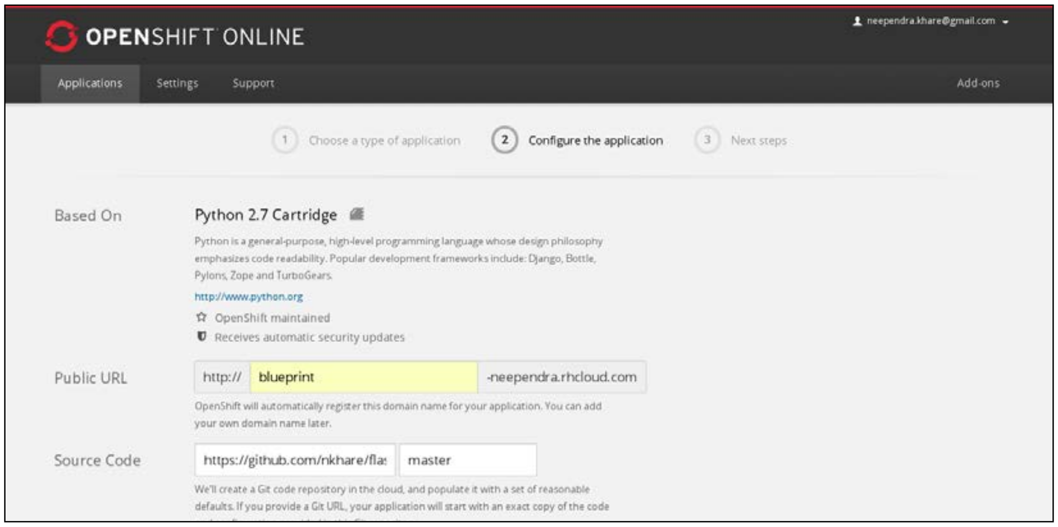
Red Hat's OpenShift is a PaaS platform to host your application. Currently, it uses non-Docker based container technology to host the application, but the next version of OpenShift (`https://github.com/openshift/origin`) is being built on Kubernetes and Docker. This tells us the pace at which Docker is being adopted in the enterprise world. We'll see how to set up OpenShift v3 later in this chapter.

For this recipe, we will use the same example code we used in the previous recipe, to first test on Shippable and then deploy it on OpenShift.

Getting ready

1. Create an account on Shippable (`https://www.shippable.com/`).
2. Fork the flask example from `https://github.com/openshift/flask-example`.

3. Create an app on OpenShift for the forked repository with the following steps:
    1. Create an account (`https://www.openshift.com/app/account/new`) on OpenShift and log in.
    2. Select **Python 2.7 Cartridge** for the application.
    3. Update the **Public URL** section you want. In the **Source Code** section, provide the URL of our forked repo. For this example, I have put down `blueprint` and `https://github.com/nkhare/flask-example` respectively:


OPENSHIFT ONLINE
neependra.khare@gmail.com
Applications
Settings
Support
Add-ons
1 Choose a type of application
2 Configure the application
3 Next steps

Based On: Python 2.7 Cartridge

Python is a general-purpose, high-level programming language whose design philosophy emphasizes code readability. Popular development frameworks include: Django, Bottle, Pylons, Zope and TurboGears.

http://www.python.org

OpenShift maintained

Receives automatic security updates

Public URL: http:// blueprint -neependra.rhcloud.com

OpenShift will automatically register this domain name for your application. You can add your own domain name later.

Source Code: https://github.com/nkhare/fla: master

We'll create a Git code repository in the cloud, and populate it with a set of reasonable defaults. If you provide a Git URL, your application will start with an exact copy of the code

    4. Click on **Create Application** to create the new app. Once created, you should be able to access the Public URL we mentioned in the previous step.

Once the app is created, OpenShift provides a way to manage/update the source code for this app in the `Making code changes` section. As we want to deploy the app using Shippable, we don't have to follow those instructions.

4. Clone the forked repository on the local system:

```
$ git clone git@github.com:nkhare/flask-example.git
```

5. Let's use the same blueprint example that we used earlier. To do so, follow these instructions:
    1. Clone the flask repository:

    ```
    $ git clone https://github.com/mitsuhiko/flask.git
    ```

    2. Copy the blueprint example:

    ```
    $ cp -Rv flask/examples/blueprintexample/* flask-example/
    wsgi/
    ```

6. Update the `flask-example/wsgi/application` file to import the `app` module from the `blueprintexample` module. So, the last line in the `flask-example/wsgi/application` file looks like the following:

   ```
   from blueprintexample import app as application
   ```

7. Add the `requirements.txt` file with the following contents at the top level of the flask-example repository:

   ```
   flask
   pytest
   ```

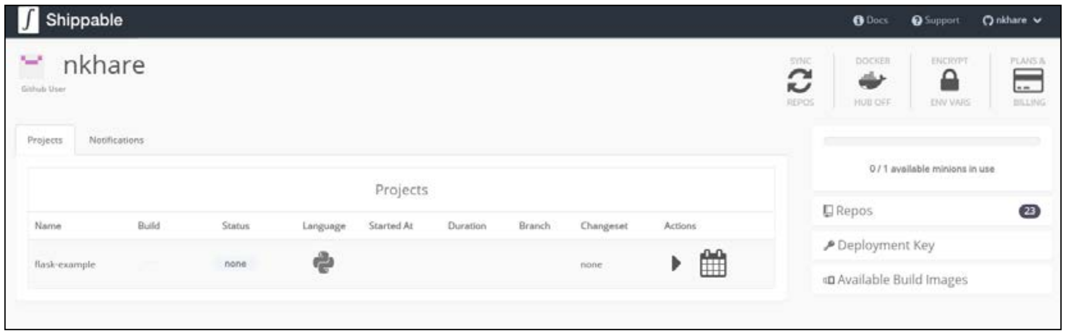
8. Add the `shippable.yml` file with following content:

   ```
   language: python

   python:
     - 2.6
     - 2.7

   install:
     - pip install -r requirements.txt

   # Make folders for the reports
   before_script:
     - mkdir -p shippable/testresults
     - mkdir -p shippable/codecoverage

   script:
     - py.test

   archive: true
   ```

9. Commit the code and push it in your forked repository.

How to do it...

1. Log in to Shippable.
2. After logging in, click on **SYNC ACCOUNT** to get your forked repository listed, if it has not already been listed. Find and enable the repo that you want to build and run tests. For this example, I chose `flask-example` from my GitHub repos. After enabling it, you should see something like the following:



Projects

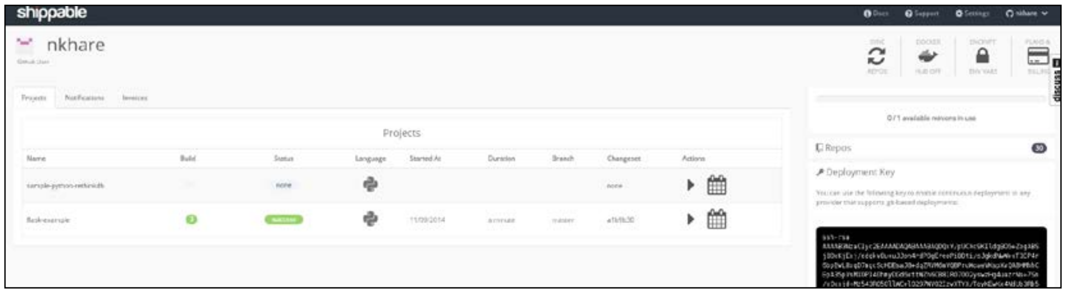
Name	Build	Status	Language	Started At	Duration	Branch	Changeset	Actions
flask-example		none					none	

3. Click on the play button and select branch to build. For this recipe, I chose master:

If the build is successful, then you will see the success icon.

Next time 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 OpenShift, let's follow the instructions provided on the Shippable website (`http://docs.shippable.com/deployment/openshift/`):

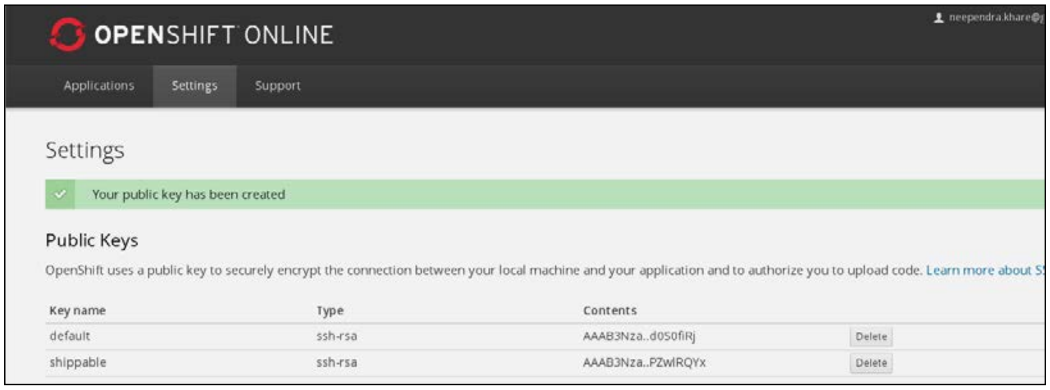
1. Get the deployment key from your Shippable dashboard (located on the right-hand side, below **Repos**):



Projects

Name	Build	Status	Language	Started At	Duration	Branch	Changeset	Actions
sample-python-nekhare		none					none	
flask-example	3	success		11/09/2014	a minute	master	a1b9b30	

2. Copy it under the (`https://openshift.redhat.com/app/console/settings`) **Settings** | **Public Keys** section on OpenShift as follows:



Settings

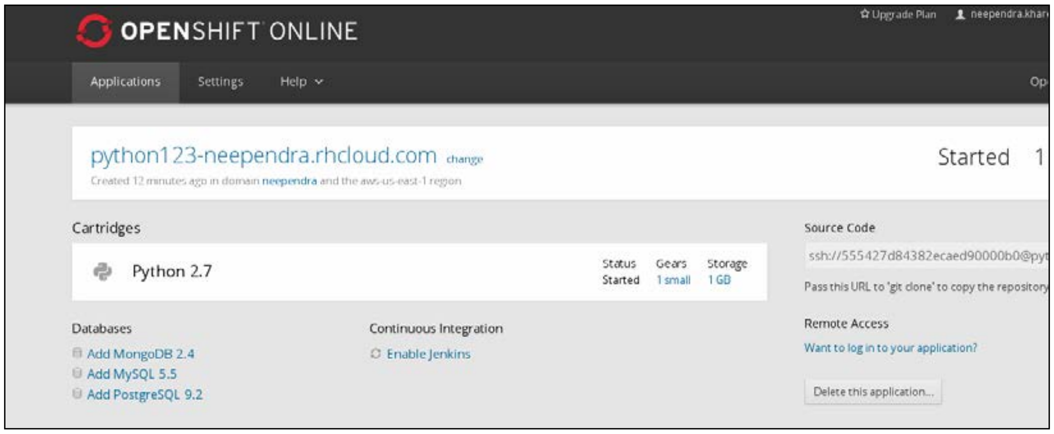
Your public key has been created

Public Keys

OpenShift uses a public key to securely encrypt the connection between your local machine and your application and to authorize you to upload code. Learn more about S

Key name	Type	Contents	
default	ssh-rsa	AAAB3Nza..d050fiRj	Delete
shippable	ssh-rsa	AAAB3Nza..PZwIRQYx	Delete

3. Get the **Source Code** repository link from the OpenShift application page, which will be used as `OPNESHIFT_REPO` in the next step:



python123-neependra.rhcloud.com

Created 12 minutes ago in domain neependra and the aws-us-east-1 region

Started 1

Cartridges

Python 2.7 — Status: Started, Gears: 1 small, Storage: 1 GB

Databases

Continuous Integration

Source Code

ssh://555427d84382ecaed90000b0@pyt

Pass this URL to 'git clone' to copy the repository

Remote Access

Want to log in to your application?

4. After the deployment key is installed, update the `shippable.yml` file as follows:

```
env:
  global:
    - OPENSHIFT_REPO=ssh://545ea4964382ec337f000009@blueprint-
neependra.rhcloud.com/~/git/blueprint.git

language: python

python:
  - 2.6
  - 2.7
```

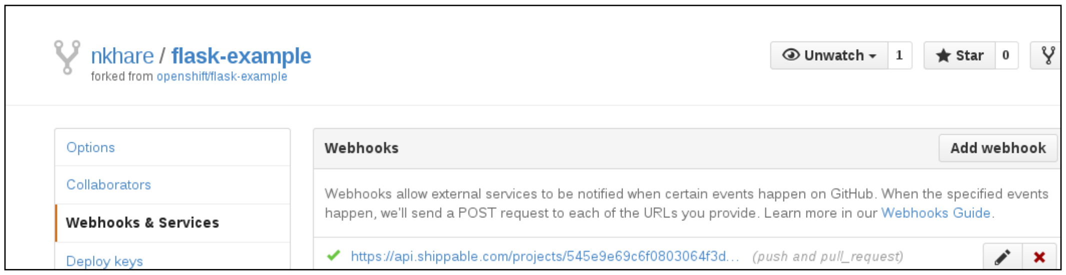
```
install:
  - pip install -r requirements.txt

# Make folders for the reports
before_script:
  - mkdir -p shippable/testresults
  - mkdir -p shippable/codecoverage
  - git remote -v | grep ^openshift || git remote add openshift
$OPENSHIFT_REPO
  - cd wsgi

script:
  - py.test

after_success:
  - git push -f openshift $BRANCH:master

archive: true
```

`OPENSHIFT_REPO` should reflect the app you have deployed using OpenShift. It will be different from what is shown in this example.

5. Now commit these changes and push it to GitHub. You will see a build on Shippable triggered and a new app deployed on OpenShift.
6. Visit your app's homepage, and you should see its updated contents.

How it works...

At every build instruction, Shippable spins off new containers depending on the image and language type specified in the `shippable.yml` file and runs the build to perform testing. In our case, Shippable will spin off two containers, one for Python 2.6 and the other for Python 2.7. Shippable adds a webhook to your GitHub repository as follows when you register it with them:



Webhooks allow external services to be notified when certain events happen on GitHub. 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/545e9e69c6f0803064f3d... (push and pull_request)

So every time a change is committed to GitHub, a build on Shippable gets triggered and after the success, it is deployed on OpenShift.

See also

- Detailed documentation is available on the Shippable website at `http://docs.shippable.com/`

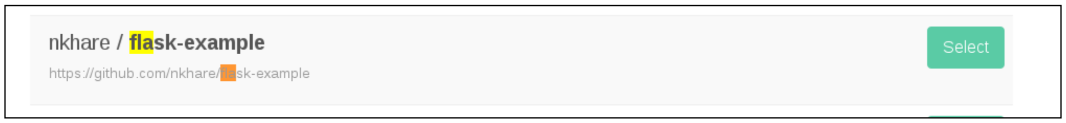
Doing CI/CD with Drone

As mentioned on the Drone website (`https://drone.io/`), Drone is a hosted Continuous Integration service. It enables you to conveniently set up projects to automatically build, test, and deploy as you make changes to your code. They provide an open source version of their platform, which you can host in your environment or on cloud. As of now, they support languages such as C/C++, Dart, Go, Haskell, Groovy, Java, Node.js, PHP, Python, Ruby, and Scala. Using Drone, you can deploy your application on platforms such as Heroku, Dotcloud, Google App Engine, and S3. You can also SSH (rsync) your code to a remote server for deployment.

For this recipe, let's use the same example that we used in the earlier recipes.

Getting ready

1. Log in to Drone (`https://drone.io/`).
2. Click on **New Project** and set up repository. In our case, we'll pick the same repository from GitHub that we used in the previous recipe (`https://github.com/nkhare/flask-example`):

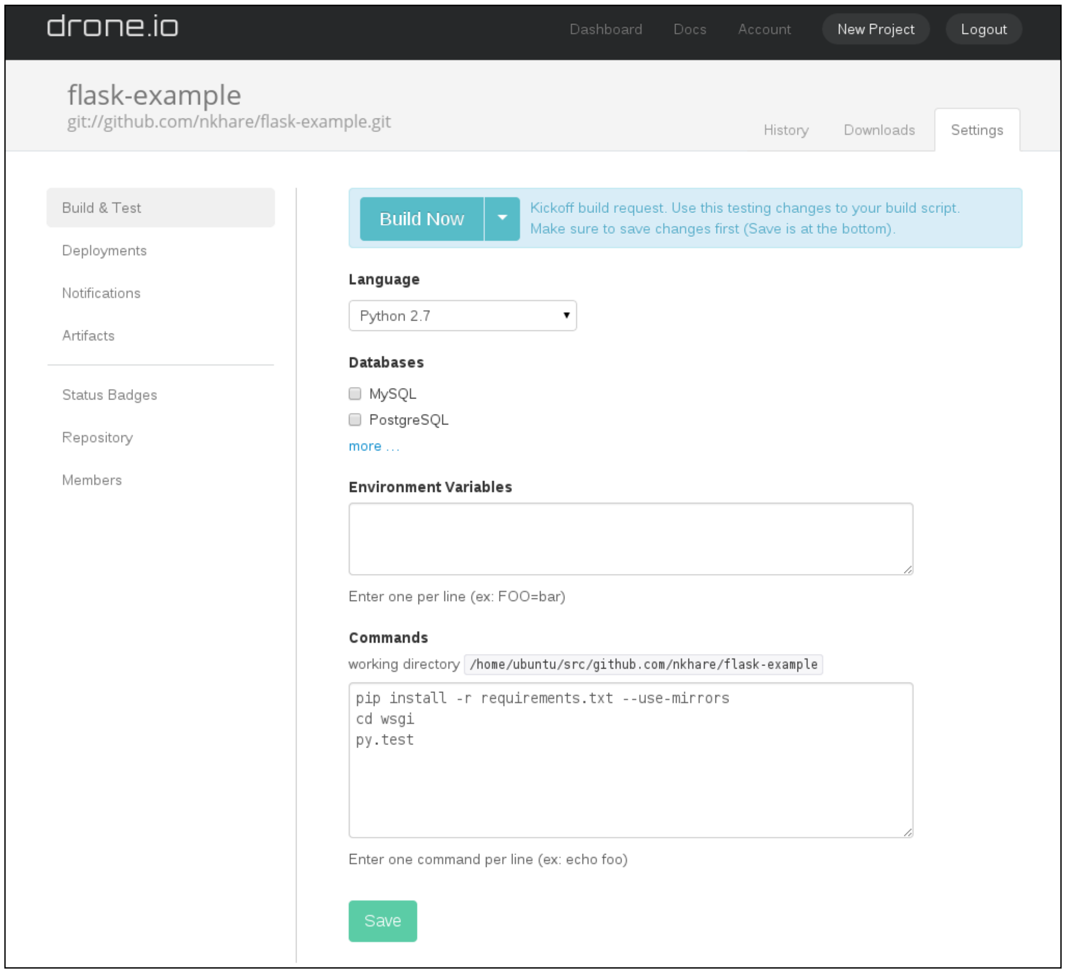


3. Once selected, it will ask you to select the programming language for the selected repository. I selected Python in this case.
4. It will then prompt you to set up the build script. For this recipe, we'll put the following and save it:

```
pip install -r requirements.txt --use-mirrors
cd wsgi
py.test
```

How to do it...

1. Trigger a manual build by clicking on **Build Now**, as shown in the following screenshot:



flask-example
git://github.com/nkhare/flask-example.git

Kickoff build request. Use this testing changes to your build script.
Make sure to save changes first (Save is at the bottom).

Language

Python 2.7

Databases

MySQL
PostgreSQL
more ...

Environment Variables

Enter one per line (ex: FOO=bar)

Commands

working directory `/home/ubuntu/src/github.com/nkhare/flask-example`

```
pip install -r requirements.txt --use-mirrors
cd wsgi
py.test
```

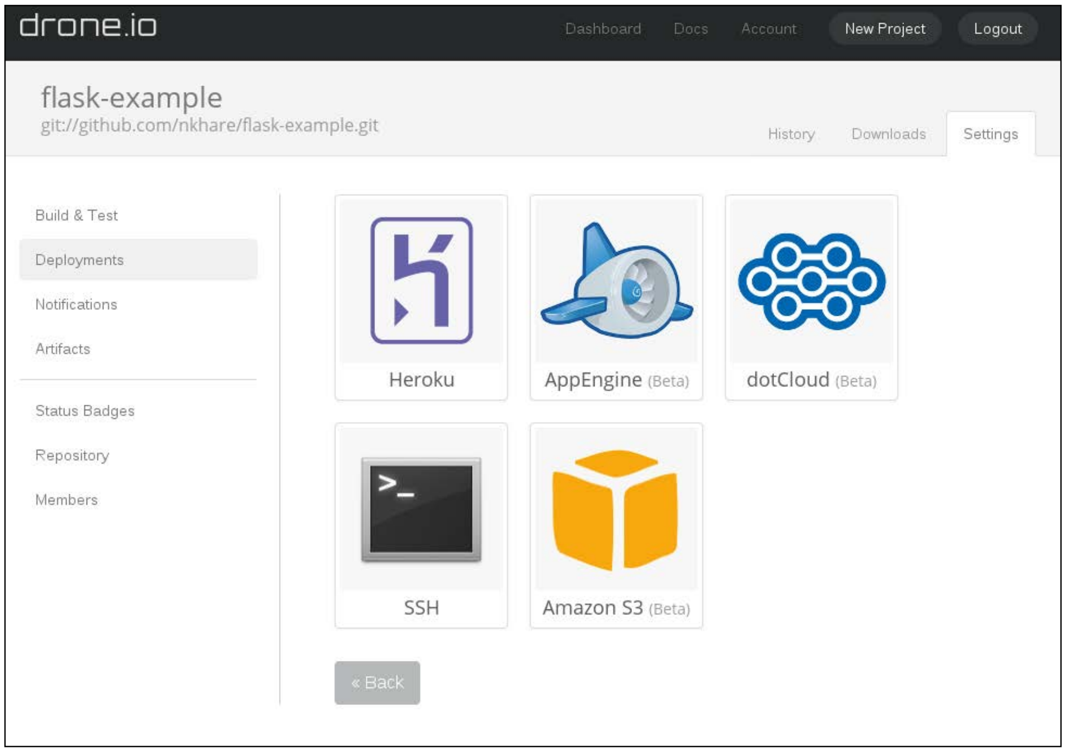
Enter one command per line (ex: echo foo)

How it works...

The build process starts a new container, clones the source code repository, and runs the commands that we specified in the **Commands** section (running the test cases) within it.

There's more...

- Once the build is complete, you can look at the console output.
- Drone also adds a webhook in GitHub; so the next time you commit changes in the repository, a build will be triggered.
- Drone also supports Continuous Deployment to different cloud environments, as we have seen in the earlier recipe. To set that up, go to the **Settings** tab, select **Deployment**, and then select **Add New Deployment**. Select your cloud provider and set it up:



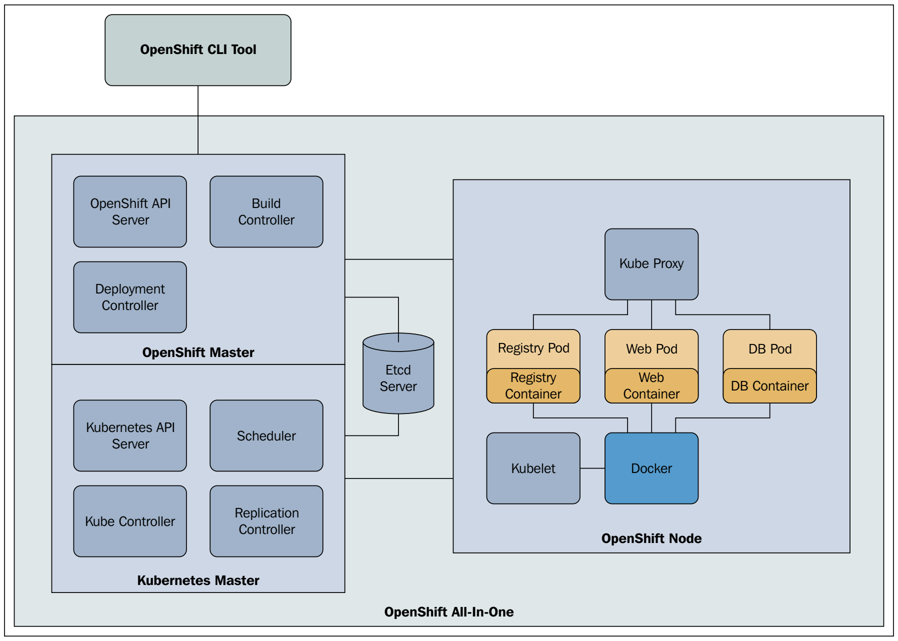
See also

- The Drone documentation at `http://docs.drone.io/`
- The steps to configure a self-hosted Drone environment, which is in the alpha stage as of now, at `https://github.com/drone/drone`

Setting up PaaS with OpenShift Origin

Platform-as-a-Service is a type of cloud service where the consumer controls the 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 (a public provider) or internal (an IT department in an organization). There are many PaaS providers, such as Amazon (`http://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 OpenShift, where we deployed our app to OpenShift PaaS. We deployed our app on Openshift Online, which is the Public Cloud Service. At the time of writing this book, the OpenShift Public Cloud Service uses non-Docker container technology to deploy apps to the Public Cloud Service. The OpenShift team has been working on OpenShift v3 (`https://github.com/openshift/origin`), which is a PaaS that leverages technologies such as Docker and Kubernetes (`http://kubernetes.io`) among others, providing a complete ecosystem to service your cloud-enabled apps. They plan to move this to the Public Cloud Service later this year. As we have talked about Kubernetes in *Chapter 8*, *Docker Orchestration and Hosting Platforms*, it is highly recommended to read that chapter first before continuing with this recipe. I am going to borrow some of the concepts from that chapter.



`https://blog.openshift.com/openshift-v3-deep-dive-docker-kubernetes/`

Kubernetes provides container cluster management with features such as scheduling pods and service discovery, but it does not have the concept of complete application, as well as the capabilities to build and deploy Docker images from the source code. OpenShift v3 extends the base Kubernetes model and fills those gaps. If we fast-forward and look at *Chapter 8*, *Docker Orchestration and Hosting Platforms*, for the Kubernetes section, you will notice that to deploy an app, we need to define Pods, Services, and Replication-Controllers. OpenShift v3 tries to abstract all that information and let you define one configuration file that takes care of all the internal wiring. Furthermore, OpenShift v3 provides other features such as automated deployment through source code push, the centralized administration and management of an application, 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 v3 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** (**STI**) build feature. As there is active development happening on OpenShift v3 Origin, I have selected a tag from the source code and used that code-base in this recipe and the next one. In the newer version, the command-line options may change. With this information in hand, you should be able to adapt to the latest release. The latest example can be found at `https://github.com/openshift/origin/tree/master/examples/hello-openshift`.

Getting ready

Set up Vagrant (`https://www.vagrantup.com/`) and install the VirtualBox provider (`https://www.virtualbox.org/`). The instructions on how to set these up are outside the scope of this book.

1. Clone the OpenShift Origin repository:

   ```
   $ git clone https://github.com/openshift/origin.git
   ```

2. Check out the `v0.4.3` tag:

   ```
   $ cd origin
   $ git checkout tags/v0.4.3
   ```

3. Start the VM:

   ```
   $ vagrant up --provider=virtualbox
   ```

4. Log in to the container:

   ```
   $ vagrant ssh
   ```

How to do it...

1. Build the OpenShift binary:

```
$ cd /data/src/github.com/openshift/origin
$ make clean build
```

2. Go to the `hello-openshift` examples:

```
$  cd /data/src/github.com/openshift/origin/examples/hello-
openshift
```

3. Start all the OpenShift services in one daemon:

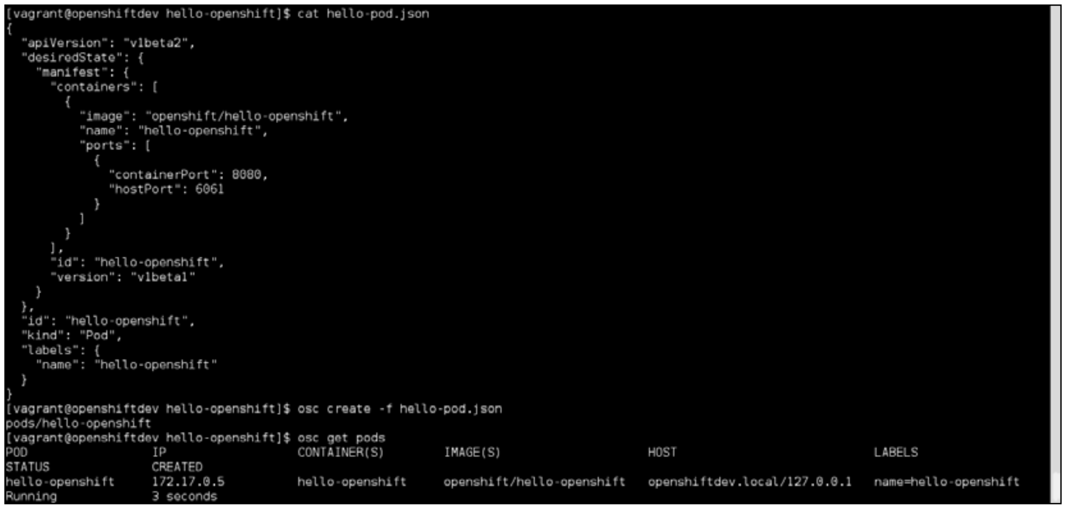
```
$ mkdir logs
$ sudo
/data/src/github.com/openshift/origin/_output/local/go/bin/ope
nshift start --public-master=localhost &> logs/openshift.log &
```

4. OpenShift services are secured by TLS. Our client will need to accept the server certificates and present its own client certificate. Those are generated as part of Openshift start in the current working directory.

```
$ export
OPENSHIFTCONFIG=`pwd`/openshift.local.certificates/admin/.kube
config
$ export
CURL_CA_BUNDLE=`pwd`/openshift.local.certificates/ca/cert.crt
$ sudo chmod a+rwX "$OPENSHIFTCONFIG"
```

5. Create the pod from the `hello-pod.json` definition:

```
$ osc create -f hello-pod.json
```



```
[vagrant@openshiftdev hello-openshift]$ cat hello-pod.json
{
  "apiVersion": "v1beta2",
  "desiredState": {
    "manifest": {
      "containers": [
        {
          "image": "openshift/hello-openshift",
          "name": "hello-openshift",
          "ports": [
            {
              "containerPort": 8080,
              "hostPort": 6061
            }
          ]
        }
      ],
      "id": "hello-openshift",
      "version": "v1beta1"
    }
  },
  "id": "hello-openshift",
  "kind": "Pod",
  "labels": {
    "name": "hello-openshift"
  }
}
[vagrant@openshiftdev hello-openshift]$ osc create -f hello-pod.json
pods/hello-openshift
[vagrant@openshiftdev hello-openshift]$ osc get pods
POD                 IP                  CONTAINER(S)        IMAGE(S)                    HOST                          LABELS
STATUS              CREATED
hello-openshift     172.17.0.5          hello-openshift     openshift/hello-openshift   openshiftdev.local/127.0.0.1  name=hello-openshift
Running             3 seconds
```

6. Connect to the pod:

   `$ curl localhost:6061`

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. In the pod configuration file, we mentioned to map port `6061` of the host machine with port `8080` of the pod. So, when we queried the host on port `6061`, we got a reply from the pod.

There's more...

If you run the `docker ps` command, you will see the corresponding containers running.

See also

- The *Learn More* section on `https://github.com/openshift/origin`
- The OpenShift 3 beta 3 Video tutorial at `https://blog.openshift.com/openshift-3-beta-3-training-commons-briefing-12/`
- The latest OpenShift training at `https://github.com/openshift/training`
- The OpenShift v3 documentation at `http://docs.openshift.org/latest/welcome/index.html`

Building and deploying an app on OpenShift v3 from the source code

OpenShift v3 provides the build process to build an image from source code. The following are the build strategies that one can follow to build images:

- **Docker build**: In this, users will supply to the Docker context (Dockerfiles and support files), which can be used to build images. OpenShift just triggers the `docker build` command to create the image.
- **Source-to-image (STI) build**: In this, the developer defines the source code repository and the builder image, which defines the environment used to create the app. STI then uses the given source code and builder image to create a new image for the app. More details about STI 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 STI build process. We are going to look at sample-app from the OpenShift v3 Origin repo (`https://github.com/openshift/origin/tree/v0.4.3/examples/sample-app`). The corresponding STI build file is located at `https://github.com/openshift/origin/blob/v0.4.3/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 `openshift/ruby-20-centos7` image. So, we will use the `openshift/ruby-20-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, depending 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 STI build file (`application-template-stibuild.json`), you will find multiple `DeploymentConfig` sections, one of each pod. A `DeploymentConfig` section has information such as exported ports, replicas, the environment variables for the pod, and other info. In simple terms, you can think of `DeploymentConfig` as an extended replication controller of Kubernetes. It also has triggers to trigger new deployment. Each time a new deployment is created, the `latestVersion` field of `DeploymentConfig` is incremented. A `deploymentCause` is also added to `DeploymentConfig` describing the change that led to the latest deployment.

`ImageRepository`, which was recently renamed as `ImageStream`, is a stream of related images. `BuildConfig` and `DeploymentConfig` watch `ImageStream` to look for image changes and react accordingly, based on their respective triggers.

The other sections that you will find in the STI build file are services for pods (database and frontend), a route for the frontend service through which the app can be accessed, and a template. 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 STI build, there are examples of Docker and custom build 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 recipe, *Setting up PaaS with OpenShift Origin*.

Your current working directory should be `/data/src/github.com/openshift/origin / examples/hello-openshift` inside the VM, started by Vagrant.

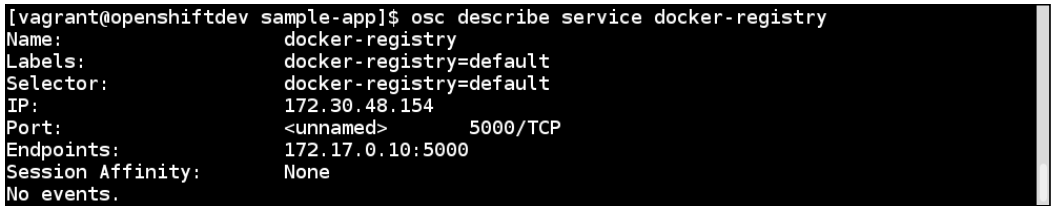
How to do it...

1. Deploy a private Docker registry to host images created by the STI build process:

   ```
   $ sudo openshift ex registry --create --
   credentials=./openshift.local.certificates/openshift-
   registry/.kubeconfig
   ```

2. Confirm the registry has started (this can take a few minutes):

   ```
   $ osc describe service docker-registry
   ```



```
[vagrant@openshiftdev sample-app]$ osc describe service docker-registry
Name:                   docker-registry
Labels:                 docker-registry=default
Selector:               docker-registry=default
IP:                     172.30.48.154
Port:                   <unnamed>       5000/TCP
Endpoints:              172.17.0.10:5000
Session Affinity:       None
No events.
```

3. Create a new project in OpenShift. This creates a namespace `test` to contain the builds and an app that we will generate later:

   ```
   $ openshift ex new-project test --display-name="OpenShift 3
   Sample" --description="This is an example project to
   demonstrate OpenShift v3" --admin=test-admin
   ```

4. Log in with the `test-admin` user and switch to the `test` project, which will be used by every command from now on:

   ```
   $ osc login -u test-admin -p pass
   ```

   ```
   $ osc project test
   ```

5. Submit the application template for processing (generating shared parameters requested in the template) and then request the creation of the processed template:

   ```
   $ osc process -f application-template-stibuild.json | osc
   create -f -
   ```

6. This will not trigger the build. To start the build of your application, run the following command:

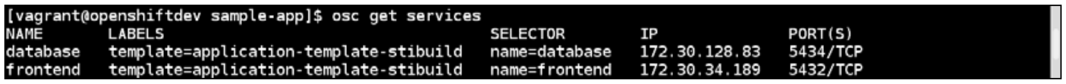
   ```
   $ osc start-build ruby-sample-build
   ```

7. Monitor the build and wait for the status to go to `complete` (this can take a few minutes):

   ```
   $ osc get builds
   ```

8. Get the list of services:

   ```
   $ osc get services
   ```



```
[vagrant@openshiftdev sample-app]$ osc get services
NAME       LABELS                                    SELECTOR        IP              PORT(S)
database   template=application-template-stibuild    name=database   172.30.128.83   5434/TCP
frontend   template=application-template-stibuild    name=frontend   172.30.34.189   5432/TCP
```

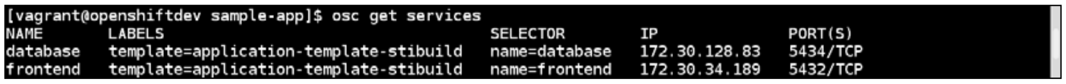
How it works...

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 `openshift/ruby-20-centos7`. So the build process takes that image, and with STI builder, a new image called `origin-ruby-sample` is created after building our source on `openshift/ruby-20-centos7`. The new image is then pushed to the Docker registry we created earlier.

With `DeploymentConfig`, frontend and backend pods are also deployed and linked to corresponding services.

There's more...

- The preceding frontend service can be accessed through the service IP and corresponding port, but it will not be accessible from the outside world. To make it accessible, we give our app an FQDN; for instance, in the following example, it is defined as `www.example.com`:



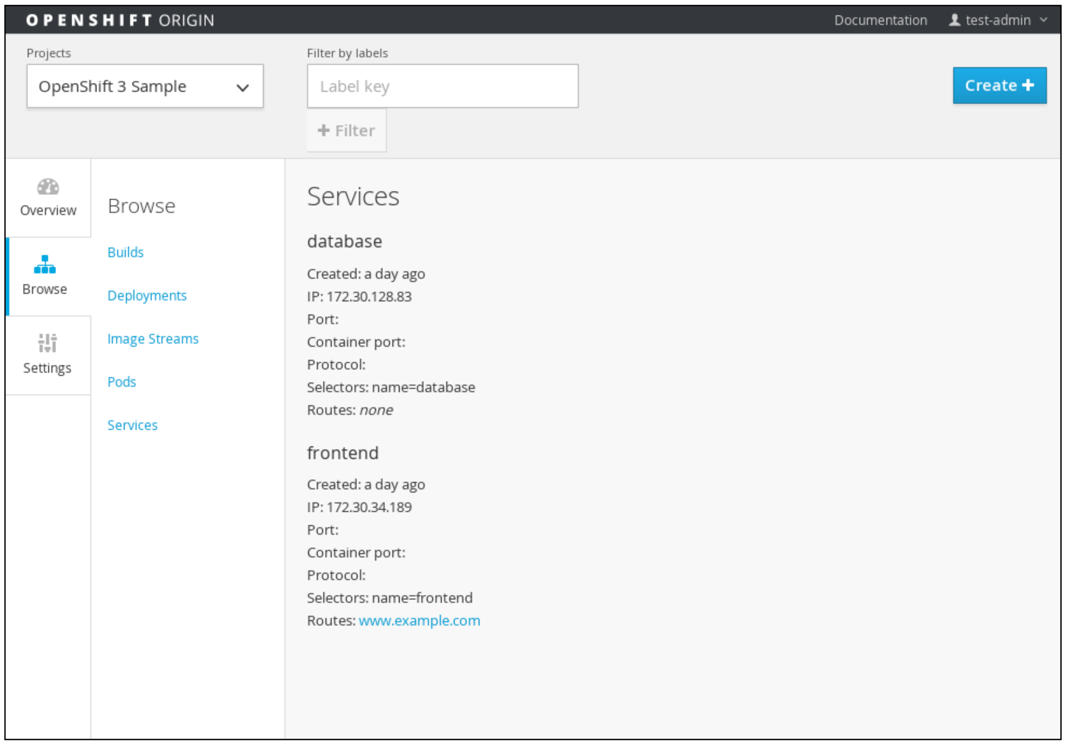
```
[vagrant@openshiftdev sample-app]$ osc get services
NAME       LABELS                                    SELECTOR        IP              PORT(S)
database   template=application-template-stibuild    name=database   172.30.128.83   5434/TCP
frontend   template=application-template-stibuild    name=frontend   172.30.34.189   5432/TCP
```

OpenShift v3 provides an HAProxy router, which can map over FQDN to the corresponding pod. For more information, please visit `http://docs.openshift.org/latest/architecture/core_objects/routing.html`. You will also require an entry in the external DNS to resolve the FQDN provided here.

- OpenShift v3 Origin is also a management GUI. To look at our deployed app on the GUI, bind the username `test-admin` to the view role in the default namespace so you can observe the progress in the web console:

  ```
  $ openshift ex policy add-role-to-user view test-admin
  ```

  Then, through the browser, connect to `https://<host>:8443/console` and log in through the `test-admin` user by giving any password. As Vagrant forwards the traffic of port `8443` on the host machine to the VM, you should be able to connect through the host on which VM is running. Then select **OpenShift 3 Sample** as the project and explore:



Services

database

Created: a day ago
IP: 172.30.128.83
Port:
Container port:
Protocol:
Selectors: name=database
Routes: *none*

frontend

Created: a day ago
IP: 172.30.34.189
Port:
Container port:
Protocol:
Selectors: name=frontend
Routes: www.example.com

- In the multiple node setup, your pods can be scheduled on different systems. OpenShift v3 connects pods though the overlay network pod running on one node can access another. It is called `openshift-sdn`. For more details, please visit `https://github.com/openshift/openshift-sdn`.

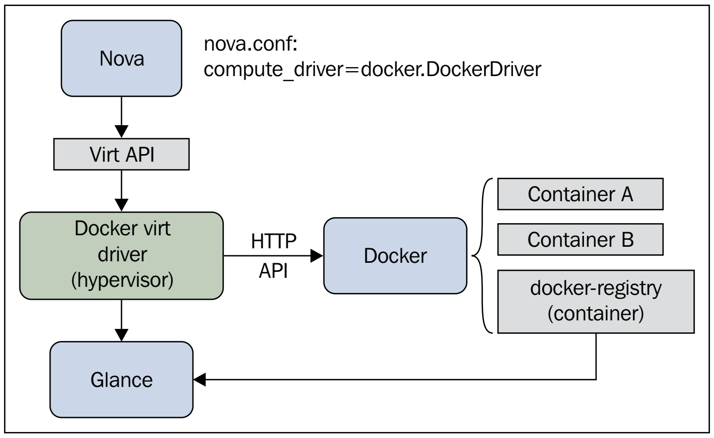
See also

- The *Learn More* section at `https://github.com/openshift/origin`
- The OpenShift 3 beta 3 video tutorial at `https://blog.openshift.com/openshift-3-beta-3-training-commons-briefing-12/`
- The latest OpenShift training at `https://github.com/openshift/training`
- The OpenShift v3 documentation at `http://docs.openshift.org/latest/welcome/index.html`

Configuring Docker as a hypervisor driver for OpenStack

I am assuming that the reader has some exposure to OpenStack for this recipe, as covering it is outside the scope of this book. For more information on OpenStack and its components, please visit `http://www.openstack.org/software/`.

In OpenStack, Nova supports different hypervisors for computation, such as KVM, XEN, VMware, HyperV, and others. We can provision VMs using these drivers. Using Ironic (`https://wiki.openstack.org/wiki/Ironic`), you can provision bare metal as well. Nova added support for containers provisioning using Docker in the Havana (`https://www.openstack.org/software/havana/`) release, but currently, it lives out of the mainline for faster dev cycle. There are plans to merge it in the mainline in the future. Under the hood, it looks like this:


Nova
nova.conf:
compute_driver=docker.DockerDriver
Virt API
Docker virt driver (hypervisor)
HTTP API
Docker
Container A
Container B
docker-registry (container)
Glance

`https://wiki.openstack.org/wiki/File:Docker-under-the-hood.png`

DevStack (`http://docs.openstack.org/developer/devstack/overview.html`) is a collection of scripts to quickly create an OpenStack development environment. It is not a general-purpose installer, but it is a very easy way to get started with OpenStack. In this recipe, we'll configure DevStack's environment with Docker as Nova driver on Fedora21.

Getting ready

1. Install Docker on the system.
2. Clone `nova-docker` and `devstack`:

   ```
   $ git clone https://git.openstack.org/stackforge/nova-docker
   /opt/stack/nova-docker
   $ git clone https://git.openstack.org/openstack-dev/devstack
   /opt/stack/devstack
   ```

3. The following step is needed until we can make use of `configure_nova_hypervisor_rootwrap`:

   ```
   $ git clone https://git.openstack.org/openstack/nova
   /opt/stack/nova
   ```

4. Prepare Devstack for installation:

   ```
   $ cd /opt/stack/nova-docker
   $ ./contrib/devstack/prepare_devstack.sh
   ```

5. Create the stack user and add it to `sudo`:

   ```
   $ /opt/stack/devstack/tools/create-stack-user.sh
   ```

6. Install `docker-py` to communicate with docker through Python:

   ```
   $ yum install python-pip
   $ pip install docker-py
   ```

How to do it...

1. After the prerequisite steps are completed, run the following commands to install Devstack:

   ```
   $ cd /opt/stack/devstack
   $ ./stack.sh
   ```

How it works...

- The `prepare_devstack.sh` driver makes the following entries in the `localrc` file set the right environment to set Docker for the Nova driver:

  ```
  export VIRT_DRIVER=docker
  export DEFAULT_IMAGE_NAME=cirros
  export NON_STANDARD_REQS=1
  export IMAGE_URLS=" "
  ```

- After running the `stackrc` file, we can see the following changes with respect to Nova and Glance:
  - The `/etc/nova/nova.conf` file changes the compute driver:

    ```
    [DEFAULT]
    compute_driver = novadocker.virt.docker.DockerDriver
    ```

  - The `/etc/nova/rootwrap.d/docker.filters` file is updated with the following content:

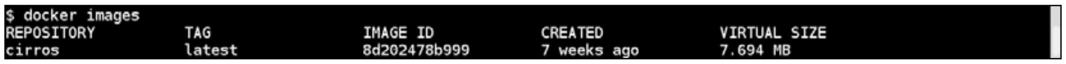
    ```
    [Filters]
    # nova/virt/docker/driver.py: 'ln', '-sf',
    '/var/run/netns/.*'
    ln: CommandFilter, /bin/ln, root
    ```

  - In `/etc/glance/glance-api.conf`, adds `docker` in the container/image format:

    ```
    [DEFAULT]
    container_formats = ami,ari,aki,bare,ovf,docker
    ```

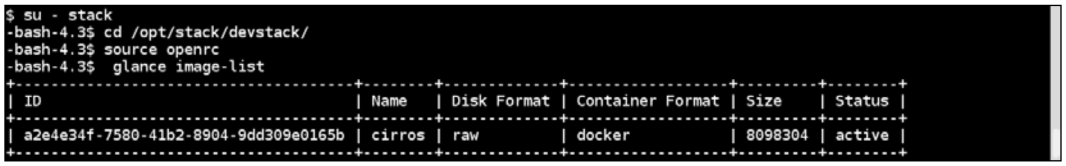
There's more...

- In `localrc`, we mentioned `cirros` as the default image, so once the setup is completed, we can see that the Docker image for `cirros` is downloaded:



```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
cirros              latest              8d202478b999        7 weeks ago         7.694 MB
```

  This is being imported to Glance automatically:



```
$ su - stack
-bash-4.3$ cd /opt/stack/devstack/
-bash-4.3$ source openrc
-bash-4.3$  glance image-list
+--------------------------------------+--------+-------------+------------------+---------+--------+
| ID                                   | Name   | Disk Format | Container Format | Size    | Status |
+--------------------------------------+--------+-------------+------------------+---------+--------+
| a2e4e34f-7580-41b2-8904-9dd309e0165b | cirros | raw         | docker           | 8098304 | active |
+--------------------------------------+--------+-------------+------------------+---------+--------+
```

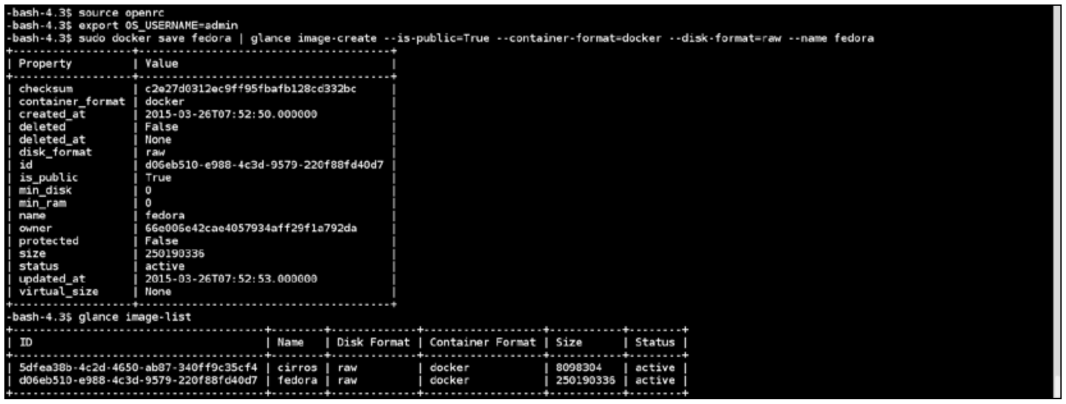
From the preceding screenshot, we can see that the container format is Docker.

- Now you can create an instance using a `cirros` image using Horizon, or from the command line, and look at the container started using the Docker command line.
- To import any image to Glance, you can do something like the following:
    - Pull the required image from Docker Hub:

      ```
      $ docker pull fedora
      ```

    - Import the image (currently only admin can import the image):

      ```
      $ source openrc
      $ export OS_USERNAME=admin
      $ sudo docker save fedora | glance image-create --is-
      public=True --container-format=docker --disk-format=raw
      --name fedora
      ```



```
-bash-4.3$ source openrc
-bash-4.3$ export OS_USERNAME=admin
-bash-4.3$ sudo docker save fedora | glance image-create --is-public=True --container-format=docker --disk-format=raw --name fedora
+------------------+--------------------------------------+
| Property         | Value                                |
+------------------+--------------------------------------+
| checksum         | c2e27d0312ec9ff95fbafb128cd332bc     |
| container_format | docker                               |
| created_at       | 2015-03-26T07:52:50.000000           |
| deleted          | False                                |
| deleted_at       | None                                 |
| disk_format      | raw                                  |
| id               | d06eb510-e988-4c3d-9579-220f88fd40d7 |
| is_public        | True                                 |
| min_disk         | 0                                    |
| min_ram          | 0                                    |
| name             | fedora                               |
| owner            | 66e006e42cae4057934aff29f1a792da     |
| protected        | False                                |
| size             | 250190336                            |
| status           | active                               |
| updated_at       | 2015-03-26T07:52:53.000000           |
| virtual_size     | None                                 |
+------------------+--------------------------------------+
-bash-4.3$ glance image-list
+--------------------------------------+--------+-------------+------------------+-----------+--------+
| ID                                   | Name   | Disk Format | Container Format | Size      | Status |
+--------------------------------------+--------+-------------+------------------+-----------+--------+
| 5dfea38b-4c2d-4650-ab87-340ff9c35cf4 | cirros | raw         | docker           | 8098304   | active |
| d06eb510-e988-4c3d-9579-220f88fd40d7 | fedora | raw         | docker           | 250190336 | active |
+--------------------------------------+--------+-------------+------------------+-----------+--------+
```

- There is a lack of integration with Cinder and Neutron, but things are catching up quickly.
- While installing, if you get the `AttributeError: 'module' object has no attribute 'PY2' error`, then run the following commands to fix it:

  ```
  $ pip uninstall  six
  $ pip install --upgrade   six
  ```

See also

- The documentation on OpenStack website at `https://wiki.openstack.org/wiki/Docker`.
- Docker is also one of the resource types for OpenStack Heat. Learn more about it at `http://docs.openstack.org/developer/heat/template_guide/contrib.html#dockerinc-resource`.
- There is an interesting project in OpenStack called Kolla, which focuses on deploying OpenStack services through Docker containers. Find more about it at `https://github.com/stackforge/kolla/`.

6
Docker APIs and Language Bindings

In this chapter, we will cover the following recipes:

- Configuring the Docker daemon remote API
- Performing image operations using remote APIs
- Performing container operations using remote APIs
- Exploring Docker remote API client libraries
- Securing the Docker daemon remote API

Introduction

In the previous chapters, we learned different commands to manage images, containers, and so on. Though we run all the commands through the command line, the communication between the Docker client (CLI) and the Docker daemon happens through APIs, which are called Docker daemon remote APIs.

Docker also provides APIs to communicate with Docker Hub and Docker registry, which the Docker client uses as well. In addition to these APIs, we have Docker bindings for different programming languages. So, if you want to build a nice GUI for Docker images, container management, and so on, understanding the APIs mentioned earlier would be a good starting point.

In this chapter, we look into the Docker daemon remote API and use the `curl` command (`http://curl.haxx.se/docs/manpage.html`) to communicate with the endpoints of different APIs, which will look something like the following command:

```
$ curl -X <REQUEST> -H <HEADER> <OPTION> <ENDPOINT>
```

The preceding request will return with a return code and an output corresponding to the endpoint and request we chose. `GET`, `PUT`, and `DELETE` are the different kinds of requests, and GET is the default request if nothing is specified. Each API endpoint has its own interpretation for the return code.

Configuring the Docker daemon remote API

As we know, Docker has a client-server architecture. When we install Docker, a user space program and a daemon get started from the same binary. The daemon binds to `unix://var/run/docker.sock` by default on the same host. This will not allow us to access the daemon remotely. To allow remote access, we need to start Docker such that it allows remote access, which can done by changing the `-H` flag appropriately.

Getting ready

Depending on the Linux distribution you are running, figure out the Docker daemon configuration file you need to change. For Fedora, /Red Hat distributions, it would be `/etc/sysconfig/docker` and for Ubuntu/Debian distributions , it would most likely be `/etc/default/docker`.

How to do it...

1. On Fedora 20 systems, add the `-H tcp://0.0.0.0:2375` option in the configuration file (`/etc/sysconfig/docker`), as follows:

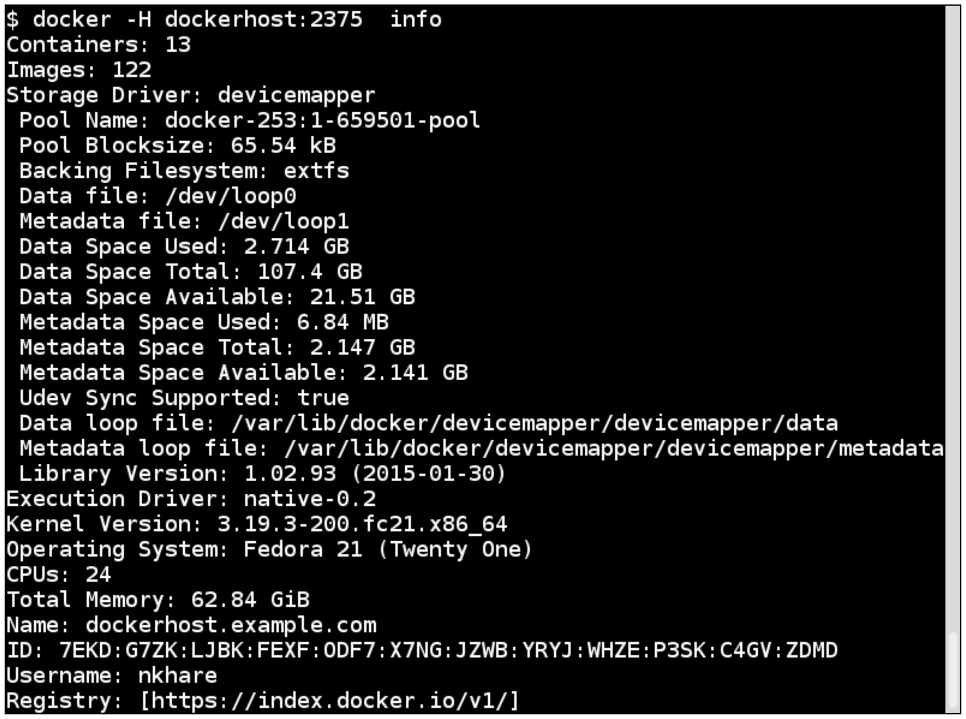
   ```
   OPTIONS=--selinux-enabled -H tcp://0.0.0.0:2375
   ```

2. Restart the Docker service. On Fedora, run the following command:

   ```
   $ sudo systemctl restart docker
   ```

3. Connect to the Docker host from the remote client:

   ```
   $ docker -H <Docker Host>:2375 info
   ```



```
$ docker -H dockerhost:2375  info
Containers: 13
Images: 122
Storage Driver: devicemapper
 Pool Name: docker-253:1-659501-pool
 Pool Blocksize: 65.54 kB
 Backing Filesystem: extfs
 Data file: /dev/loop0
 Metadata file: /dev/loop1
 Data Space Used: 2.714 GB
 Data Space Total: 107.4 GB
 Data Space Available: 21.51 GB
 Metadata Space Used: 6.84 MB
 Metadata Space Total: 2.147 GB
 Metadata Space Available: 2.141 GB
 Udev Sync Supported: true
 Data loop file: /var/lib/docker/devicemapper/devicemapper/data
 Metadata loop file: /var/lib/docker/devicemapper/devicemapper/metadata
 Library Version: 1.02.93 (2015-01-30)
Execution Driver: native-0.2
Kernel Version: 3.19.3-200.fc21.x86_64
Operating System: Fedora 21 (Twenty One)
CPUs: 24
Total Memory: 62.84 GiB
Name: dockerhost.example.com
ID: 7EKD:G7ZK:LJBK:FEXF:ODF7:X7NG:JZWB:YRYJ:WHZE:P3SK:C4GV:ZDMD
Username: nkhare
Registry: [https://index.docker.io/v1/]
```

Make sure the firewall allows access to port `2375` on the system where the Docker daemon is installed.

How it works…

With the preceding command, we allowed the Docker daemon to listen on all network interfaces through port `2375`, using TCP.

There's more…

- With the communication that we mentioned earlier between the client and Docker, the host is insecure. Later in this chapter, we'll see how to enable TLS between them.
- The Docker CLI looks for environment variables; if it is being set then the CLI uses that endpoint to connect, for example, if we connect set as follows:

  ```
  $ export DOCKER_HOST=tcp://dockerhost.example.com:2375
  ```

  Then, the future docker commands in that session connect to remote Docker Host by default and run this:

  ```
  $ docker info
  ```

See also

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

Performing image operations using remote APIs

After enabling the Docker daemon remote API, we can do all image-related operations through a client. To get a better understanding of the APIs, let's use `curl` to connect to the remote daemon and do some image-related operations.

Getting ready

Configure the Docker daemon and allow remote access, as explained in the previous recipe.

How to do it…

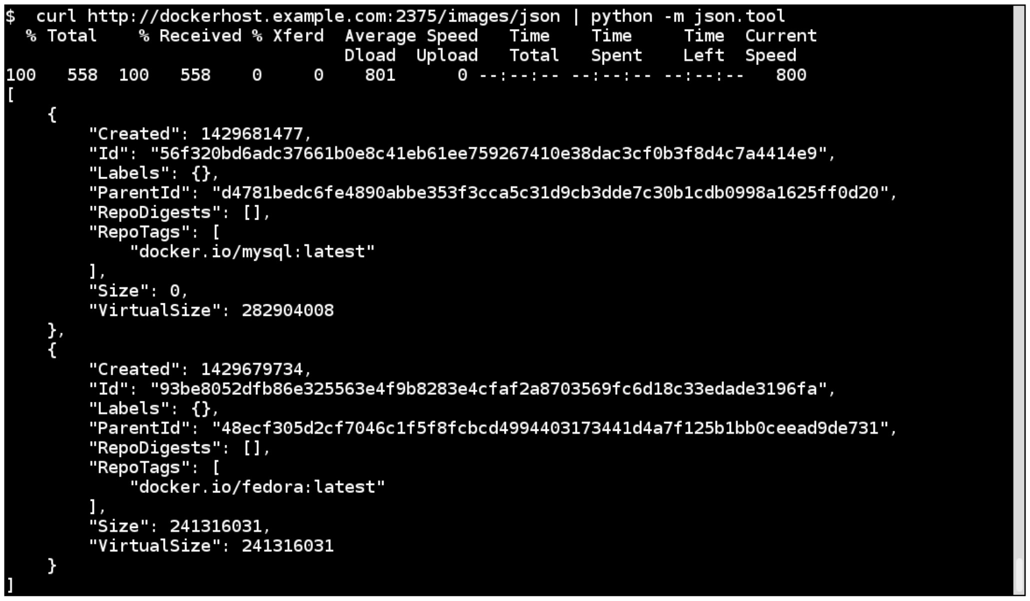
In this recipe, we'll look at a few image operations as follows:

1. To list images, use the following API:

   `GET /images/json`

   Here is an example of the preceding syntax:

   ```
   $ curl http://dockerhost.example.com:2375/images/json
   ```



```
$  curl http://dockerhost.example.com:2375/images/json | python -m json.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   558  100   558    0     0    801      0 --:--:-- --:--:-- --:--:--   800
[
    {
        "Created": 1429681477,
        "Id": "56f320bd6adc37661b0e8c41eb61ee759267410e38dac3cf0b3f8d4c7a4414e9",
        "Labels": {},
        "ParentId": "d4781bedc6fe4890abbe353f3cca5c31d9cb3dde7c30b1cdb0998a1625ff0d20",
        "RepoDigests": [],
        "RepoTags": [
            "docker.io/mysql:latest"
        ],
        "Size": 0,
        "VirtualSize": 282904008
    },
    {
        "Created": 1429679734,
        "Id": "93be8052dfb86e325563e4f9b8283e4cfaf2a8703569fc6d18c33edade3196fa",
        "Labels": {},
        "ParentId": "48ecf305d2cf7046c1f5f8fcbcd4994403173441d4a7f125b1bb0ceead9de731",
        "RepoDigests": [],
        "RepoTags": [
            "docker.io/fedora:latest"
        ],
        "Size": 241316031,
        "VirtualSize": 241316031
    }
]
```

2. To create an image, use the following API:

   `POST /images/create`

   Here are a few examples:

   - Get the Fedora image from Docker Hub:

     ```
     $ curl -X POST
     http://dockerhost.example.com:2375/images/
     create?fromImage=fedora
     ```

   - Get the WordPress image with the `latest` tag:

     ```
     $  curl -X POST
     http://dockerhost.example.com:2375/images/create?fromImage=
     wordpress&tag=latest
     ```

   - Create an image from the `tar` file, which is hosted on the accessible web server:

     ```
     $ curl -X POST
     http://dockerhost.example.com:2375/images/
     create?fromSrc=http://localhost/image.tar
     ```

3. To build an image, use the following API:

   ```
   POST  /commit
   ```

   Here are a few examples:

   - Build an image from the container (`container id = 704a7c71f77d`)

     ```
     $ curl -X POST
     http://dockerhost.example.com:2375/
     commit?container=704a7c71f77d
     ```

   - Build an image from the Docker file:

     ```
     $  curl -X POST  -H "Content-type:application/tar" --data-
     binary '@/tmp/Dockerfile.tar.gz'
     http://dockerhost.example.com:2375/build?t=apache
     ```

   As the API expects the content as a `tar` file, we need to put the Docker file inside a tar and call the API.

4. To delete an image, use the following API:

   ```
   DELETE  /images/<name>
   ```

   Here is an example of the preceding syntax:

   ```
   $ curl -X DELETE
   http://dockerhost.example.com:2375/images/wordpress:3.9.1
   ```

How it works...

In all the cases mentioned earlier, the APIs will connect to the Docker daemon and perform the requested operations.

There's more...

We have not covered all the options of the APIs discussed earlier and Docker provides APIs for other image-related operations. Visit the API documentation for more details.

See also

- Each API endpoint can have different inputs to control the operations. For more details, visit the documentation on the Docker website `https://docs.docker.com/reference/api/docker_remote_api_v1.18/#22-images`.

Performing container operations using remote APIs

In a similar way to how we performed image operations using APIs, we can also do all container-related operations using APIs.

Getting ready

Configure the Docker daemon and allow remote access, as explained in the earlier recipe.

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
   ```

   Here are a few examples:

   - Get all the running containers:

     ```
     $ curl -X GET
     http://shadowfax.example.com:2375/containers/json
     ```

   - Get all the running containers, including the stopped ones

     ```
     $ curl -X GET http://shadowfax.example.com:2375/containers/
     json?all=True
     ```

2. To create a new container, use the following API:

   ```
   POST  /containers/create
   ```

   Here are a few examples

   - Create a container from the `fedora` image:

     ```
     $ curl -X POST  -H "Content-type:application/json" -d
     '{"Image": "fedora", "Cmd": ["ls"] }'
     http://dockerhost.example.com:2375/containers/create
     ```

   - Create a container from the `fedora` image and name it `f21`:

     ```
     $ curl -X POST  -H "Content-type:application/json" -d
     '{"Image": "fedora", "Cmd": ["ls"] }'
     http://dockerhost.example.com:2375/containers/
     create?name=f21
     ```

3. To start a container, use the following API:

   ```
   POST /containers/<id>/start
   ```

   For example, start a container with the `591ab8ac2650` ID:

   ```
   $ curl -X POST  -H "Content-type:application/json" -d '{"Dns":
   ["4.2.2.1"] }'
   http://dockerhost.example.com:2375/containers/591ab8ac2650/sta
   rt
   ```

   Note that while starting the stopped container, we also passed the DNS option, which will change the DNS configuration of the container.

4. To inspect a container, use the following API:

   ```
   GET  /containers/<id>/json
   ```

   For example, inspect a container with the `591ab8ac2650` ID:

   ```
   $ curl -X GET
   http://dockerhost.example.com:2375/containers/591ab8ac2650/json
   ```

5. To get a list of processes running inside a container, use the following API:

   ```
   GET /containers/<id>/top
   ```

   For example, get the processes running in the container with the `591ab8ac2650` ID:

   ```
   $ curl -X GET
   http://dockerhost.example.com:2375/containers/591ab8ac2650/top
   ```

6. To stop a container, use the following API:

   ```
   POST /containers/<id>/stop
   ```

   For example, stop a container with the `591ab8ac2650` ID:

   ```
   $ curl -X POST
   http://dockerhost.example.com:2375/containers/591ab8ac2650/sto
   p
   ```

How it works…

We have not covered all the options of the APIs discussed earlier and Docker provides APIs for other container-related operations. Visit the API documentation for more details.

See also

- The documentation on the Docker website at `https://docs.docker.com/reference/api/docker_remote_api_v1.18/#21-containers`

Exploring Docker remote API client libraries

In the last few recipes, we explored the APIs provided by Docker to connect and perform operations to the remote Docker daemon. The Docker community has added bindings for different programming languages to access those APIs. Some of them are listed at `https://docs.docker.com/reference/api/remote_api_client_libraries/`.

Note that Docker Maintainers do not maintain these libraries. Let's explore Python bindings with a few examples and see how it uses the Docker remote API.

Getting ready

- Install `docker-py` on Fedora:

  ```
  $ sudo yum install python-docker-py
  ```

  Alternatively, use `pip` to install the package:

  ```
  $ sudo pip install docker-py
  ```

- Import the module:

  ```
  $ python
  >>> import docker
  ```

How to do it...

1. Create the client, using the following steps:
    1. Connect through the Unix Socket:

       ```
       >>> client =
       docker.Client(base_url='unix://var/run/docker.sock',
       version='1.18',  timeout=10)
       ```

    2. Connect over HTTP:

       ```
       >>> client =
       docker.Client(base_url='http://dockerhost.example.com:2375',
       version='1.18',  timeout=10)
       ```

   Here, `base_url` is the endpoint to connect, `version` is the API version the client will use, and `timeout` is the timeout value in seconds.

2. Search for an image using the following code:

   ```
   >>> client.search ("fedora")
   ```

3. Pull an image using the following code:

```
>>> client.pull("fedora", tag="latest")
```

4. Start a container using the following code:

```
>>> client.create_container("fedora", command="ls",
hostname=None, user=None, detach=False, stdin_open=False,
tty=False, mem_limit=0, ports=None, environment=None,
dns=None, volumes=None,
volumes_from=None,network_disabled=False, name=None,
entrypoint=None, cpu_shares=None,
working_dir=None,memswap_limit=0)
```

How it works...

In all the preceding cases, the Docker Python module will send RESTful requests to the endpoint using the API provided by Docker. Look at the methods such as `search`, `pull`, and `start` in the following code of `docker-py` available at `https://github.com/docker/docker-py/blob/master/docker/client.py`.

There's more...

You can explore different user interfaces written for Docker. Some of them are as follows:

- Shipyard (`http://shipyard-project.com/`)—written in Python
- DockerUI (`https://github.com/crosbymichael/dockerui`)—written in JavaScript using AngularJS

Securing the Docker daemon remote API

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 the host running the Docker daemon is `dockerhost.example.com`.

Getting ready

Make sure you have the `openssl` library installed.

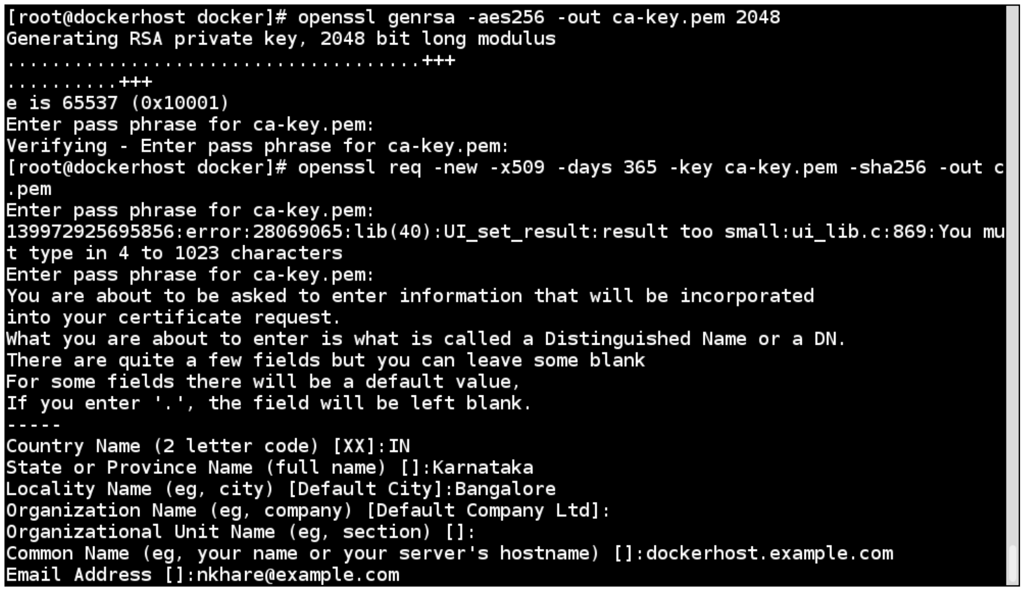
How to do it…

1. Create a directory on your host to put our CA and other related files:

   ```
   $ mkdirc-p /etc/docker
   $ cd  /etc/docker
   ```

2. Create the CA private and public keys:

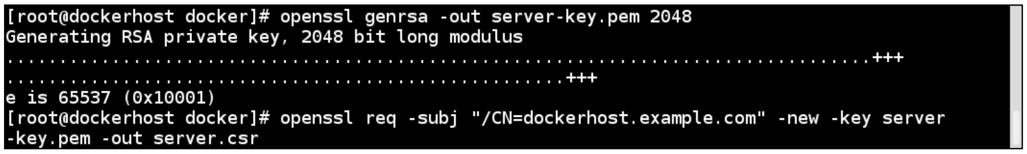
   ```
   $ openssl genrsa -aes256 -out ca-key.pem 2048
   $ openssl req -new -x509 -days 365 -key ca-key.pem -sha256 -
   out ca.pem
   ```



```
[root@dockerhost docker]# openssl genrsa -aes256 -out ca-key.pem 2048
Generating RSA private key, 2048 bit long modulus
...................................+++
.........+++
e is 65537 (0x10001)
Enter pass phrase for ca-key.pem:
Verifying - Enter pass phrase for ca-key.pem:
[root@dockerhost docker]# openssl req -new -x509 -days 365 -key ca-key.pem -sha256 -out c
.pem
Enter pass phrase for ca-key.pem:
139972925695856:error:28069065:lib(40):UI_set_result:result too small:ui_lib.c:869:You mu
t type in 4 to 1023 characters
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) [XX]:IN
State or Province Name (full name) []:Karnataka
Locality Name (eg, city) [Default City]:Bangalore
Organization Name (eg, company) [Default Company Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (eg, your name or your server's hostname) []:dockerhost.example.com
Email Address []:nkhare@example.com
```

3. Now, let's create the server key and certificate signing request. Make sure that `Common Name` matches the Docker daemon system's hostname. In our case, it is `dockerhost.example.com`.

   ```
   $ openssl genrsa -out server-key.pem 2048
   $ openssl req -subj "/CN=dockerhost.example.com" -new -key
   server-key.pem -out server.csr
   ```

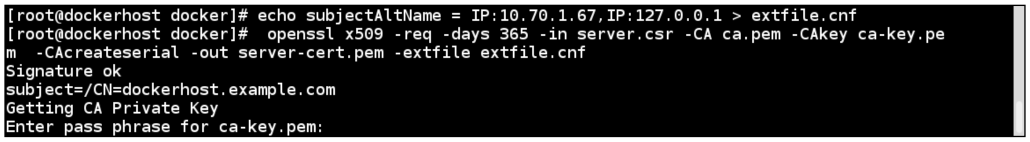


```
[root@dockerhost docker]# openssl genrsa -out server-key.pem 2048
Generating RSA private key, 2048 bit long modulus
.............................................................................+++
..................................................+++
e is 65537 (0x10001)
[root@dockerhost docker]# openssl req -subj "/CN=dockerhost.example.com" -new -key server
-key.pem -out server.csr
```

4. To allow connections from 127.0.0.1 and a specific host, for example, 10.70.1.67, create an extensions configuration file and sign the public key with our CA:

   ```
   $ echo subjectAltName = IP:10.70.1.67,IP:127.0.0.1 > extfile.cnf
   ```

   ```
   $ openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkey
   ca-key.pem    -CAcreateserial -out server-cert.pem -extfile
   extfile.cnf
   ```

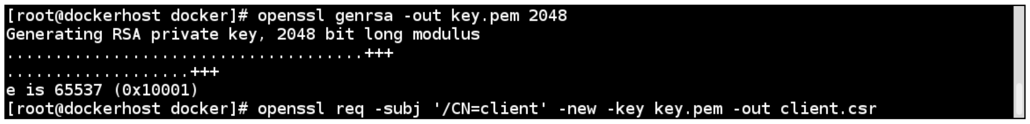


```
[root@dockerhost docker]# echo subjectAltName = IP:10.70.1.67,IP:127.0.0.1 > extfile.cnf
[root@dockerhost docker]#  openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkey ca-key.pe
m  -CAcreateserial -out server-cert.pem -extfile extfile.cnf
Signature ok
subject=/CN=dockerhost.example.com
Getting CA Private Key
Enter pass phrase for ca-key.pem:
```

5. For client authentication, create a client key and certificate signing request:

   ```
   $ openssl genrsa -out key.pem 2048
   ```

   ```
   $ openssl req -subj '/CN=client' -new -key key.pem -out client.csr
   ```

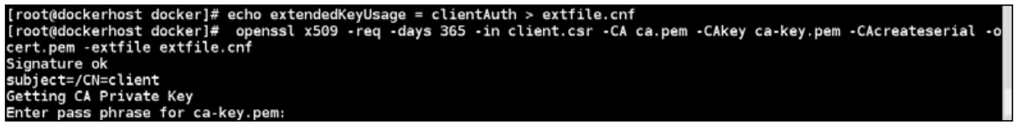


```
[root@dockerhost docker]# openssl genrsa -out key.pem 2048
Generating RSA private key, 2048 bit long modulus
....................................+++
..................+++
e is 65537 (0x10001)
[root@dockerhost docker]# openssl req -subj '/CN=client' -new -key key.pem -out client.csr
```

6. To make the key suitable for client authentication, create an extensions configuration file and sign the public key:

   ```
   $ echo extendedKeyUsage = clientAuth > extfile_client.cnf
   ```

   ```
   $ openssl x509 -req -days 365 -in client.csr -CA ca.pem -CAkey
   ca-key.pem  -CAcreateserial -out cert.pem -extfile_client.cnf
   ```



```
[root@dockerhost docker]# echo extendedKeyUsage = clientAuth > extfile.cnf
[root@dockerhost docker]#  openssl x509 -req -days 365 -in client.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -o
cert.pem -extfile extfile.cnf
Signature ok
subject=/CN=client
Getting CA Private Key
Enter pass phrase for ca-key.pem:
```

7. After generating `cert.pem` and `server-cert.pem`, we can safely remove both the certificate signing requests:

   ```
   $ rm -rf client.csr server.csr
   ```

8. To set tight security and protect keys from accidental damage, let's change the permissions:

   ```
   $ chmod -v 0600 ca-key.pem key.pem server-key.pem ca.pem server-
   cert.pem cert.pem
   ```

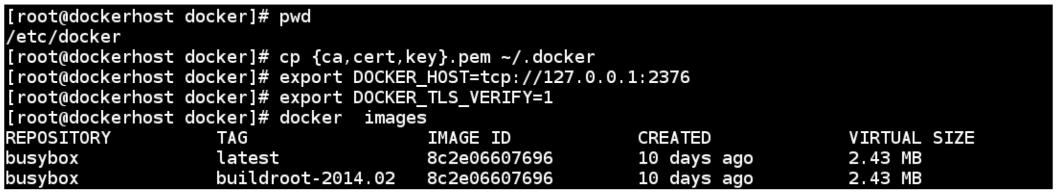
9. Stop the daemon if it is running on `dockerhost.example.com`. Then, start the Docker daemon manually from `/etc/docker`:

   ```
    $ pwd
      /etc/docker
       $ docker -d --tlsverify --tlscacert=ca.pem --
   tlscert=server-cert.pem    --tlskey=server-key.pem   -
   H=0.0.0.0:2376
   ```

10. From another terminal, go to `/etc/docker`. Run the following command to connect to the Docker daemon:

    ```
    $ cd /etc/docker
    $ docker --tlsverify --tlscacert=ca.pem --tlscert=cert.pem --
    tlskey=key.pem -H=127.0.0.1:2376 version
    ```

    You will see that a TLS connection is established and you can run the commands over it. You can also put the CA public key and the client's TLS certificate and key in the `.docker` folder in the home directory of the user and use the `DOCKER_HOST` and `DOCKER_TLS_VERIFY` environment variables to make a secure connection by default.



```
[root@dockerhost docker]# pwd
/etc/docker
[root@dockerhost docker]# cp {ca,cert,key}.pem ~/.docker
[root@dockerhost docker]# export DOCKER_HOST=tcp://127.0.0.1:2376
[root@dockerhost docker]# export DOCKER_TLS_VERIFY=1
[root@dockerhost docker]# docker  images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
busybox             latest              8c2e06607696        10 days ago         2.43 MB
busybox             buildroot-2014.02   8c2e06607696        10 days ago         2.43 MB
```

11. To connect from the remote host we mentioned while signing the server key with our CA, we will need to copy the CA public key and the client's TLS certificate and key to the remote machine and then connect to the Docker host as shown in the preceding screenshot.

How it works...

We setup the TLS connection between the Docker daemon and the client for a secure communication.

There's more...

- To set up the Docker daemon to start with the TLS configuration by default, we will need to update the Docker configuration file. For example, on Fedora, you update the `OPTIONS` parameter as follows in `/etc/sysconfig/docker`:

  ```
  OPTIONS='--selinux-enabled -H tcp://0.0.0.0:2376 --tlsverify
  --tlscacert=/etc/docker/ca.pem --tlscert=/etc/docker/server-
  cert.pem --tlskey=/etc/docker/server-key.pem'
  ```

- If you recall, in *Chapter 1*, *Introduction and Installation*, we saw how we can set up the Docker host using the Docker Machine (`http://docs.docker.com/machine/`) and as part of this setup, the TLS setup happens between the client and the host running the Docker daemon. After configuring the Docker host with the Docker Machine, check `.docker/machine` for the user on the client system.

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 and 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 the different use cases of Docker, and in *Chapter 6*, *Docker APIs and Language Bindings*, we looked at how to use remote APIs to connect to a remote Docker host.

Ease of use is all good, but before going into production, performance is one of the key aspects that is considered. In this chapter, we'll see the performance impacting features of Docker 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 look at the approach you can follow to do performance evaluation rather than performance numbers collected from runs to do 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:

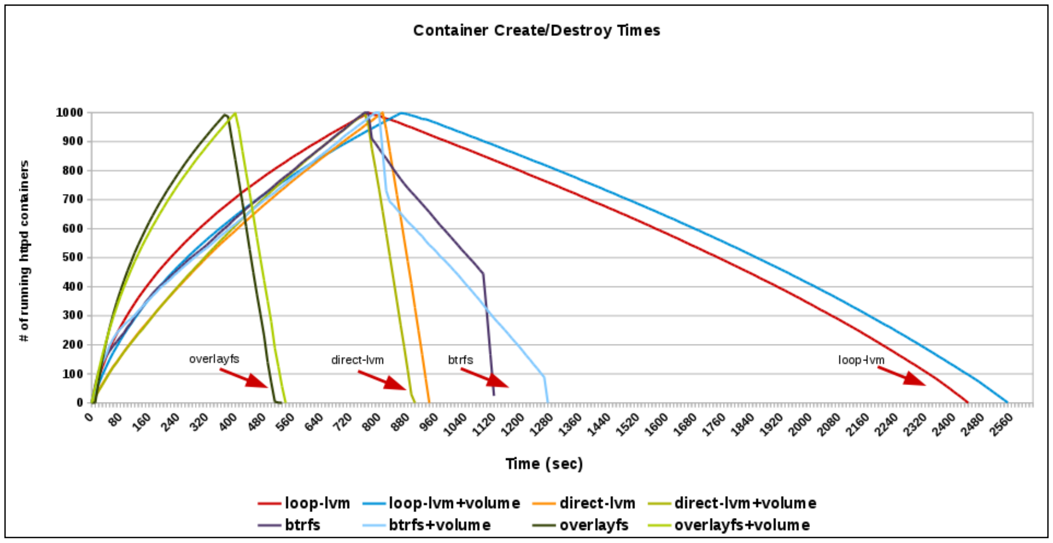
- **Volumes**: While putting down any enterprise class workload, you would like to tune the underlying storage accordingly. You should not use the primary/root filesystem used by containers to store data. Docker provides the facility to attach/mount external storage through volumes. As we have seen in *Chapter 4*, *Network and Data Management for Containers*, there are two types of volumes, which are as follows:
    - Volumes that are mounted through host machines 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*, *Installation and Introduction*, which are vfs, aufs, btrfs, devicemapper, and overlayFS. Support for ZFS has been merged recently as well. You can check the currently supported storage drivers and their priority of selection if nothing is chosen as the Docker start time at `https://github.com/docker/docker/blob/master/daemon/graphdriver/driver.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/docker/docker/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 across service restart. For Fedora/RHEL/CentOS, you will have the update `OPTIONS` field in `/etc/sysconfig/docker`. Something like the following to use the `btrfs` backend:

    ```
    OPTIONS=-s btrfs
    ```

The following graph shows you how much time it takes to start and stop 1,000 containers with different configurations of storage driver:



http://developerblog.redhat.com/2014/09/30/overview-storage-scalability-docker/

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 host networking stack to the container by skipping the creation of a network namespace for the container. From this, it is clear that this option always gives better performance compared to the bridged one.

  This has some limitations, such as not being able to 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 and accordingly the resource will be shared. Consider the following example:

    ```
    $ docker run -it -c 100 fedora bash
    ```

  - **CPUsets**: This allows you to create CPU masks, using which execution of threads inside a container on host CPUs is controlled. For example, the following code will run threads inside a container on the 0th and 3rd core:

    ```
    $ docker run -it  --cpuset=0,3 fedora bash
    ```

- **Memory limits**: We can set memory limits to a container. For example, the following command will limit the memory usage to 512 MB for the container:

  ```
  $ docker run -it -m 512M fedora bash
  ```

- **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. With Docker 1.6 (`https://docs.docker.com/v1.6/release-notes/`) and above we can change the `ulimit` settings with the following command:

  ```
  $ docker run -it --ulimit data=8192 fedora bash
  ```

  The preceding command will change the settings for just that given container, it is a per container tuning variable. We can also set some of these settings through the systemd configuration file of Docker daemon, which will be applicable to all containers by default. For example, looking at the systemd configuration file for Docker on Fedora, you will see something like the following in the service section:

  ```
  LimitNOFILE=1048576  # Open file descriptor setting
  LimitNPROC=1048576   # Number of processes settings
  LimitCORE=infinity   # Core size settings
  ```

  You can update this as per your need.

You can learn about Docker performance by studying the work done by others. Over the last year, some Docker performance-related studies have been published by a few companies:

- From Red Hat:
  - Performance Analysis of Docker on Red Hat Enterprise Linux:

    `http://developerblog.redhat.com/2014/08/19/performance-analysis-docker-red-hat-enterprise-linux-7/`

    `https://github.com/jeremyeder/docker-performance`

  - Comprehensive Overview of Storage Scalability in Docker:

    `http://developerblog.redhat.com/2014/09/30/overview-storage-scalability-docker/`

  - Beyond Microbenchmarks—breakthrough container performance with Tesla efficiency:

    `http://developerblog.redhat.com/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 similar workload on different environments (bare metal/VM/Docker) and then collect the results with the help of different performance stats. To simplify things, we can write common benchmark scripts which can be used to 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 is listed earlier (`https://github.com/jeremyeder/docker-performance/blob/master/Dockerfiles/Dockerfile`), the author has used a Dockerfile to create a CentOS image and used the `container` environment variable to select Docker and non-Docker environment for benchmark script `run-sysbench.sh`.

Similarly, Dockerfiles and related scripts are published by IBM for their study 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 of 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 the benchmark on bare metal and inside the container. Similar steps can be performed in other environments, as mentioned earlier.

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 GitHub repository that we referred to earlier:

```
$ git clone https://github.com/jeremyeder/docker-performance.git
$ cd docker-performance/Dockerfiles/
$ docker build -t c7perf --rm=true - < Dockerfile
$ docker images
REPOSITORY           TAG            IMAGE ID          CREATED
VIRTUAL SIZE
c7perf              latest         59a10df39a82    About a minute ago
678.3 MB
```

How to do it...

Inside the same GitHub repository, we have a script to run sysbench, `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, run the following commands:

   ```
   $ cd docker-performance/bench/sysbench
   $ export container=no
   $ sh ./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.

2. To run the benchmark inside the container, we need to first start the container and then run the benchmark script:

   ```
   $ mkdir /results_container
   $ docker run -it -v /results_container:/results c7perf bash
   $ docker-performance/bench/sysbench/run-sysbench.sh cpu test1
   ```

   As we mounted the host directory, `/results_container`, inside the `/results` container, the result will be collected on the host.

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 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 the 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 in 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`) 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

In the 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 and on VM it can be mounted using the virtual disk driver or we can do device pass through. On Docker, we can attach the filesystem from the host machine using Docker volumes.

Prepare the workload file. We can pick `https://github.com/thewmf/kvm-docker-comparison/blob/master/fio/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
```

Using the preceding job file, we can do random direct I/O 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 is 8. It is a mix workload, which does 70 percent read and 30 percent write.

How to do it...

1. For the bare metal and VM tests, you can just run the FIO job file and collect the result:

   ```
   $ fio mixed.fio
   ```

2. For the Docker test, you can prepare a Docker file as follows:

   ```
   FROM ubuntu
   MAINTAINER nkhare@example.com
   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 build -t docker_fio_perf .
   ```

4. Start the container as follows to run the benchmark and collect the results:

   ```
   $ docker run --rm -v /ferrari:/ferrari docker_fio_perf
   mixed.fio
   ```

5. While running the preceding test on Fedora/RHEL/CentOS, where SELinux is enabled, you will get the `Permission denied` error. To fix it, re-label the host directory while mounting it inside the container as follows:

   ```
   $ docker 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 the result comparison. You can even try out different kinds of I/O patterns using the job file and get the desired result.

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

Network is one of the key aspects to consider while deploying the applications in the container environment. To do performance comparison with bare metal, VM and containers, we have to consider different scenarios as follows:

- Bare metal to bare metal
- VM to VM
- Docker container to container with the default networking mode (bridge)
- Docker container to container with host net (`--net=host`)
- Docker container running inside 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` (`http://netperf.org/netperf/`) to measure the network bandwidth and request/response, respectively.

Getting ready

Make sure both the endpoints can reach each other and have the necessary packages/software installed. On Fedora 21, you can install `nuttcp` with the following command:

```
$ yum install -y nuttcp
```

And, get `netperf` from its website.

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:

   ```
   $ nuttcp -t <SERVER_IP>
   ```

3. Measure the receiver throughput on the client (server to client) with the following command:

   ```
   $ nuttcp -r <SERVER_IP>
   ```

4. To run the request/response benchmark using `netperf`, perform the following steps:
5. Start `netserver` on one endpoint:

   ```
   $ netserver
   ```

6. Connect to the server from the other endpoint and run the request/response test:
   - For TCP:

     ```
     $ netperf  -H 172.17.0.6 -t TCP_RR
     ```

   - For UDP:

     ```
     $ netperf  -H 172.17.0.6 -t UDP_RR
     ```

How it works...

In both 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

With the release of version 1.5, Docker added a feature to get container resource usage from in-built commands.

Getting ready

A Docker host with version 1.5 or later installed, which can be accessed via the Docker client. Also, start a few containers to get stats.

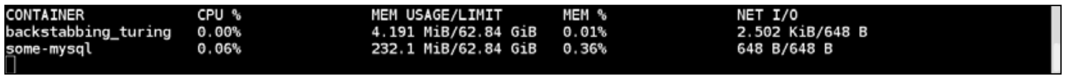
How to do it...

1. Run the following command to get stats from one or more containers:

   ```
   $ docker stats [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:

   ```
   $ 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 the resource information from the Cgroups and serves it through the APIs.

See also

- Refer to the release notes of Docker 1.5 at `https://docs.docker.com/v1.5/release-notes/`

Setting up performance monitoring

We have tools such as SNMP, Nagios, and so on to monitor bare metal and VM performance. Similarly, there are a few tools/plugins available to monitor container performance such as cAdvisor (`https://github.com/google/cadvisor`) and sFlow (`http://blog.sflow.com/2014/06/docker-performance-monitoring.html`). In this recipe, let's see how we can configure cAdvisor.

Getting ready

Setting up cAdvisor.

- The easiest way to run cAdvisor is to run its Docker container, which can be done with the following command:

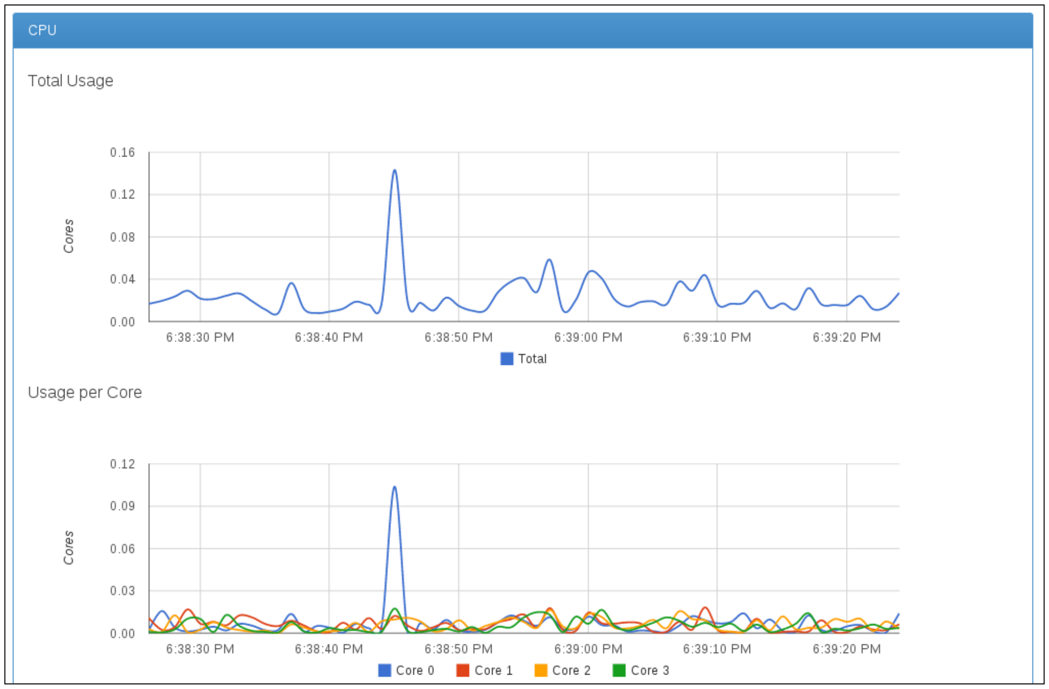
```
sudo docker run \
  --volume=/:/rootfs:ro \
  --volume=/var/run:/var/run:rw \
  --volume=/sys:/sys:ro \
  --volume=/var/lib/docker/:/var/lib/docker:ro \
  --publish=8080:8080 \
  --detach=true \
  --name=cadvisor \
  google/cadvisor:latest
```

- If you want to run cAdvisor outside Docker, then follow the instructions given on the cAdvisor home page at `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 the 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 the 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 is the screenshot of one such container:



How it works...

With the `docker run` command, we have mounted few volumes from host machines in read-only mode. cAdvisor will read the relevant information from those like the Cgroup details for containers and show them graphically.

There's more...

cAdvisor supports exporting the performance matrices to influxdb (`http://influxdb.com/`). Heapster (`https://github.com/GoogleCloudPlatform/heapster`) is another project from Google, which allows cluster-wide (Kubernetes) monitoring using cAdvisor.

See also

- You can look at the matrices used by cAdvisor from Cgroups in the documentation on the Docker website `https://docs.docker.com/articles/runmetrics/`

8
Docker Orchestration and Hosting Platforms

In this chapter, we will cover the following recipes:

- Running applications with Docker Compose
- Setting up Cluster with Docker Swarm
- Setting up CoreOS for Docker orchestration
- Setting up a Project Atomic host
- Doing atomic update/rollback with Project Atomic
- Adding more storage for Docker in Project Atomic
- Setting up Cockpit for Project Atomic
- Setting up a Kubernetes cluster
- Scaling up and down in a Kubernetes cluster
- Setting up WordPress with a Kubernetes cluster

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 an easy task. You have to orchestrate these containers. So, in this chapter, we'll cover some of the 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 and Docker Swarm (`https://docs.docker.com/swarm/`) to cluster multiple Docker hosts. Docker Compose was previously called Fig (`http://www.fig.sh/`).

CoreOS (`https://coreos.com/`) created etcd (`https://github.com/coreos/etcd`) for consensus and service discovery, fleet (`https://coreos.com/using-coreos/clustering`) to deploy containers in a cluster, and flannel (`https://github.com/coreos/flannel`) for overlay networking.

Google started Kubernetes (`http://kubernetes.io/`) for Docker orchestration. Kubernetes provides mechanisms for application deployment, scheduling, updating, maintenance, and scaling.

Red Hat launched a container-specific operating system called Project Atomic (`http://www.projectatomic.io/`), which can leverage the orchestration capabilities of Kubernetes.

Even Microsoft announced a specialized operating system for Docker (`http://azure.microsoft.com/blog/2015/04/08/microsoft-unveils-new-container-technologies-for-the-next-generation-cloud/`).

Apache Mesos (`http://mesos.apache.org/`), which provides resource management and scheduling across entire datacenter and cloud environments, also added support for Docker (`http://mesos.apache.org/documentation/latest/docker-containerizer/`).

VMware also launched the container-specific host VMware Photon (`http://vmware.github.io/photon/`).

This is definitely a very interesting space, but the policy management tools of many orchestration engines do not make the lives of developers and operators easy. They have to learn different tools and formats when they move from one platform to another. It would be great if we could have a standard way to build and launch composite, multicontainer apps. The Project Atomic community seems to be working on one such platform-neutral specification called Nulecule (`https://github.com/projectatomic/nulecule/`). A good description about Nulecule is available at `http://www.projectatomic.io/blog/2015/05/announcing-the-nulecule-specification-for-composite-applications/`:

> *"Nulecule defines a pattern and model for packaging complex multi-container applications, referencing all their dependencies, including orchestration metadata, in a single container image for building, deploying, monitoring, and active management. Just create a container with a Nulecule file and the app will 'just work'. In the Nulecule spec, you define orchestration providers, container locations and configuration parameters in a graph, and the Atomic App implementation will piece them together for you with the help of Providers. The Nulecule specification supports aggregation of multiple composite applications, and it's also container and orchestration agnostic, enabling the use of any container and orchestration engine."*

AtomicApp is a reference implementation (`https://github.com/projectatomic/atomicapp/`) of the Nulecule specification. It can be used to bootstrap container applications and to install and run them. AtomicApp currently has a limited number of providers (Docker, Kubernetes, OpenShift), but support for others will be added soon.

On a related note, the CentOS community is building a CI environment, which will take advantage of Nulecule and AtomicApp. For further information, visit `http://wiki.centos.org/ContainerPipeline`.

All of the preceding tools and platforms need separate chapters for themselves. In this chapter, we'll explore Compose, Swarm, CoreOS, Project Atomic, and Kubernetes briefly.

Running applications with Docker Compose

Docker Compose (`http://docs.docker.com/compose/`) is the native Docker tool to run the 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. At the time of writing, Compose is still not production-ready. In this recipe, we'll once again use WordPress as a sample application to run.

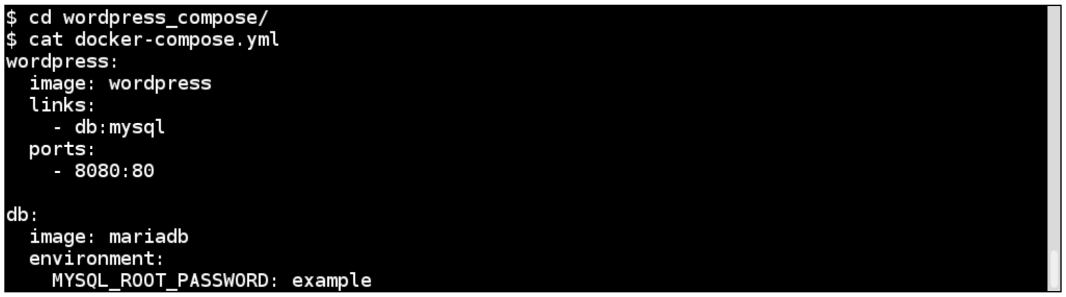
Getting ready

Make sure you have Docker Version 1.3 or later installed on the system. To install Docker Compose, run the following command:

```
$ sudo pip install docker-compose
```

How to do it...

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
wordpress:
  image: wordpress
  links:
    - db:mysql
  ports:
    - 8080:80

db:
  image: mariadb
  environment:
    MYSQL_ROOT_PASSWORD: example
```

2. We have taken the preceding example from the official WordPress Docker repo on Docker Hub (`https://registry.hub.docker.com/_/wordpress/`).

3. Within the app directory, run the following command to build the app:

   ```
   $ docker-compose up
   ```

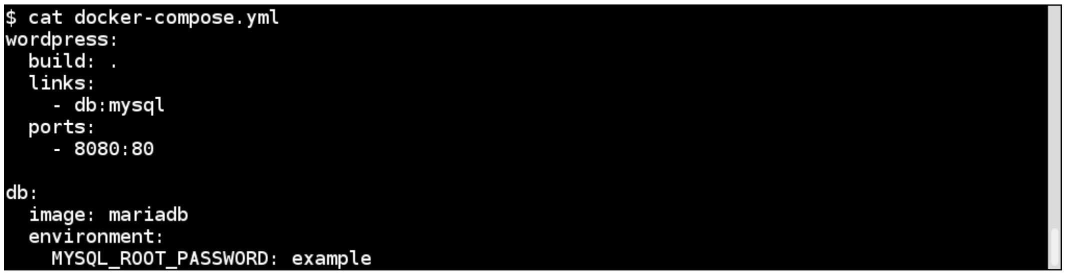
4. Once the build is complete, access the WordPress installation page from `http://localhost:8080` or `http://<host-ip>:8080`.

How it works...

Docker Compose downloads both the `mariadb` `wordpress` images, if not available locally from the official Docker registry. First, it starts the `db` container from the `mariadb` image; then it starts the `wordpress` container. Next, it links with the `db` container and exports the port to the host machine.

There's more...

We can even build images from the Dockerfile during the compose and then use it for the app. For example, to build the `wordpress` image, we can get the corresponding Dockerfile and other supporting file from within the application's Compose directory and update the `docker-compose.yml` file in a similar manner as follows:



```
$ cat docker-compose.yml
wordpress:
  build: .
  links:
    - db:mysql
  ports:
    - 8080:80

db:
  image: mariadb
  environment:
    MYSQL_ROOT_PASSWORD: example
```

We can start, stop, rebuild, and get the status of the app. Visit its documentation on the Docker website.

See also

- The Docker Compose YAML file reference at `http://docs.docker.com/compose/yml/`
- The Docker Compose command-line reference at `http://docs.docker.com/compose/cli/`
- The Docker Compose GitHub repository at `https://github.com/docker/compose`

Setting up cluster with Docker Swarm

Docker Swarm (`http://docs.docker.com/swarm/`) is native clustering to Docker. It groups multiple Docker hosts into a single pool in which one can launch containers. In this recipe, we'll use Docker Machine (`http://docs.docker.com/machine/`) to set up a Swarm cluster. At the time of writing, Swarm is still not production-ready. If you recall, we used Docker Machine to set up a Docker host on Google Compute Engine in *Chapter 1*, *Introduction and Installation*. To keep things simple, here we'll use VirtualBox as the backend for Docker Machine to configure hosts.

Getting ready

1. Install VirtualBox on your system (`https://www.virtualbox.org/`). Instructions to configure VirtualBox are outside the scope of this book.
2. Download and set up Docker Machine. In Fedora x86_64, run the following commands:

```
$ wget
https://github.com/docker/machine/releases/download/v0.2.0/doc
ker-machine_linux-amd64
$ sudo mv  docker-machine_linux-amd64 /usr/local/bin/docker-
machine
$ chmod a+x  /usr/local/bin/docker-machine
```

How to do it...

1. Using the Swarm discovery service, we first need to create a Swarm token to identify our cluster uniquely. Other than the default hosted discovery service, Swarm supports different types of discovery services such as etcd, consul, and zookeeper. For more details, please visit `https://docs.docker.com/swarm/discovery/`. To create a token using the default hosted discovery service, we'll first set up the Docker host using Docker Machine on a VM and then get the token:

```
$ docker-machine create -d virtualbox local
```

2. To access the Docker we just created from your local Docker client, run the following command:

```
$ eval "$(docker-machine env local)"
```

3. To get the token, run the following command:

```
$ docker run swarm create
7c3a21b42708cde81d99884116d68fa1
```

4. Using the token created in the preceding step, set up Swarm master:

   ```
   $ docker-machine create  -d virtualbox  --swarm  --swarm-
   master  --swarm-discovery
   token://7c3a21b42708cde81d99884116d68fa1  swarm-master
   ```

5. Similarly, let's create two Swarm nodes:

   ```
   $ docker-machine create -d virtualbox  --swarm  --swarm-
   discovery token://7c3a21b42708cde81d99884116d68fa1 swarm-node-
   1
   ```

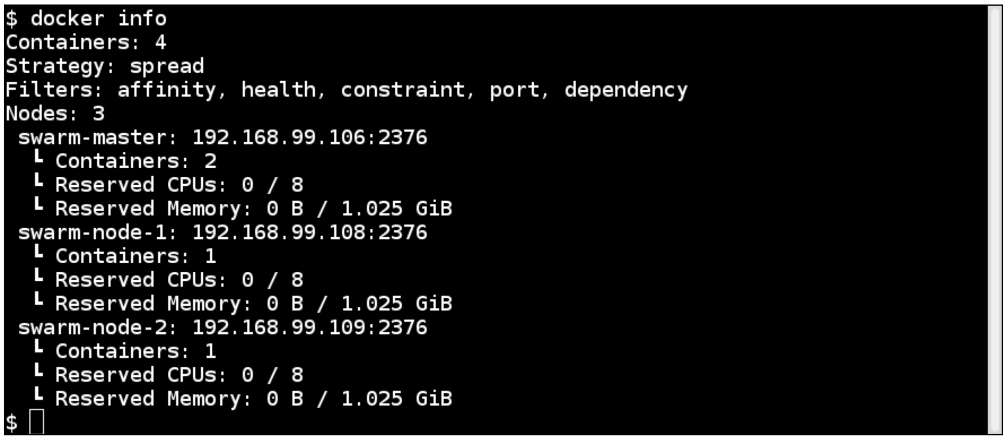
   ```
   $ docker-machine create -d virtualbox  --swarm  --swarm-
   discovery token://7c3a21b42708cde81d99884116d68fa1 swarm-node-
   2
   ```

6. Now, connect to Docker Swarm from your local Docker client:

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

7. Swarm APIs are compatible with Docker client APIs. Let's run the `docker info` command to see Swarm's current configuration/setup:

   ```
   $ docker info
   ```



```
$ docker info
Containers: 4
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 3
 swarm-master: 192.168.99.106:2376
  └ Containers: 2
  └ Reserved CPUs: 0 / 8
  └ Reserved Memory: 0 B / 1.025 GiB
 swarm-node-1: 192.168.99.108:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 8
  └ Reserved Memory: 0 B / 1.025 GiB
 swarm-node-2: 192.168.99.109:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 8
  └ Reserved Memory: 0 B / 1.025 GiB
$ 
```

As you can see, we have three nodes in the cluster: one master and two nodes.

How it works...

Using the unique token we got from the hosted discovery service, we registered the master and nodes in a cluster.

There's more...

- In the preceding `docker info` output, we also scheduled policy (strategy) and filters. More information on these can be found at `https://docs.docker.com/swarm/scheduler/strategy/` and `https://docs.docker.com/swarm/scheduler/filter/`. These define where the container will run.
- There is active development happening to integrate Docker Swarm and Docker Compose so that we point and compose the app to the Swarm cluster. The app will then start on the cluster. Visit `https://github.com/docker/compose/blob/master/SWARM.md`

See also

- The Swarm documentation on the Docker website at `https://docs.docker.com/swarm/`
- Swarm's GitHub repository at `https://github.com/docker/swarm`

Setting up CoreOS for Docker orchestration

CoreOS (`https://coreos.com/`) is a Linux distribution that has been rearchitected to provide the features needed to run modern infrastructure stacks. It is Apache 2.0 Licensed. It has a product called CoreOS Managed Linux (`https://coreos.com/products/managed-linux/`) for which the CoreOS team provides commercial support.

Essentially, CoreOS provides platforms to host a complete applications stack. We can set up CoreOS on different cloud providers, bare metal, and in the VM environment. Let's look at the building blocks of CoreOS:

- etcd
- Container runtime
- Systemd
- Fleet

Let's discuss each in detail:

- **etcd**: From the GitHub page of etcd (`https://github.com/coreos/etcd/#etcd`). `etcd` is a highly available key-value store for shared configuration and service discovery. It is inspired by Apache ZooKeeper and doozer with a focus on being:
    - **Simple**: Curl-able user-facing API (HTTP plus JSON)
    - **Secure**: Optional SSL client certificate authentication

- **Fast**: Benchmark of 1,000s of writes per instance
- **Reliable**: Proper distribution using Raft

It is written in Go and uses the Raft consensus algorithm (`https://raftconsensus.github.io/`) to manage a highly available replicated log. etcd can be used independent of CoreOS. We can:

- Set up a single or multinode cluster. More information on this can be found at `https://github.com/coreos/etcd/blob/master/Documentation/clustering.md`.
- Access using CURL and different libraries, found at `https://github.com/coreos/etcd/blob/master/Documentation/libraries-and-tools.md`.

In CoreOS, `etcd` is meant for the coordination of clusters. It provides a mechanism to store configurations and information about services in a consistent way.

- **Container runtime**: CoreOS supports Docker as a container runtime environment. In December 2014, CoreOS announced a new container runtime Rocket (`https://coreos.com/blog/rocket/`). Let's restrict our discussion to Docker, which is currently installed on all CoreOS machines.
- **systemd**: `systemd` is an init system used to start, stop, and manage processes. In CoreOS, it is used to:
    - Launch Docker containers
    - Register services launched by containers to etcd

  Systemd manages unit files. A sample unit file looks like the following:

```
[Unit]
Description=Docker Application Container Engine
Documentation=http://docs.docker.com
After=network.target docker.socket
Requires=docker.socket

[Service]
Type=notify
EnvironmentFile=-/etc/sysconfig/docker
EnvironmentFile=-/etc/sysconfig/docker-storage
ExecStart=/usr/bin/docker -d -H fd:// $OPTIONS
$DOCKER_STORAGE_OPTIONS
LimitNOFILE=1048576
LimitNPROC=1048576

[Install]
WantedBy=multi-user.target
```

This unit file starts the Docker daemon with the command mentioned in `ExecStart` on Fedora 21. The Docker daemon will start after the `network target` and `docker socket` services. `docker socket` is a prerequisite for the Docker daemon to start. Systemd targets are ways to group processes so that they can start at the same time. `multi-user` is one of the targets with which the preceding unit file is registered. For more details, you can look at the upstream documentation of Systemd at `http://www.freedesktop.org/wiki/Software/systemd/`.

- **Fleet**: Fleet (`https://coreos.com/using-coreos/clustering/`) is the cluster manager that controls `systemd` at the cluster level. systemd unit files are combined with some Fleet-specific properties to achieve the goal. From the Fleet documentation (`https://github.com/coreos/fleet/blob/master/Documentation/architecture.md`):

  > *"Every system in the fleet cluster runs a single* `fleetd` *daemon. Each daemon encapsulates two roles: the engine and the agent. An engine primarily makes scheduling decisions while an agent executes units. Both the engine and agent use the reconciliation model, periodically generating a snapshot of 'current state' and 'desired state' and doing the necessary work to mutate the former towards the latter."*

  `etcd` is the sole datastore in a `fleet` cluster. All persistent and ephemeral data is stored in `etcd`; unit files, cluster presence, unit state, and so on. `etcd` is also used for all internal communication between fleet engines and agents.

Now we know of all the building blocks of CoreOS. Let's try out CoreOS on our local system/laptop. To keep things simple, we will use Vagrant to set up the environment.

Getting ready

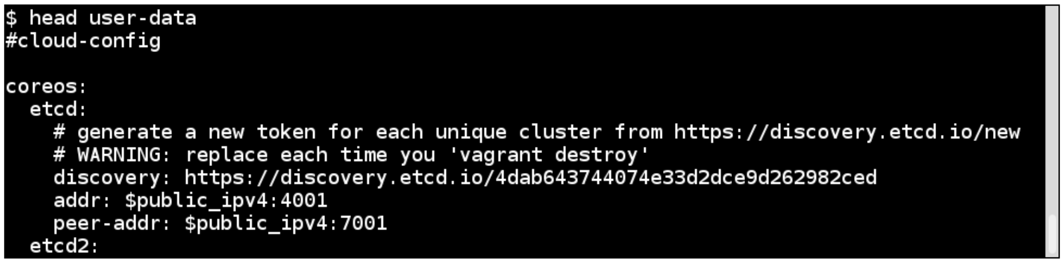
1. Install VirtualBox on the system (`https://www.virtualbox.org/`) and Vagrant (`https://www.vagrantup.com/`). The instructions to configure both of these things are outside the scope of this book.
2. Clone the `coreos-vagrant` repository:

   ```
   $ git clone https://github.com/coreos/coreos-vagrant.git
   $ cd coreos-vagrant
   ```

3. Copy the sample file `user-data.sample` to `user-data` and set up the token to bootstrap the cluster:

   ```
   $ cp user-data.sample user-data
   ```

4. When we configure the CoreOS cluster with more than one node, we need a token to bootstrap the cluster to select the initial etcd leader. This service is provided free by the CoreOS team. We just need to open `https://discovery.etcd.io/new` in the browser to get the token and update it within the `user-data` file as follows:



```
$ head user-data
#cloud-config

coreos:
  etcd:
    # generate a new token for each unique cluster from https://discovery.etcd.io/new
    # WARNING: replace each time you 'vagrant destroy'
    discovery: https://discovery.etcd.io/4dab643744074e33d2dce9d262982ced
    addr: $public_ipv4:4001
    peer-addr: $public_ipv4:7001
  etcd2:
```

5. Copy `config.rb.sample` to `config.rb` and make changes to the following line:

   `$num_instances=1`

   It should now look like this:

   `$num_instances=3`

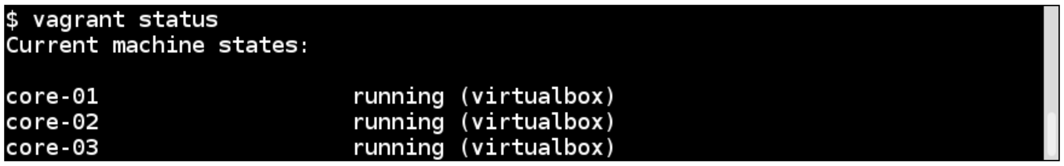
This will ask Vagrant to set up three node clusters. By default, Vagrant is configured to get the VM images from the alpha release. We can change it to beta or stable by updating the `$update_channel` parameter in Vagrantfile. For this recipe, I chose stable.

How to do it...

1. Run the following command to set up the cluster:

   `$ vagrant up`

   Now, check the status, using the command shown in the following screenshot:



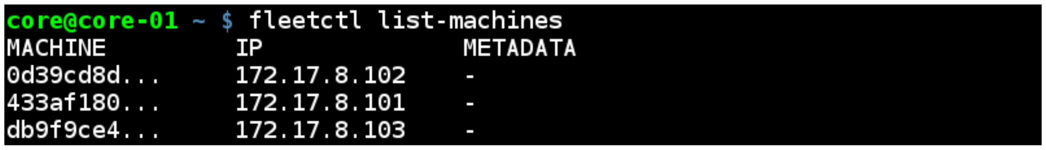
```
$ vagrant status
Current machine states:

core-01                   running (virtualbox)
core-02                   running (virtualbox)
core-03                   running (virtualbox)
```

2. Log in to one of the VMs using SSH, look at the status of services, and list the machines in the cluster:

   `$ vagrant ssh core-01`

   `$ systemctl status etcd fleet`

   `$ fleetctl list-machines`



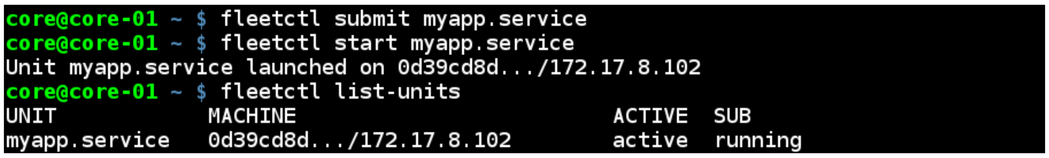
```
core@core-01 ~ $ fleetctl list-machines
MACHINE         IP              METADATA
0d39cd8d...     172.17.8.102    -
433af180...     172.17.8.101    -
db9f9ce4...     172.17.8.103    -
```

3. Create a service unit file called `myapp.service` with the following content:

```
[Unit]
Description=MyApp
After=docker.service
Requires=docker.service

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill busybox1
ExecStartPre=-/usr/bin/docker rm busybox1
ExecStartPre=/usr/bin/docker pull busybox
ExecStart=/usr/bin/docker run --name busybox1 busybox /bin/sh -c
"while true; do echo Hello World; sleep 1; done"
ExecStop=/usr/bin/docker stop busybox1
```

4. Let's now submit the service for scheduling and start the service:

```
$ fleetctl submit myapp.service
$ fleetctl start myapp.service
$ fleetctl list-units
```



```
core@core-01 ~ $ fleetctl submit myapp.service
core@core-01 ~ $ fleetctl start myapp.service
Unit myapp.service launched on 0d39cd8d.../172.17.8.102
core@core-01 ~ $ fleetctl list-units
UNIT            MACHINE                         ACTIVE  SUB
myapp.service   0d39cd8d.../172.17.8.102        active  running
```

As we can see, our service has started on one of the nodes in the cluster.

How it works...

Vagrant uses the cloud configuration file (`user-data`) to boot the VMs. As they have the same token to bootstrap the cluster, they select the leader and start operating. Then, with `fleetctl`, which is the fleet cluster management tool, we submit the unit file for scheduling, which starts on one of the nodes.

There's more...

- Using the cloud configuration file in this recipe, we can start `etcd` and `fleet` on all the VMs. We can choose to run `etcd` just on selected nodes and then configure worker nodes running `fleet` to connect to etcd servers. This can be done by setting the cloud configuration file accordingly. For more information, please visit `https://coreos.com/docs/cluster-management/setup/cluster-architectures/`.
- With `fleet`, we can configure services for high availability. For more information, take a look at `https://coreos.com/docs/launching-containers/launching/fleet-unit-files/`.
- Though your service is running on the host, you will not be able to reach it from the outside world. You will need to add some kind of router and wildcard DNS configuration to reach your service from the outside world.

See also

- The CoreOS documentation for more details at `https://coreos.com/docs/`
- The visualization of RAFT consensus algorithm at `http://thesecretlivesofdata.com/raft`
- How to configure the cloud config file at `https://coreos.com/docs/cluster-management/setup/cloudinit-cloud-config/` and `https://coreos.com/validate/`
- Documentation on systemd at `https://coreos.com/docs/launching-containers/launching/getting-started-with-systemd/`
- How to launch containers with fleet at `https://coreos.com/docs/launching-containers/launching/launching-containers-fleet/`

Setting up a Project Atomic host

Project Atomic facilitates application-centric IT architecture by providing an end-to-end solution to deploy containerized applications quickly and reliably, with atomic update and rollback for the application and host alike.

This is achieved by running applications in containers on a Project Atomic host, which is a lightweight operating system specially designed to run containers. The hosts can be based on Fedora, CentOS, or Red Hat Enterprise Linux.

Next, we will elaborate on the building blocks of the Project Atomic host.

- **OSTree and rpm-OSTree**: OSTree (`https://wiki.gnome.org/action/show/Projects/OSTree`) is a tool to manage bootable, immutable, and versioned filesystem trees. Using this, we can build client-server architecture in which the server hosts an OSTree repository and the client subscribed to it can incrementally replicate the content.

  rpm-OSTree is a system to decompose RPMs on the server side into the OSTree repository to which the client can subscribe and perform updates. With each update, a new root is created, which is used for the next reboot. During updates, `/etc` is rebased and `/var` is untouched.

- **Container runtime**: As of now Project Atomic only supports Docker as container runtime.
- **systemd**: As we saw in earlier recipes, systemd is a new init system. It also helps to set up SELinux policies to containers for complete multitenant security and to control Cgroups policies, which we looked in at *Chapter 1*, *Introduction and Installation*.

  Project Atomic uses Kubernetes (`http://kubernetes.io/`) for application deployment over clusters of container hosts. Project Atomic can be installed on bare metal, cloud providers, VMs, and so on. In this recipe, let's see how we can install it on a VM using virt-manager on Fedora.

Getting ready

1. Download the image:

   ```
   $ wget
   http://download.fedoraproject.org/pub/fedora/linux/releases/te
   st/22_Beta/Cloud/x86_64/Images/Fedora-Cloud-Atomic-22_Beta-
   20150415.x86_64.raw.xz
   ```

   I have downloaded the beta image for Fedora 22 Cloud image *For Containers*. You should look for the latest cloud image *For Containers* at `https://getfedora.org/en/cloud/download/`.

2. Uncompress this image by using the following command:

   ```
   $ xz -d Fedora-Cloud-Atomic-22_Beta-20150415.x86_64.raw.xz
   ```

How to do it...

1. We downloaded the cloud image that does not have any password set for the default user `fedora`. While booting the VM, we have to provide a cloud configuration file through which we can customize the VM. To do this, we need to create two files, `meta-data` and `user-data`, as follows:

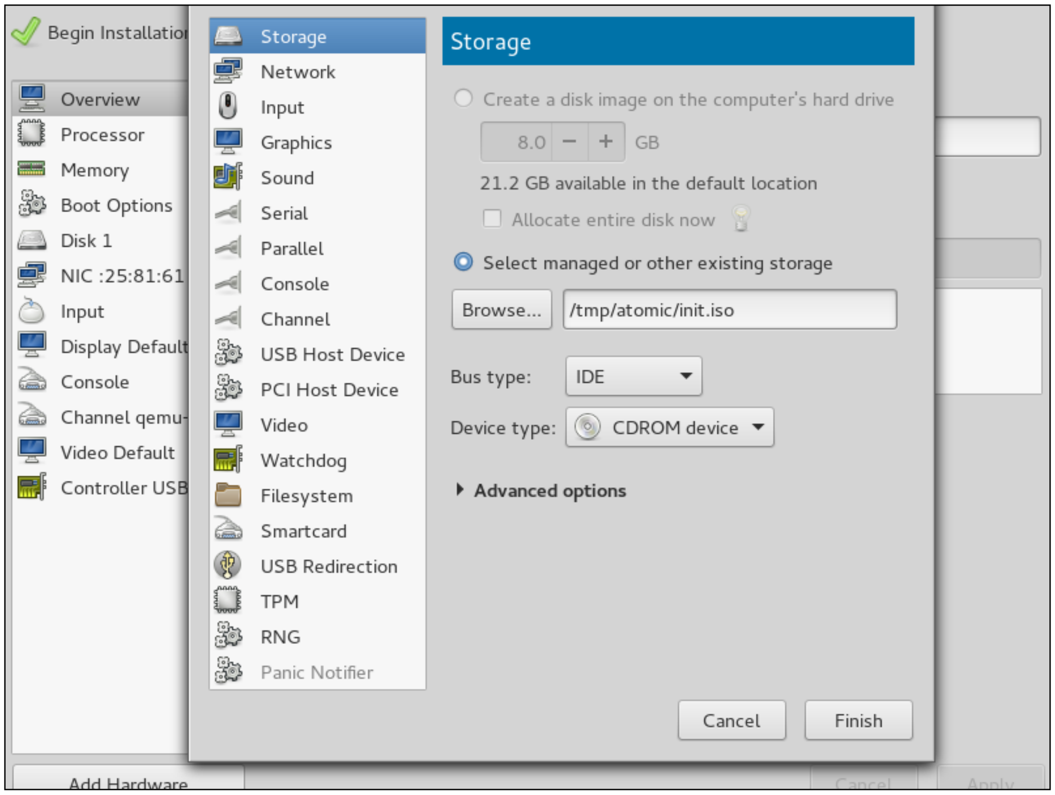
   ```
   $ cat  meta-data
   instance-id: iid-local01
   local-hostname: atomichost

   $ cat user-data
   #cloud-config
   password: atomic
   ssh_pwauth: True
   chpasswd: { expire: False }

   ssh_authorized_keys:
   - ssh-rsa AAAAB3NzaC1yc.........
   ```

   In the preceding code, we need to provide the complete SSH public key. We then need to create an ISO image consisting of these files, which we will use to boot to the VM. As we are using a cloud image, our setting will be applied to the VM during the boot process. This means the hostname will be set to `atomichost`, the password will be set to `atomic`, and so on. To create the ISO, run the following command:

   ```
   $ genisoimage -output init.iso -volid cidata -joliet -rock
   user-data meta-data
   ```

2. Start virt-manager.
3. Select **New Virtual Machine** and then import the existing disk image. Enter the image path of the Project Atomic image we downloaded earlier. Select **OS type** as **Linux** and **Version** as **Fedora 20/Fedora 21 (or later)**, and click on **Forward**. Next, assign CPU and Memory and click on **Forward**. Then, give a name to the VM and select **Customize configuration** before install. Finally, click on **Finish** and review the details.

4. Next, click on **Add Hardware**, and after selecting **Storage**, attach the ISO (`init.iso`) file we created to the VM and select **Begin Installation**:



Create a disk image on the computer's hard drive

8.0 GB

21.2 GB available in the default location

Allocate entire disk now

Select managed or other existing storage

Browse... /tmp/atomic/init.iso

Bus type: IDE

Device type: CDROM device

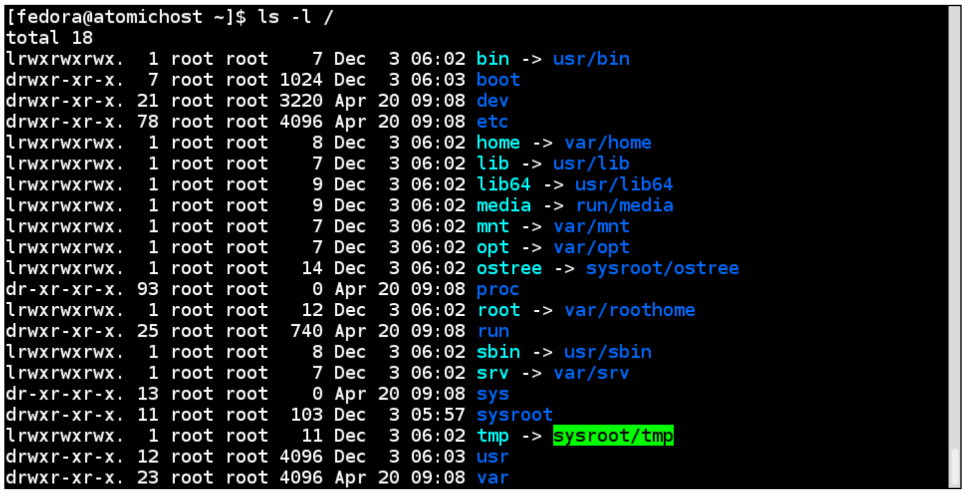
Once booted, you can see that its hostname is correctly set and you will be able to log in through the password given in the cloud init file. The default user is `fedora` and password is `atomic` as set in the `user-data` file.

How it works...

In this recipe, we took a Project Atomic Fedora cloud image and booted it using `virt-manager` after supplying the cloud init file.

There's more...

- After logging in, if you do file listing at `/`, you will see that most of the traditional directories are linked to `/var` because it is preserved across upgrades.



```
[fedora@atomichost ~]$ ls -l /
total 18
lrwxrwxrwx.  1 root root    7 Dec  3 06:02 bin -> usr/bin
drwxr-xr-x.  7 root root 1024 Dec  3 06:03 boot
drwxr-xr-x. 21 root root 3220 Apr 20 09:08 dev
drwxr-xr-x. 78 root root 4096 Apr 20 09:08 etc
lrwxrwxrwx.  1 root root    8 Dec  3 06:02 home -> var/home
lrwxrwxrwx.  1 root root    7 Dec  3 06:02 lib -> usr/lib
lrwxrwxrwx.  1 root root    9 Dec  3 06:02 lib64 -> usr/lib64
lrwxrwxrwx.  1 root root    9 Dec  3 06:02 media -> run/media
lrwxrwxrwx.  1 root root    7 Dec  3 06:02 mnt -> var/mnt
lrwxrwxrwx.  1 root root    7 Dec  3 06:02 opt -> var/opt
lrwxrwxrwx.  1 root root   14 Dec  3 06:02 ostree -> sysroot/ostree
dr-xr-xr-x. 93 root root    0 Apr 20 09:08 proc
lrwxrwxrwx.  1 root root   12 Dec  3 06:02 root -> var/roothome
drwxr-xr-x. 25 root root  740 Apr 20 09:08 run
lrwxrwxrwx.  1 root root    8 Dec  3 06:02 sbin -> usr/sbin
lrwxrwxrwx.  1 root root    7 Dec  3 06:02 srv -> var/srv
dr-xr-xr-x. 13 root root    0 Apr 20 09:08 sys
drwxr-xr-x. 11 root root  103 Dec  3 05:57 sysroot
lrwxrwxrwx.  1 root root   11 Dec  3 06:02 tmp -> sysroot/tmp
drwxr-xr-x. 12 root root 4096 Dec  3 06:03 usr
drwxr-xr-x. 23 root root 4096 Apr 20 09:08 var
```

- After logging in, you can run the Docker command as usual:

  `$sudo docker run -it fedora bash`

See also

- The virtual manager documentation at `https://virt-manager.org/documentation/`
- More information on package systems, image systems, and RPM-OSTree at `https://github.com/projectatomic/rpm-ostree/blob/master/doc/background.md`
- The quick-start guide on the Project Atomic website at `http://www.projectatomic.io/docs/quickstart/`
- The resources on cloud images at `https://www.technovelty.org//linux/running-cloud-images-locally.html` and `http://cloudinit.readthedocs.org/en/latest/`
- How to set up Kubernetes with an Atomic host at `http://www.projectatomic.io/blog/2014/11/testing-kubernetes-with-an-atomic-host/` and `https://github.com/cgwalters/vagrant-atomic-cluster`

Doing atomic update/rollback with Project Atomic

To get to the latest version or to roll back to the older version of Project Atomic, we use the `atomic host` command, which internally calls rpm-ostree.

Getting ready

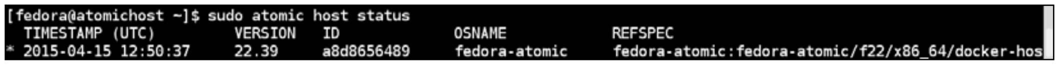
Boot and log in to the Atomic host.

How to do it...

1. Just after the boot, run the following command:

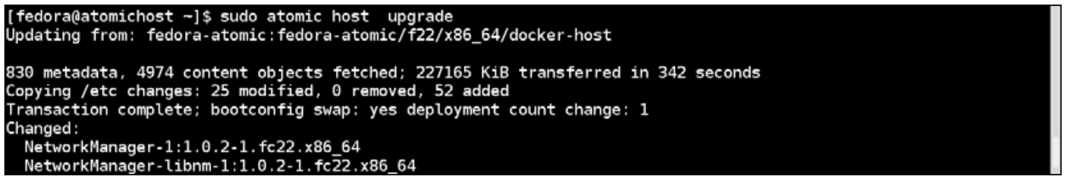
   `$ atomic host status`

   You will see details about one deployment that is in use now.



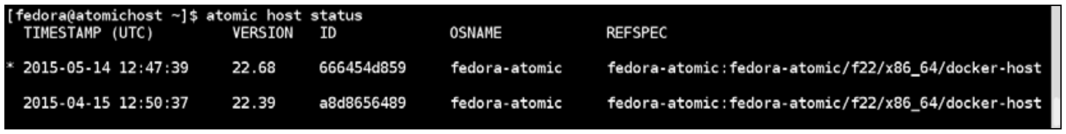
```
[fedora@atomichost ~]$ sudo atomic host status
  TIMESTAMP (UTC)         VERSION   ID             OSNAME            REFSPEC
* 2015-04-15 12:50:37     22.39     a8d8656489     fedora-atomic     fedora-atomic:fedora-atomic/f22/x86_64/docker-hos
```

   To upgrade, run the following command:



```
[fedora@atomichost ~]$ sudo atomic host  upgrade
Updating from: fedora-atomic:fedora-atomic/f22/x86_64/docker-host

830 metadata, 4974 content objects fetched; 227165 KiB transferred in 342 seconds
Copying /etc changes: 25 modified, 0 removed, 52 added
Transaction complete; bootconfig swap: yes deployment count change: 1
Changed:
  NetworkManager-1:1.0.2-1.fc22.x86_64
  NetworkManager-libnm-1:1.0.2-1.fc22.x86_64
```

2. This changes and/or adds new packages. After the upgrade, we will need to reboot the system to use the new update. Let's reboot and see the outcome:



```
[fedora@atomichost ~]$ atomic host status
  TIMESTAMP (UTC)         VERSION   ID             OSNAME            REFSPEC

* 2015-05-14 12:47:39     22.68     666454d859     fedora-atomic     fedora-atomic:fedora-atomic/f22/x86_64/docker-host

  2015-04-15 12:50:37     22.39     a8d8656489     fedora-atomic     fedora-atomic:fedora-atomic/f22/x86_64/docker-host
```

   As we can see, the system is now booted with the new update. The `*`, which is at the beginning of the first line, specifies the active build.

3. To roll back, run the following command:

   ```
   $ sudo atomic host rollback
   ```

   We will have to reboot again if we want to use older bits.

How it works...

For updates, the Atomic host connects to the remote repository hosting the newer build, which is downloaded and used from the next reboot onwards until the user upgrades or rolls back. In the case rollback older build available on the system used after the reboot.

See also

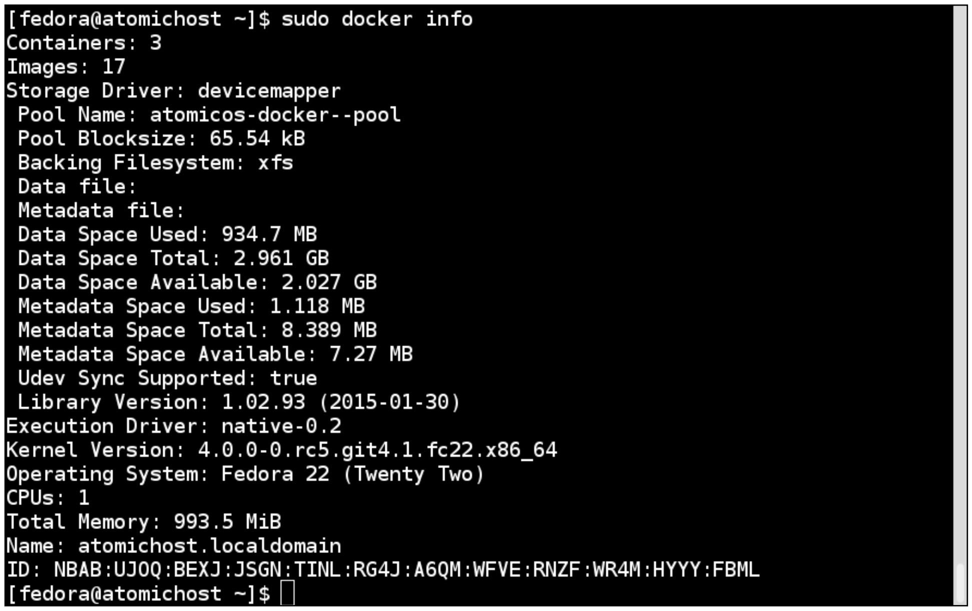
- The documentation Project Atomic website, which can be found at `http://www.projectatomic.io/docs/os-updates/`

Adding more storage for Docker in Project Atomic

The Atomic host is a minimal distribution and, as such, is distributed on a 6 GB image to keep the footprint small. This is very less amount of storage to build and store lots of Docker images, so it is recommended to attach external storage for those operations.

By default, Docker uses `/var/lib/docker` as the default directory where all Docker-related files, including images, are stored. In Project Atomic, we use direct LVM volumes via the devicemapper backend to store Docker images and metadata in `/dev/atomicos/docker-data` and `/dev/atomicos/docker-meta` respectively.

So, to add more storage, Project Atomic provides a helper script called `docker-storage-helper` to add an external disk into the existing LVM thin pool. Let's look at the current available storage to Docker with the `docker info` command:



```
[fedora@atomichost ~]$ sudo docker info
Containers: 3
Images: 17
Storage Driver: devicemapper
 Pool Name: atomicos-docker--pool
 Pool Blocksize: 65.54 kB
 Backing Filesystem: xfs
 Data file:
 Metadata file:
 Data Space Used: 934.7 MB
 Data Space Total: 2.961 GB
 Data Space Available: 2.027 GB
 Metadata Space Used: 1.118 MB
 Metadata Space Total: 8.389 MB
 Metadata Space Available: 7.27 MB
 Udev Sync Supported: true
 Library Version: 1.02.93 (2015-01-30)
Execution Driver: native-0.2
Kernel Version: 4.0.0-0.rc5.git4.1.fc22.x86_64
Operating System: Fedora 22 (Twenty Two)
CPUs: 1
Total Memory: 993.5 MiB
Name: atomichost.localdomain
ID: NBAB:UJOQ:BEXJ:JSGN:TINL:RG4J:A6QM:WFVE:RNZF:WR4M:HYYY:FBML
[fedora@atomichost ~]$
```

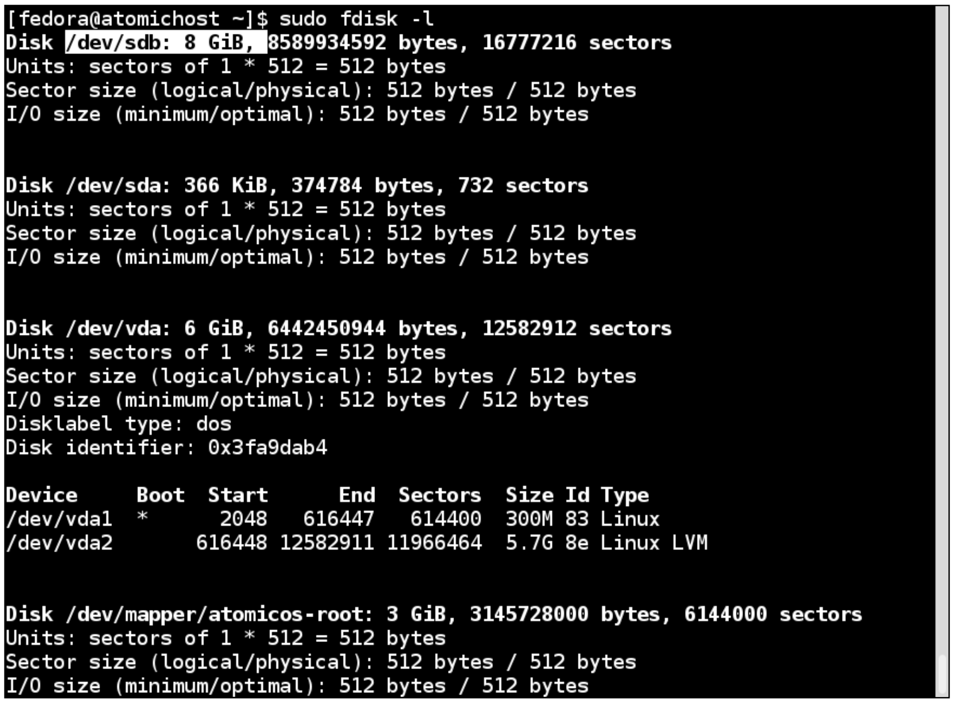
As we can see, the total data space is 2.96 GB and the total metadata space is 8.38 MB.

Getting ready

1. Stop the VM, if it is running.
2. Add an additional disk of the size you want to the Project Atomic VM. I have added 8 GB.
3. Boot the VM.
4. Check whether the newly attached disk is visible to the VM or not.

How to do it...

1. Check if the additional disk is available to the Atomic host VM:



```
[fedora@atomichost ~]$ sudo fdisk -l
Disk /dev/sdb: 8 GiB, 8589934592 bytes, 16777216 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes


Disk /dev/sda: 366 KiB, 374784 bytes, 732 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes


Disk /dev/vda: 6 GiB, 6442450944 bytes, 12582912 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x3fa9dab4

Device     Boot  Start      End  Sectors  Size Id Type
/dev/vda1  *      2048   616447   614400  300M 83 Linux
/dev/vda2       616448 12582911 11966464  5.7G 8e Linux LVM


Disk /dev/mapper/atomicos-root: 3 GiB, 3145728000 bytes, 6144000 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

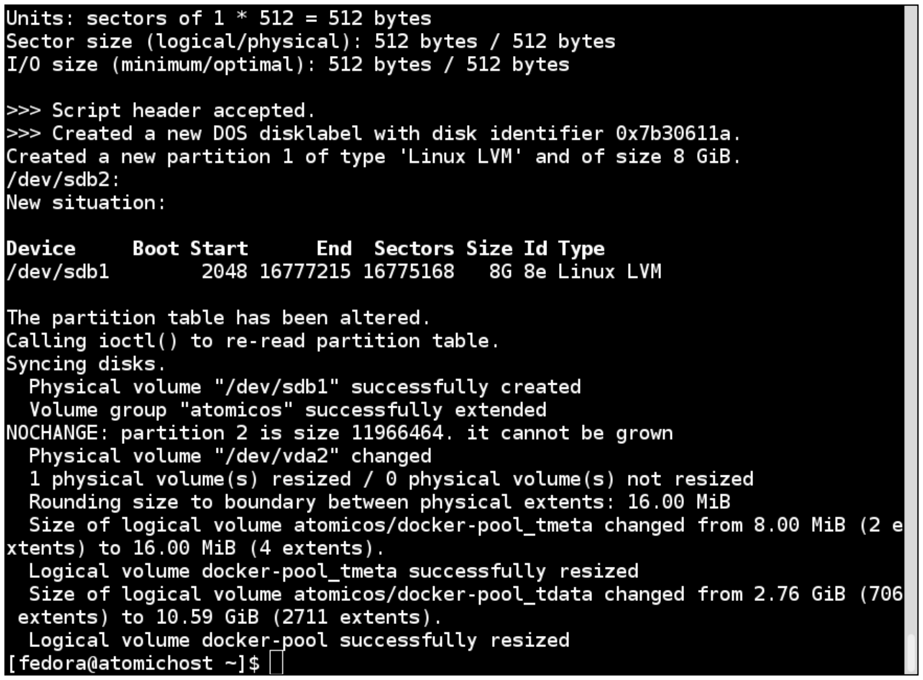
As we can see, the newly created 8 GB disk is available to the VM.

2. As the newly attached disk is `/dev/sdb`, create a file called `/etc/sysconfig/docker-storage-setup` with the following content:

```
DEVS="/dev/sdb"
[fedora@atomichost ~]$ cat /etc/sysconfig/docker-storage-setup
DEVS="/dev/sdb"
```

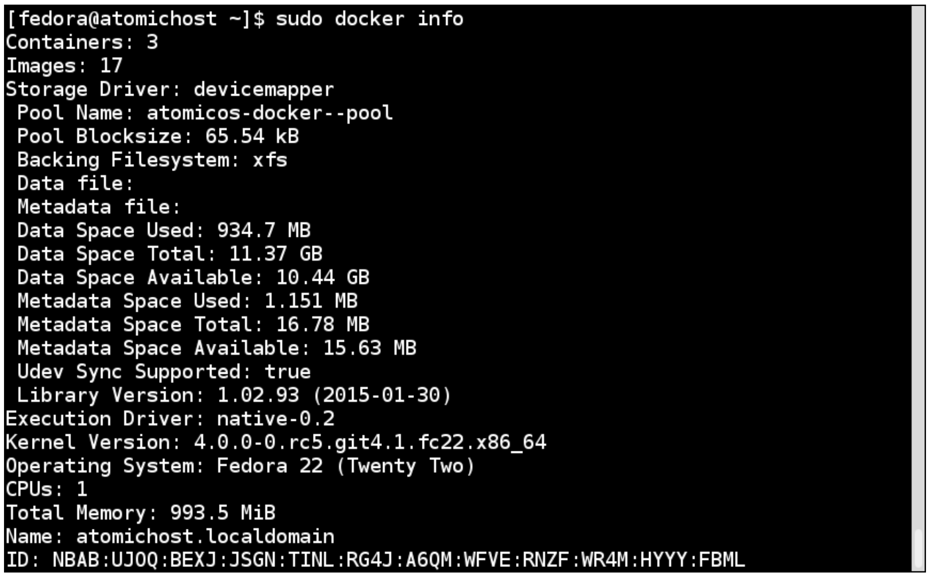
3. Run the `docker-storage-setup` command to add `/dev/sdb` to the existing volume:

```
$ sudo docker-storage-setup
```



```
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes

>>> Script header accepted.
>>> Created a new DOS disklabel with disk identifier 0x7b30611a.
Created a new partition 1 of type 'Linux LVM' and of size 8 GiB.
/dev/sdb2:
New situation:

Device     Boot Start      End  Sectors Size Id Type
/dev/sdb1        2048 16777215 16775168   8G 8e Linux LVM

The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.
  Physical volume "/dev/sdb1" successfully created
  Volume group "atomicos" successfully extended
NOCHANGE: partition 2 is size 11966464. it cannot be grown
  Physical volume "/dev/vda2" changed
  1 physical volume(s) resized / 0 physical volume(s) not resized
  Rounding size to boundary between physical extents: 16.00 MiB
  Size of logical volume atomicos/docker-pool_tmeta changed from 8.00 MiB (2 e
xtents) to 16.00 MiB (4 extents).
  Logical volume docker-pool_tmeta successfully resized
  Size of logical volume atomicos/docker-pool_tdata changed from 2.76 GiB (706
 extents) to 10.59 GiB (2711 extents).
  Logical volume docker-pool successfully resized
[fedora@atomichost ~]$ 
```

4. Now, let's look at the current available storage to Docker once again with the `docker info` command:



```
[fedora@atomichost ~]$ sudo docker info
Containers: 3
Images: 17
Storage Driver: devicemapper
 Pool Name: atomicos-docker--pool
 Pool Blocksize: 65.54 kB
 Backing Filesystem: xfs
 Data file:
 Metadata file:
 Data Space Used: 934.7 MB
 Data Space Total: 11.37 GB
 Data Space Available: 10.44 GB
 Metadata Space Used: 1.151 MB
 Metadata Space Total: 16.78 MB
 Metadata Space Available: 15.63 MB
 Udev Sync Supported: true
 Library Version: 1.02.93 (2015-01-30)
Execution Driver: native-0.2
Kernel Version: 4.0.0-0.rc5.git4.1.fc22.x86_64
Operating System: Fedora 22 (Twenty Two)
CPUs: 1
Total Memory: 993.5 MiB
Name: atomichost.localdomain
ID: NBAB:UJOQ:BEXJ:JSGN:TINL:RG4J:A6QM:WFVE:RNZF:WR4M:HYYY:FBML
```

As we can see, both the total data space and metadata space have increased.

How it works...

The procedure is the same as extending any other LVM volume. We create a physical volume on the added disk, add that physical volume to the volume group, and then extend the LVM volumes. Since we are directly accessing the thin pool within Docker, we won't need to create or extend a filesystem or mount the LVM volumes.

There's more...

- In addition to the `DEVS` option, you can also add the `VG` option to the `/etc/sysconfig/docker-storage-setup` file to use a different volume group.
- You can add more than one disk with the `DEVS` option.
- If a disk that is already part of the Volume Group has been mentioned with the `DEVS` option, then the `docker-storage-setup` script will exit, as the existing device has a partition and physical volume already created.
- The `docker-storage-setup` script reserves 0.1 percent of the size for `meta-data`. This is why we saw an increase in the Metadata Space as well.

See also

- The documentation on the Project Atomic website at `http://www.projectatomic.io/docs/docker-storage-recommendation/`
- Supported filesystems with Project Atomic at `http://www.projectatomic.io/docs/filesystems/`

Setting up Cockpit for Project Atomic

Cockpit (`http://cockpit-project.org/`) is a server manager that makes it easy to administer your GNU/Linux servers via a web browser. It can be used to manage the Project Atomic host as well. More than one host can be managed through one Cockpit instance. Cockpit does not come by default with the latest Project Atomic, and you will need to start it as a **Super Privileged Container** (**SPC**). SPCs are specially built containers that run with security turned off (`--privileged`); they turn off one or more of the namespaces or "volume mounts in" parts of the host OS into the container. For more details on SPC, refer to `https://developerblog.redhat.com/2014/11/06/introducing-a-super-privileged-container-concept/` and `https://www.youtube.com/watch?v=eJIeGnHtIYg`.

Because Cockpit runs as an SPC, it can access the resources needed to manage the Atomic host within the container.

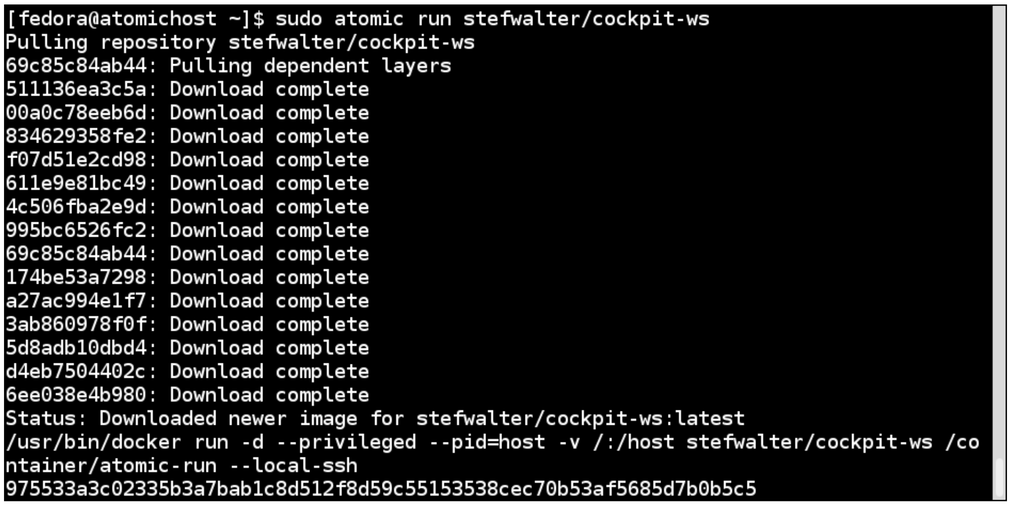
Getting ready

Set up the Project Atomic host and log in to it.

How to do it...

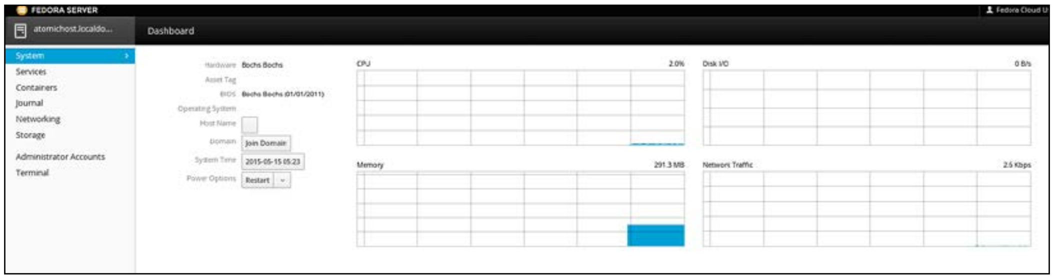
1. Run the following command to start the Cockpit container:

   ```
   [fedora@atomichost ~]$ sudo atomic run stefwalter/cockpit-ws
   ```



```
[fedora@atomichost ~]$ sudo atomic run stefwalter/cockpit-ws
Pulling repository stefwalter/cockpit-ws
69c85c84ab44: Pulling dependent layers
511136ea3c5a: Download complete
00a0c78eeb6d: Download complete
834629358fe2: Download complete
f07d51e2cd98: Download complete
611e9e81bc49: Download complete
4c506fba2e9d: Download complete
995bc6526fc2: Download complete
69c85c84ab44: Download complete
174be53a7298: Download complete
a27ac994e1f7: Download complete
3ab860978f0f: Download complete
5d8adb10dbd4: Download complete
d4eb7504402c: Download complete
6ee038e4b980: Download complete
Status: Downloaded newer image for stefwalter/cockpit-ws:latest
/usr/bin/docker run -d --privileged --pid=host -v /:/host stefwalter/cockpit-ws /co
ntainer/atomic-run --local-ssh
975533a3c02335b3a7bab1c8d512f8d59c55153538cec70b53af5685d7b0b5c5
```

2. Open the browser (`http://<VM IP>:9090`) and log in with the default user/password `fedora/atomic`. Once logged in, you can select the current host to manage. You will see a screen as shown here:



Hardware Bochs Bochs
Asset Tag
BIOS Bochs Bochs (01/01/2011)
Operating System
Host Name
Domain Join Domain
System Time 2015-05-15 05:23
Power Options Restart

How it works...

Here, we used the `atomic` command instead of the `docker` command to start the container. Let's look at the Cockpit Dockerfile (`https://github.com/fedora-cloud/Fedora-Dockerfiles/blob/master/cockpit-ws/Dockerfile`) to see why we did that. In the Dockerfile you will see some instructions:

```
LABEL INSTALL /usr/bin/docker run -ti --rm --privileged -v /:/host
IMAGE /container/atomic-install
LABEL UNINSTALL /usr/bin/docker run -ti --rm --privileged -v
/:/host IMAGE /cockpit/atomic-uninstall
LABEL RUN /usr/bin/docker run -d --privileged --pid=host -v
/:/host IMAGE /container/atomic-run --local-ssh
```

If you recall from *Chapter 2*, *Working with Docker Containers* and *Chapter 3*, *Working with Docker Images*, we could assign metadata to images and containers using labels. `INSTALL`, `UNINSTALL`, and `RUN` are labels here. The `atomic` command is a command specific to Project Atomic, which reads those labels and performs operations. As the container is running as an SPC, it does not need port forwarding from host to container. For more details on the `atomic` command, please visit `https://developerblog.redhat.com/2015/04/21/introducing-the-atomic-command/`.

There's more...

You can perform almost all administrator tasks from the GUI for the given system. You can manage Docker images/containers through this. You can perform operations such as:

- Pulling an image
- Starting/stopping the containers

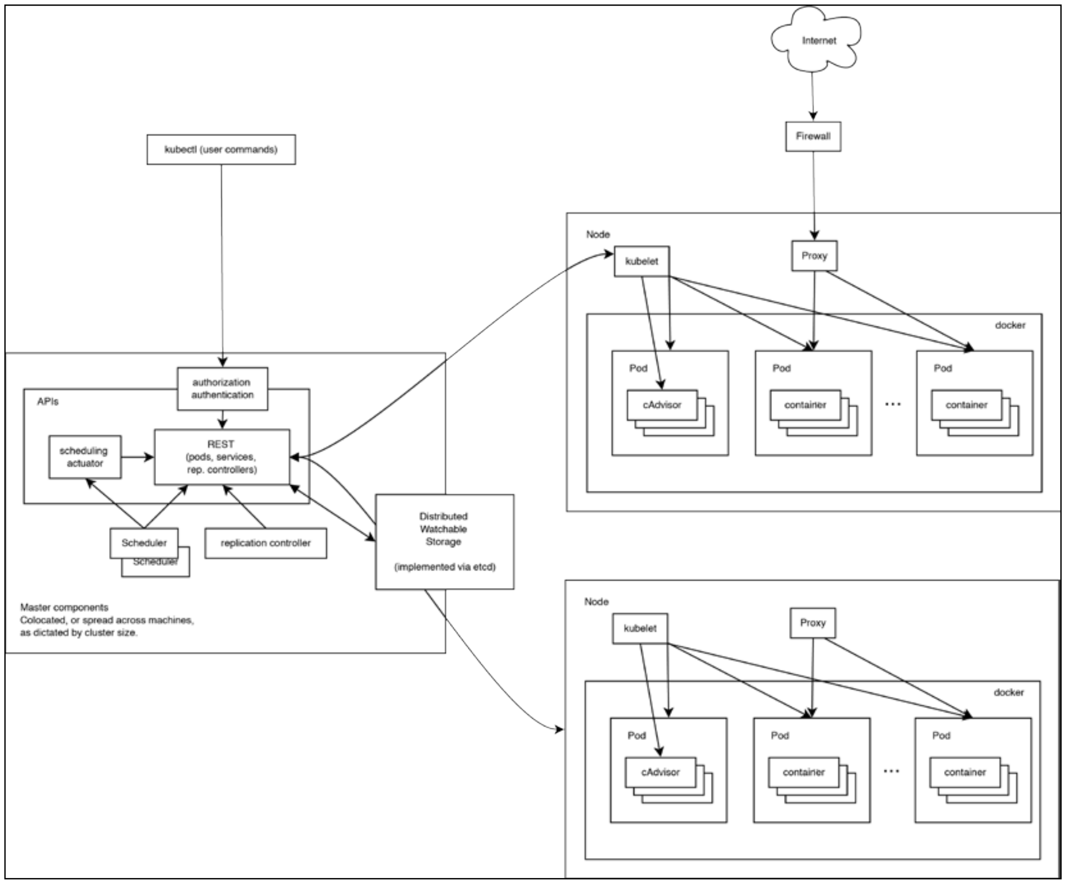
You can also add other machines to the same Cockpit instance so that you manage them from one central location.

See also

- The Cockpit documentation at `http://files.cockpit-project.org/guide/`

Setting up a Kubernetes cluster

Kubernetes is an open source container orchestration tool across multiple nodes in the cluster. Currently, it only supports Docker. 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, 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 the recipe. These YAML and JSON files also contain the API Version (the `apiVersion` field) to identify the schema. The following is the architectural diagram of Kubernetes:



`https://raw.githubusercontent.com/GoogleCloudPlatform/kubernetes/master/docs/architecture.png`

Let's look at some of the key components and concepts of 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 can all communicate through localhost.
- **Node/Minion**: A node, which was previously known as a minion, is a worker node in the Kubernetes cluster and is managed through master. Pods are deployed on a node, which has the necessary services to run them:
    - docker, to run containers
    - kubelet, to interact with master
    - proxy (kube-proxy), which connects the service to the corresponding pod
- **Master**: Master hosts cluster-level control services such as the following:
    - **API server**: This has RESTful APIs to interact with 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.
    - **Replication controller**: This ensures that the user-specified number of pod replicas is running at any given time. To manage replicas with replication controller, we have to define a configuration file with the replica count for a pod.

  Master also communicates with etcd, which is a distributed key-value pair. etcd is used to store the configuration information, which is used by both master and nodes. The watch functionality of etcd is used to notify the changes in the cluster. etcd can be hosted on 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 a pod's IP address to cater an app. To overcome this problem, Kubernetes defines an abstraction, which defines a logical set of pods and policies to access them. This abstraction is called a service. Labels are used to define the logical set, which a service manages.
- **Labels**: Labels are key-value pairs that can be attached to objects like, using which we 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), HostDir, GCEPersistentDisk, and NFS. Active development is happening to support more types of volumes. More details can be found at `https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/volumes.md`.

Kubernetes can be installed on VMs, physical machines, and the cloud. For the complete matrix, take a look at `https://github.com/GoogleCloudPlatform/kubernetes/tree/master/docs/getting-started-guides`. In this recipe, we'll see how to install it on VMs, using Vagrant with VirtualBox provider. This recipe and the following recipes on Kubernetes, were tried on v0.17.0 of Kubernetes.

Getting ready

1. Install latest Vagrant >= 1.6.2 from `http://www.vagrantup.com/downloads.html`.
2. Install the latest VirtualBox from `https://www.virtualbox.org/wiki/Downloads`. Detailed instructions on how to set this up are outside the scope of this book.

How to do it...

1. Run the following command to set up Kubernetes on Vagrant VMs:

```
$ export KUBERNETES_PROVIDER=vagrant
$ export VAGRANT_DEFAULT_PROVIDER=virtualbox
$ curl -sS https://get.k8s.io | bash
```

How it works...

The bash script downloaded from the `curl` command, first downloads the latest Kubernetes release and then runs the `./kubernetes/cluster/kube-up.sh` bash script to set up the Kubernetes environment. As we have specified Vagrant as `KUBERNETES_PROVIDER`, the script first downloads the Vagrant images and then, using Salt (`http://saltstack.com/`), configures one master and one node (minion) VM. Initial setup takes a few minutes to run.

Vagrant creates a credential file in `~/.kubernetes_vagrant_auth` for authentication.

There's more...

Similar to `./cluster/kube-up.sh`, there are other helper scripts to perform different operations from the host machine itself. Make sure you are in the `kubernetes` directory, which was created with the preceding installation, while running the following commands:

- Get the list of nodes:

```
$ ./cluster/kubectl.sh get nodes
```

- Get the list of pods:

```
$ ./cluster/kubectl.sh get pods
```

- Get the list of services:

  ```
  $ ./cluster/kubectl.sh get services
  ```

- Get the list of replication controllers:

  ```
  $ ./cluster/kubectl.sh get replicationControllers
  ```

- Destroy the vagrant cluster:

  ```
  $ ./cluster/kube-down.sh
  ```

- Then bring back the vagrant cluster:

  ```
  $ ./cluster/kube-up.sh
  ```

You will see some `pods`, `services`, and `replicationControllers` listed, as Kubernetes creates them for internal use.

See also

- Setting up the Vagrant environment at `https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/getting-started-guides/vagrant.md`
- The Kubernetes user guide at `https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/user-guide.md`
- Kubernetes API conventions at `https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/api-conventions.md`

Scaling up and down in a Kubernetes cluster

In the previous section, we mentioned that the replication controller ensures that the user-specified number of pod replicas is running at any given time. To manage replicas with the replication controller, we have to define a configuration file with the replica count for a pod. This configuration can be changed at runtime.

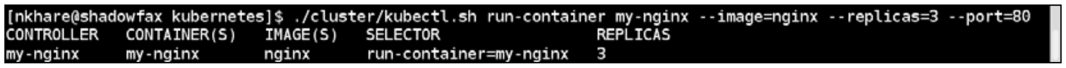
Getting ready

Make sure 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...

1. Start the `nginx` container with a replica count of 3:

   ```
   $ ./cluster/kubectl.sh run-container my-nginx --image=nginx
   --replicas=3 --port=80
   ```



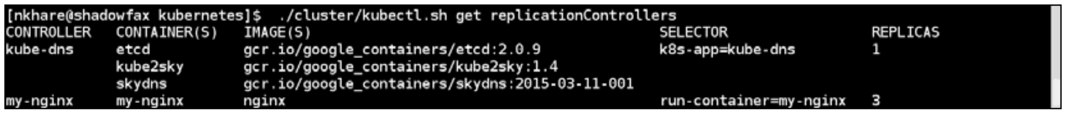
```
[nkhare@shadowfax kubernetes]$ ./cluster/kubectl.sh run-container my-nginx --image=nginx --replicas=3 --port=80
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR                 REPLICAS
my-nginx     my-nginx       nginx      run-container=my-nginx   3
```

   This will start three replicas of the `nginx` container. List the pods to get the status:

   ```
   $  ./cluster/kubectl.sh get pods
   ```

2. Get the replication controller configuration:

   ```
   $ ./cluster/kubectl.sh get replicationControllers
   ```

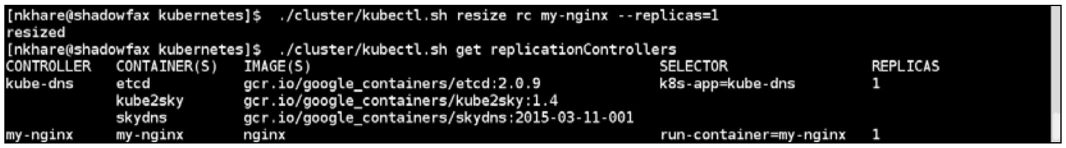


```
[nkhare@shadowfax kubernetes]$  ./cluster/kubectl.sh get replicationControllers
CONTROLLER   CONTAINER(S)   IMAGE(S)                                         SELECTOR                 REPLICAS
kube-dns     etcd           gcr.io/google_containers/etcd:2.0.9              k8s-app=kube-dns         1
             kube2sky       gcr.io/google_containers/kube2sky:1.4
             skydns         gcr.io/google_containers/skydns:2015-03-11-001
my-nginx     my-nginx       nginx                                            run-container=my-nginx   3
```

   As you can see, we have a `my-nginx` controller, which has a replica count of 3. There is a replication controller for `kube-dns`, which we will explore in next recipe.

3. Request the replication controller service to scale down to replica of 1 and update the replication controller:

   ```
   $ ./cluster/kubectl.sh resize rc my-nginx –replicas=1
   $ ./cluster/kubectl.sh get rc
   ```



```
[nkhare@shadowfax kubernetes]$  ./cluster/kubectl.sh resize rc my-nginx --replicas=1
resized
[nkhare@shadowfax kubernetes]$  ./cluster/kubectl.sh get replicationControllers
CONTROLLER   CONTAINER(S)   IMAGE(S)                                         SELECTOR                 REPLICAS
kube-dns     etcd           gcr.io/google_containers/etcd:2.0.9              k8s-app=kube-dns         1
             kube2sky       gcr.io/google_containers/kube2sky:1.4
             skydns         gcr.io/google_containers/skydns:2015-03-11-001
my-nginx     my-nginx       nginx                                            run-container=my-nginx   1
```

4. Get the list of pods to verify; you should see only one pod for `nginx`:

   ```
   $  ./cluster/kubectl.sh get pods
   ```

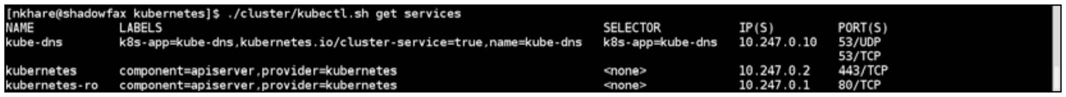
How it works...

We request the replication controller service running on master to update the replicas for a pod, which updates the configuration and requests nodes/minions to act accordingly to honor the resizing.

There's more...

Get the services:

```
$ ./cluster/kubectl.sh get services
```



```
[nkhare@shadowfax kubernetes]$ ./cluster/kubectl.sh get services
NAME            LABELS                                                           SELECTOR           IP(S)          PORT(S)
kube-dns        k8s-app=kube-dns,kubernetes.io/cluster-service=true,name=kube-dns   k8s-app=kube-dns   10.247.0.10    53/UDP
                                                                                                                   53/TCP
kubernetes      component=apiserver,provider=kubernetes                             <none>             10.247.0.2     443/TCP
kubernetes-ro   component=apiserver,provider=kubernetes                             <none>             10.247.0.1     80/TCP
```

As you can see, we don't have any service defined for our `nginx` containers started earlier. This means that though we have a container running, we cannot access them from outside because the corresponding service is not defined.

See also

- Setting up the Vagrant environment at `https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/getting-started-guides/vagrant.md`
- The Kubernetes user guide at `https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/user-guide.md`

Setting up WordPress with a Kubernetes cluster

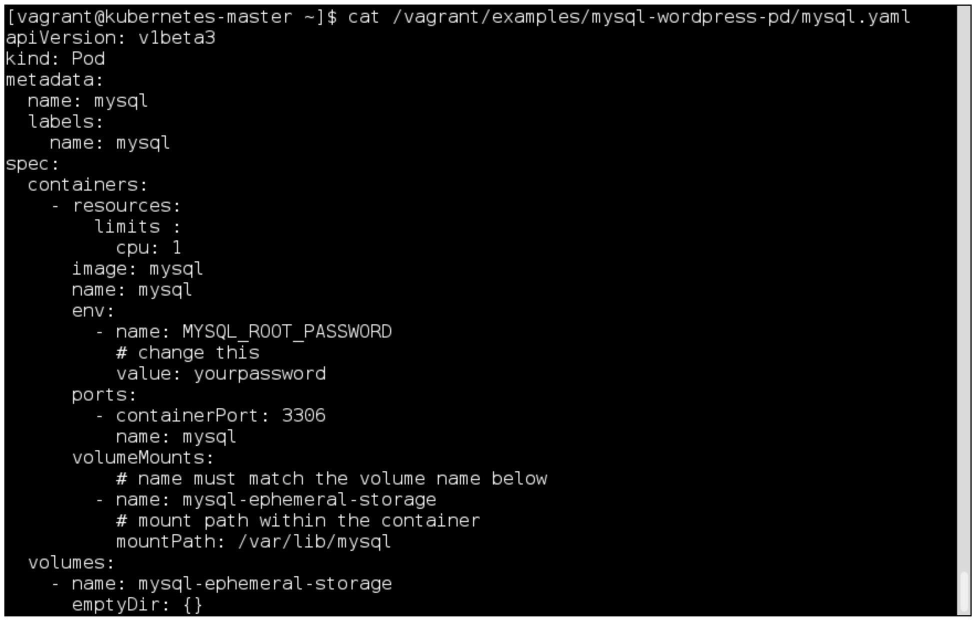
In this recipe, we will use the WordPress example given in the Kubernetes GitHub (`https://github.com/GoogleCloudPlatform/kubernetes/tree/master/examples/mysql-wordpress-pd`). The given example requires some changes, as we'll be running it on the Vagrant environment instead of the default Google Compute engine. Also, instead of using the helper functions (for example, `<kubernetes>/cluster/kubectl.sh`), we'll log in to master and use the `kubectl` binary.

Getting ready

- Make sure the Kubernetes cluster has been set up as described in the previous recipe.
- In the `kubernetes` directory that was downloaded during the setup, you will find an examples directory that contains many examples. Let's go to the `mysql-wordpress-pd` directory:

  ```
  $ cd kubernetes/examples/mysql-wordpress-pd
  $  ls *.yaml
  mysql-service.yaml mysql.yaml wordpress-service.yaml  wordpress.
  yaml
  ```

- These `.yaml` files describe pods and services for `mysql` and `wordpress` respectively.
- In the pods files (`mysql.yaml` and `wordpress.yaml`), you will find the section on volumes and the corresponding `volumeMount` file. The original example assumes that you have access to Google Compute Engine and that you have the corresponding storage setup. For simplicity, we will not set up that and instead use ephemeral storage with the `EmptyDir` volume option. For reference, our `mysql.yaml` will look like the following:



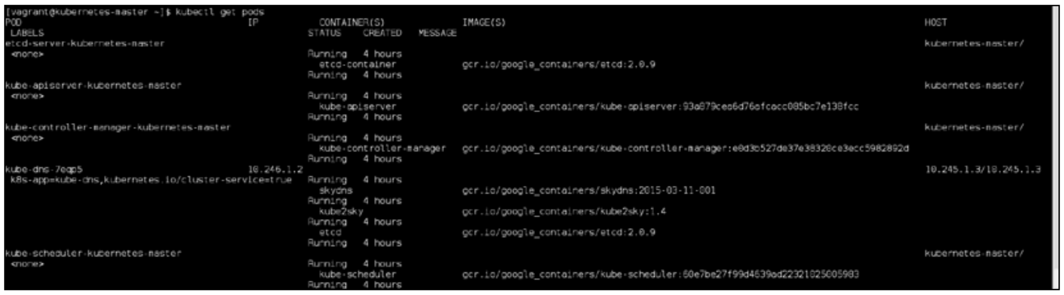
```
[vagrant@kubernetes-master ~]$ cat /vagrant/examples/mysql-wordpress-pd/mysql.yaml
apiVersion: v1beta3
kind: Pod
metadata:
  name: mysql
  labels:
    name: mysql
spec:
  containers:
    - resources:
        limits :
          cpu: 1
      image: mysql
      name: mysql
      env:
        - name: MYSQL_ROOT_PASSWORD
          # change this
          value: yourpassword
      ports:
        - containerPort: 3306
          name: mysql
      volumeMounts:
          # name must match the volume name below
        - name: mysql-ephemeral-storage
          # mount path within the container
          mountPath: /var/lib/mysql
  volumes:
    - name: mysql-ephemeral-storage
      emptyDir: {}
```

- Make the similar change to `wordpress.yaml`.

How to do it...

1. With SSH, log in to the master node and look at the running pods:

   ```
   $ vagrant ssh master
   $ kubectl get pods
   ```



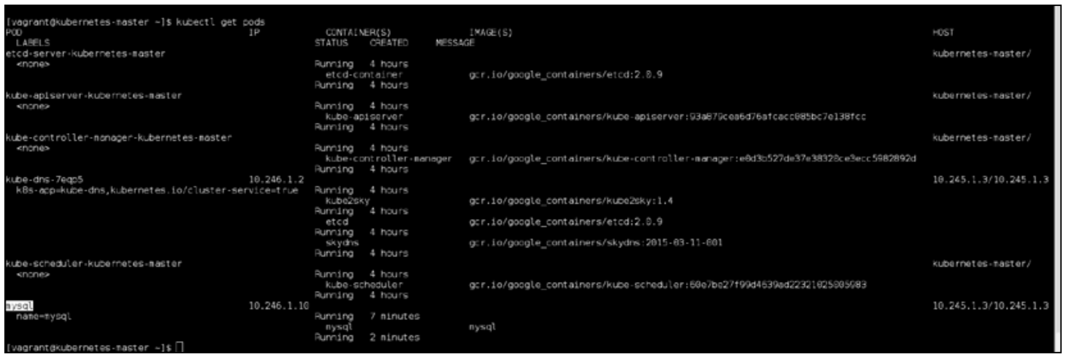
```
[vagrant@kubernetes-master ~]$ kubectl get pods
POD                                            IP             CONTAINER(S)              IMAGE(S)                                                                              HOST
 LABELS                                                       STATUS     CREATED   MESSAGE
etcd-server-kubernetes-master                                                                                                                                                   kubernetes-master/
 <none>                                                       Running    4 hours
                                                                etcd-container          gcr.io/google_containers/etcd:2.0.9
                                                              Running    4 hours
kube-apiserver-kubernetes-master                                                                                                                                                kubernetes-master/
 <none>                                                       Running    4 hours
                                                                kube-apiserver          gcr.io/google_containers/kube-apiserver:93a879cea6d76afcacc085bc7e138fcc
                                                              Running    4 hours
kube-controller-manager-kubernetes-master                                                                                                                                       kubernetes-master/
 <none>                                                       Running    4 hours
                                                                kube-controller-manager gcr.io/google_containers/kube-controller-manager:e0d3b527de37e38328ce3ecc5982892d
                                                              Running    4 hours
kube-dns-7eqp5                                 10.246.1.2                                                                                                                       10.245.1.3/10.245.1.3
 k8s-app=kube-dns,kubernetes.io/cluster-service=true          Running    4 hours
                                                                skydns                  gcr.io/google_containers/skydns:2015-03-11-001
                                                              Running    4 hours
                                                                kube2sky                gcr.io/google_containers/kube2sky:1.4
                                                              Running    4 hours
                                                                etcd                    gcr.io/google_containers/etcd:2.0.9
                                                              Running    4 hours
kube-scheduler-kubernetes-master                                                                                                                                                kubernetes-master/
 <none>                                                       Running    4 hours
                                                                kube-scheduler          gcr.io/google_containers/kube-scheduler:60e7be27f99d4639ad22321025005983
                                                              Running    4 hours
```

The `kube-dns-7eqp5` pod consists of three containers: `etcd`, `kube2sky`, and `skydns`, which are used to configure an internal DNS server for service name to IP resolution. We'll see it in action later in this recipe.

The Vagrantfile used in this example is created so that the `kubernetes` directory that we created earlier is shared under `/vagrant` on VM, which means that the changes we made to the host system will be visible here as well.

2. From the master node, create the `mysql` pod and check the running pods:

   ```
   $ kubectl create -f /vagrant/examples/mysql-wordpress-
   pd/mysql.yaml
   $ kubectl get pods
   ```



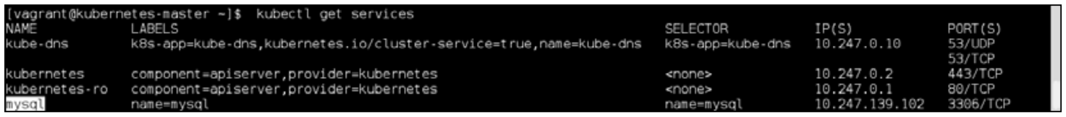
```
[vagrant@kubernetes-master ~]$ kubectl get pods
POD                                            IP             CONTAINER(S)              IMAGE(S)                                                                              HOST
 LABELS                                                       STATUS     CREATED   MESSAGE
etcd-server-kubernetes-master                                                                                                                                                   kubernetes-master/
 <none>                                                       Running    4 hours
                                                                etcd-container          gcr.io/google_containers/etcd:2.0.9
                                                              Running    4 hours
kube-apiserver-kubernetes-master                                                                                                                                                kubernetes-master/
 <none>                                                       Running    4 hours
                                                                kube-apiserver          gcr.io/google_containers/kube-apiserver:93a879cea6d76afcacc085bc7e138fcc
                                                              Running    4 hours
kube-controller-manager-kubernetes-master                                                                                                                                       kubernetes-master/
 <none>                                                       Running    4 hours
                                                                kube-controller-manager gcr.io/google_containers/kube-controller-manager:e0d3b527de37e38328ce3ecc5982892d
                                                              Running    4 hours
kube-dns-7eqp5                                 10.246.1.2                                                                                                                       10.245.1.3/10.245.1.3
 k8s-app=kube-dns,kubernetes.io/cluster-service=true          Running    4 hours
                                                                kube2sky                gcr.io/google_containers/kube2sky:1.4
                                                              Running    4 hours
                                                                etcd                    gcr.io/google_containers/etcd:2.0.9
                                                              Running    4 hours
                                                                skydns                  gcr.io/google_containers/skydns:2015-03-11-001
                                                              Running    4 hours
kube-scheduler-kubernetes-master                                                                                                                                                kubernetes-master/
 <none>                                                       Running    4 hours
                                                                kube-scheduler          gcr.io/google_containers/kube-scheduler:60e7be27f99d4639ad22321025005983
                                                              Running    4 hours
mysql                                          10.246.1.10                                                                                                                      10.245.1.3/10.245.1.3
 name=mysql                                                   Running    7 minutes
                                                                mysql                   mysql
                                                              Running    2 minutes
[vagrant@kubernetes-master ~]$
```

As we can see, a new pod with the `mysql` name has been created and it is running on host `10.245.1.3`, which is our node (minion).

3. Now let's create the service for `mysql` and look at all the services:

   `$ kubectl create -f /vagrant/examples/mysql-wordpress-pd/mysql-service.yaml`

   `$ kubectl get services`



```
[vagrant@kubernetes-master ~]$  kubectl get services
NAME            LABELS                                                           SELECTOR             IP(S)            PORT(S)
kube-dns        k8s-app=kube-dns,kubernetes.io/cluster-service=true,name=kube-dns   k8s-app=kube-dns     10.247.0.10      53/UDP
                                                                                                                       53/TCP
kubernetes      component=apiserver,provider=kubernetes                          <none>               10.247.0.2       443/TCP
kubernetes-ro   component=apiserver,provider=kubernetes                          <none>               10.247.0.1       80/TCP
mysql           name=mysql                                                       name=mysql           10.247.139.102   3306/TCP
```

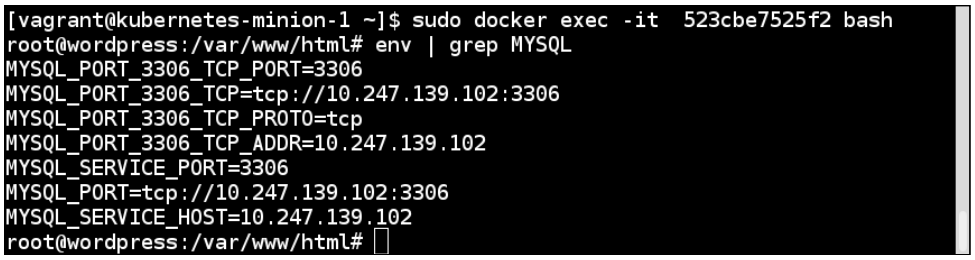
   As we can see, a service named `mysql` has been created. Each service has a Virtual IP. Other than the `kubernetes` services, we see a service named `kube-dns`, which is used as the service name for the `kube-dns` pod we saw earlier.

4. Similar to `mysql`, let's create a pod for `wordpress`:

   `$ kubectl create -f /vagrant/examples/mysql-wordpress-pd/wordpress.yaml`

   With this command, there are a few things happening in the background:

   - The `wordpress` image gets downloaded from the official Docker registry and the container runs.
   - By default, whenever a pod starts, information about all the existing services is exported as environment variables. For example, if we log in to the `wordpress` pod and look for `MYSQL`-specific environment variables, we will see something like the following:



```
[vagrant@kubernetes-minion-1 ~]$ sudo docker exec -it  523cbe7525f2 bash
root@wordpress:/var/www/html# env | grep MYSQL
MYSQL_PORT_3306_TCP_PORT=3306
MYSQL_PORT_3306_TCP=tcp://10.247.139.102:3306
MYSQL_PORT_3306_TCP_PROTO=tcp
MYSQL_PORT_3306_TCP_ADDR=10.247.139.102
MYSQL_SERVICE_PORT=3306
MYSQL_PORT=tcp://10.247.139.102:3306
MYSQL_SERVICE_HOST=10.247.139.102
root@wordpress:/var/www/html# 
```

   - When the WordPress container starts, it runs the `/entrypoint.sh` script, which looks for the environment variables mentioned earlier to start the service. `https://github.com/docker-library/wordpress/blob/master/docker-entrypoint.sh`.
   - With the `kube-dns` service, PHP scripts of `wordpress` are able to the reserve lookup to proceed forward.

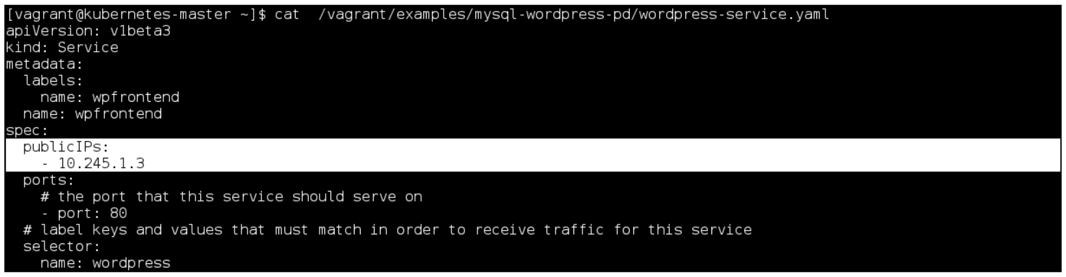
5. After starting the pod, the last step here is to set up the `wordpress` service. In the default example, you will see an entry like the following in the service file (`/vagrant/examples/mysql-wordpress-pd/mysql-service.yaml`):

   ```
   createExternalLoadBalancer: true
   ```

   This has been written to keep in mind that this example will run on the Google Compute Engine. So it is not valid here. In place of that, we will need to make an entry like the following:

   ```
   publicIPs:
       - 10.245.1.3
   ```

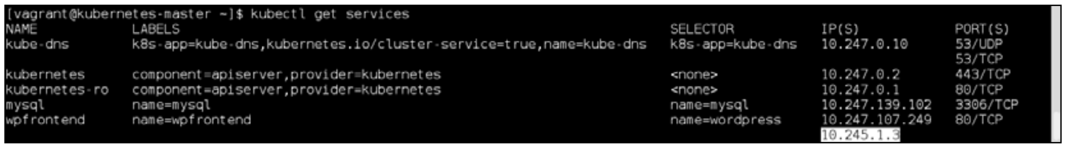
   We have replaced the load-balancer entry with the public IP of the node, which in our case is the IP address of the node (minion). So, the `wordpress` file would look like the following:



```
[vagrant@kubernetes-master ~]$ cat  /vagrant/examples/mysql-wordpress-pd/wordpress-service.yaml
apiVersion: v1beta3
kind: Service
metadata:
  labels:
    name: wpfrontend
  name: wpfrontend
spec:
  publicIPs:
    - 10.245.1.3
  ports:
    # the port that this service should serve on
    - port: 80
  # label keys and values that must match in order to receive traffic for this service
  selector:
    name: wordpress
```

6. To start the `wordpress` service, run the following command from the master node:

   ```
   $ kubectl create -f /vagrant/examples/mysql-wordpress-pd/wordpress-service.yaml
   ```



```
[vagrant@kubernetes-master ~]$ kubectl get services
NAME            LABELS                                                              SELECTOR            IP(S)            PORT(S)
kube-dns        k8s-app=kube-dns,kubernetes.io/cluster-service=true,name=kube-dns   k8s-app=kube-dns    10.247.0.10      53/UDP
                                                                                                                         53/TCP
kubernetes      component=apiserver,provider=kubernetes                             <none>              10.247.0.2       443/TCP
kubernetes-ro   component=apiserver,provider=kubernetes                             <none>              10.247.0.1       80/TCP
mysql           name=mysql                                                          name=mysql          10.247.139.102   3306/TCP
wpfrontend      name=wpfrontend                                                     name=wordpress      10.247.107.249   80/TCP
                                                                                                        10.245.1.3
```

   We can see here that our service is also available through the node (minion) IP.

7. To verify if everything works fine, we can install the links package on master by which we can browse a URL through the command line and connect to the public IP we mentioned:

   ```
   $ sudo yum install links -y
   $ links 10.245.1.3
   ```

   With this, you should see the `wordpress` installation page.

How it works...

In this recipe, we first created a `mysql` pod and service. Later, we connected it to a `wordpress` pod, and to access it, we created a `wordpress` service. Each YAML file has a `kind` key that defines the type of object it is. For example, in pod files, the `kind` is set to pod and in service files, it is set to service.

There's more...

- In this example setup, we have only one Node (minion). If you log in to it, you will see all the running containers:

  ```
  $ vagrant ssh minion-1
  $ sudo docker ps
  ```

- In this example, we have not configured replication controllers. We can extend this example by creating them.

See also

- Setting up the Vagrant environment at `https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/getting-started-guides/vagrant.md`
- The Kubernetes User Guide at `https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/user-guide.md`
- The documentation on kube-dns at `https://github.com/GoogleCloudPlatform/kubernetes/tree/master/cluster/addons/dns`

9
Docker Security

In this chapter, we will cover the following recipes:

- Setting Mandatory Access Control (MAC) with SELinux
- Allowing writes to volume mounted from the host with SELinux ON
- Removing capabilities to breakdown the power of a root user inside the container
- Sharing namespaces between the host and the container

Introduction

Docker containers are not actually 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`, `/proc/bus` are also not namespaced but they are mounted as read only by default with the libcontainer execution driver.

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 blocks, it is very important that we choose the right base image to start with. Docker has the 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 fedora
  ```

  We will see a column `OFFICIAL`, and if the images are official, you will see `[OK]` against that image in that column. There is an experimental feature added in Docker 1.3 (`http://blog.docker.com/2014/10/docker-1-3-signed-images-process-injection-security-options-mac-shared-directories/`), which does Digital Signal Verification of official images after pulling the image. If the image has been tampered with, the user will be notified, but it will not prevent the user from running it. At present, this feature works only with official images. More details about official images can be found at `https://github.com/docker-library/official-images`. The image signing and verification feature is not ready, so as of now, don't completely rely on it.

- In *Chapter 6*, *Docker APIs and Language Bindings*, we saw how we can secure Docker remote API, when Docker daemon access is configured over TCP.
- We can also consider turning off the default intercontainer 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 the processes from creating and using device nodes that could be used to attack the host.
- Any device node precreated on the image cannot be used to talk to kernel because images are mounted with the `nodev` option.

The following are some guidelines (may not be complete), which one can follow to have a secure Docker environment:

- Run services as nonroot 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. Red Hat carries patches to add and block registries to give more control to enterprises (`http://rhelblog.redhat.com/2015/04/15/understanding-the-changes-to-docker-search-and-docker-pull-in-red-hat-enterprise-linux-7-1/`).
- 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 only. Consider using Project Atomic, CoreOS, or similar solutions.
- Mount the devices with the `--device` option rather than using the `--privileged` option to use devices inside the container.
- Prohibit SUID and SGID inside the container.

Recently, Docker and the Center for Internet Security (`http://www.cisecurity.org/`) released a best practices guide for Docker security, which covers most of the preceding guidelines and more guidelines at `https://blog.docker.com/2015/05/understanding-docker-security-and-best-practices/`.

To set the context for some of the recipes in this chapter, let's try an experiment on the default installation on Fedora 21 with Docker installed.

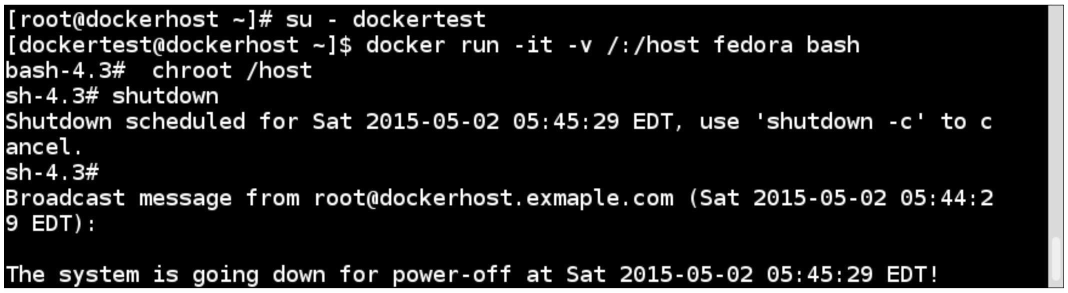
1. Disable SELinux using the following command:
   ```
   $ sudo setenforce 0
   ```
2. Create a user and add it to the default Docker group so that the user can run Docker commands without `sudo`:
   ```
   $ sudo useradd dockertest
   $ sudo passwd dockertest
   $ sudo groupadd docker
   $ sudo gpasswd -a dockertest docker
   ```

3. Log in using the user we created earlier, start a container as follows:

   ```
   $ su - dockertest
   $ docker run -it -v /:/host fedora bash
   ```

4. From the container chroot to `/host` and run the `shutdown` command:

   ```
   $ chroot /host
   $ shutdown
   ```



```
[root@dockerhost ~]# su - dockertest
[dockertest@dockerhost ~]$ docker run -it -v /:/host fedora bash
bash-4.3#  chroot /host
sh-4.3# shutdown
Shutdown scheduled for Sat 2015-05-02 05:45:29 EDT, use 'shutdown -c' to c
ancel.
sh-4.3#
Broadcast message from root@dockerhost.exmaple.com (Sat 2015-05-02 05:44:2
9 EDT):

The system is going down for power-off at Sat 2015-05-02 05:45:29 EDT!
```

As we can see, a user in the 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, only with 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

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:

   ```
   $ getenforce
   ```

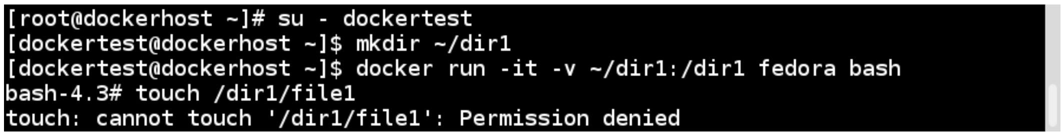
   If the preceding command returns `enforcing`, then it's all good, else we need to change it by updating SELinux configuration file (`/etc/selinux/config`) and rebooting the system.

2. Docker should be running with the `--selinux-enabled` option. You can check the `OPTIONS` section in the Docker daemon configuration (`/etc/sysconfig/docker`) file. Also, cross-check whether the Docker service has started with the SELinux option:

   ```
   $ systemctl status docker
   ```

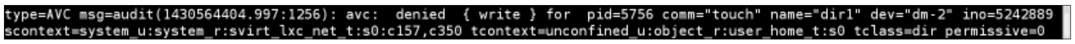
   The preceding command assumes that you are not starting Docker in daemon mode manually.

Let's start a container (without the privileged option) after mounting a host directory as volume and try to create a file in that:



```
[root@dockerhost ~]# su - dockertest
[dockertest@dockerhost ~]$ mkdir ~/dir1
[dockertest@dockerhost ~]$ docker run -it -v ~/dir1:/dir1 fedora bash
bash-4.3# touch /dir1/file1
touch: cannot 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.log`) on the host, we will see messages similar to the following:



```
type=AVC msg=audit(1430564404.997:1256): avc:  denied  { write } for  pid=5756 comm="touch" name="dir1" dev="dm-2" ino=5242889
scontext=system_u:system_r:svirt_lxc_net_t:s0:c157,c350 tcontext=unconfined_u:object_r:user_home_t:s0 tclass=dir permissive=0
```

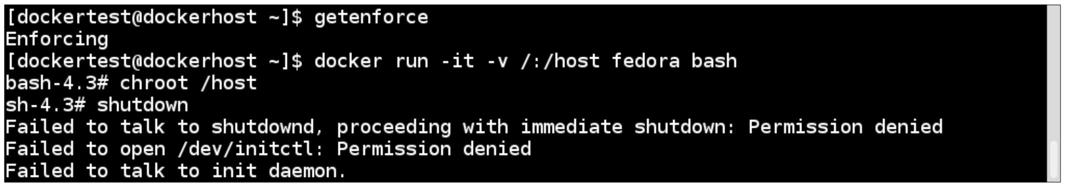
The `s0:c157,c350` label is the MCS label on the container.

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...

- If SELinux is in 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:



```
[dockertest@dockerhost ~]$ getenforce
Enforcing
[dockertest@dockerhost ~]$ docker run -it -v /:/host fedora bash
bash-4.3# chroot /host
sh-4.3# 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.
```

- 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. Visit the *Adjusting SELinux labels* section of `http://opensource.com/business/15/3/docker-security-tuning`.
- Setting up MLS with Docker containers is also possible. Visit the *Multi Level Security mode* section of `http://opensource.com/business/15/3/docker-security-tuning`.

See also

- *The SELinux Coloring Book*; visit `https://people.redhat.com/duffy/selinux/selinux-coloring-book_A4-Stapled.pdf`

Allowing writes to volume mounted from the host with SELinux ON

As we saw in the earlier recipe, when SELinux is configured, a nonprivileged 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.

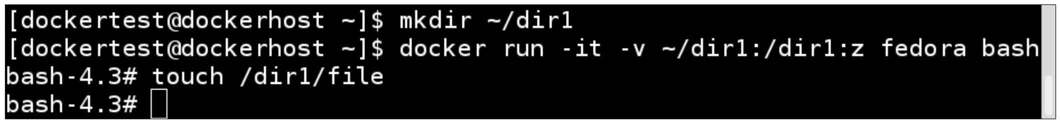
Getting ready

A Fedora/RHEL/CentOS host with the latest version of Docker installed, which can be accessed through a Docker client. Also, SELinux is set to enforcing mode and the Docker daemon is configured to use SELinux.

How to do it…

1. Mount the volume with the `z` or `Z` option as follows:

   ```
   $ docker run -it -v /tmp/:/tmp/host:z docker.io/fedora bash
   $ docker run -it -v /tmp/:/tmp/host:Z docker.io/fedora bash
   ```



```
[dockertest@dockerhost ~]$ mkdir ~/dir1
[dockertest@dockerhost ~]$ docker run -it -v ~/dir1:/dir1:z fedora bash
bash-4.3# touch /dir1/file
bash-4.3# 
```

How it works…

While mounting the volume, Docker will relabel to the volume to allow access. From the man page of Docker run.

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 labels allow 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 breakdown the power of a root user inside a container

In simple terms, with capabilities, we can breakdown the power of a root user. From the man page for *capabilities*:

> *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 nonzero). 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 superuser into distinct units, known as capabilities, which can be independently enabled and disabled. Capabilities are a per-thread attribute.*

Some example capabilities are:

- `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 the kernel. There is one capability, `CAP_SYS_ADMIN`, that catches all capabilities; this is used whenever in doubt.

In version 1.2, Docker added some features to add or remove the 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 the process capabilities
- `CAP_SYS_MODULE`: This inserts/removes the kernel modules
- `CAP_SYS_RAWIO`: This modifies the kernel memory
- `CAP_SYS_PACCT`: This configures process accounting
- `CAP_SYS_NICE`: This modifies the priority of processes

- `CAP_SYS_RESOURCE`: This overrides the resource limits
- `CAP_SYS_TIME`: This modifies the system clock
- `CAP_SYS_TTY_CONFIG`: This configures `tty` devices
- `CAP_AUDIT_WRITE`: This writes the audit log
- `CAP_AUDIT_CONTROL`: This configures the audit subsystem
- `CAP_MAC_OVERRIDE`: This ignores the kernel MAC policy
- `CAP_MAC_ADMIN`: This configures MAC configuration
- `CAP_SYSLOG`: This modifies kernel printk behavior
- `CAP_NET_ADMIN`: This configures the network
- `CAP_SYS_ADMIN`: This helps you catch all the 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 the capabilities for the container, you can use the `--cap-add` and `--cap-drop` options respectively.

Getting ready

A host with the latest version of Docker installed, which can be accessed through a Docker client.

How to do it...

1. To drop capabilities, run a command similar to the following:

   ```
   $ docker run --cap-drop <CAPABILITY> <image> <command>
   ```

   To remove the `setuid` and `setgid` capabilities from the container so that it cannot run binaries, which have these bits set, run the following command:

   ```
   $ docker run -it --cap-drop  setuid --cap-drop setgid fedora
   bash
   ```

2. Similarly, to add capabilities, run a command similar to the following:

   ```
   $ docker run --cap-add <CAPABILITY> <image> <command>
   ```

   To add all the capabilities and just drop `sys-admin`, run the following command:

   ```
   $ docker run -it --cap-add all --cap-drop sys-admin fedora
   bash
   ```

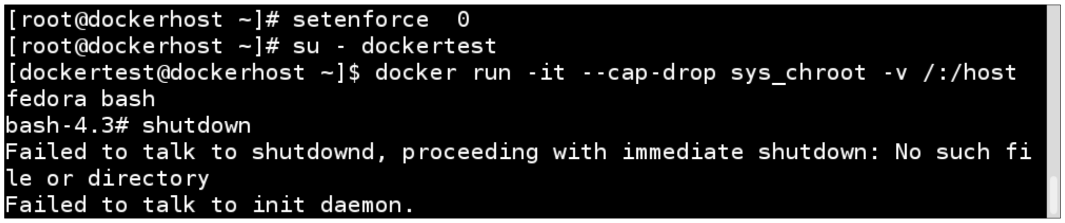
How it works...

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, through which 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 run -it --cap-drop  sys_chroot -v /:/host  fedora bash
$ shutdown
```



```
[root@dockerhost ~]# setenforce  0
[root@dockerhost ~]# su - dockertest
[dockertest@dockerhost ~]$ docker run -it --cap-drop sys_chroot -v /:/host
fedora bash
bash-4.3# shutdown
Failed to talk to shutdownd, proceeding with immediate shutdown: No such fi
le or directory
Failed to talk to init daemon.
```

See also

- Dan Walsh's articles on opensource.com at `http://opensource.com/business/14/9/security-for-docker`.
- The Docker 1.2 release notes at `http://blog.docker.com/2014/08/announcing-docker-1-2-0/`.
- There are efforts on to selectively disable system calls from container processes to provide tighter security. Visit the *Seccomp* section of `http://opensource.com/business/15/3/docker-security-future`.
- Similar to custom namespaces and capabilities with version 1.6, Docker supports the `--cgroup-parent` flag to pass specific Cgroup to run containers. `https://docs.docker.com/v1.6/release-notes/`.

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 would 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

A host with the latest version of Docker installed, which can be accessed through a Docker client.

How to do it…

1. To share the host network namespace with the container, run the following command:

   ```
   $ docker run -it  --net=host fedora bash
   ```

   If you see the network details inside the container, run the following command:

   ```
   $ ip a
   ```

   You will see a result same as the host.

2. To share the host network, PID, and IPC namespaces with the container, run the following command:

   ```
   $ docker run -it --net=host --pid=host --ipc=host fedora bash
   ```

How it works…

Docker does not create separate namespaces for containers when such arguments are passed to the container.

There's more…

For hosts that are built to run just containers, such as Project Atomic (`http://www.projectatomic.io/`), which we saw in *Chapter 8*, *Docker Orchestration and Hosting Platforms*, doesn't have debugging tools such as `tcpdump` and `sysstat` on the host system. So we have created containers with those tools and have access to host resources. In such cases, sharing namespaces between the host and the container becomes handy. You can read more about it at the following links:

- `http://developerblog.redhat.com/2014/11/06/introducing-a-super-privileged-container-concept/`
- `http://developerblog.redhat.com/2015/03/11/introducing-the-rhel-container-for-rhel-atomic-host/`

See also

- Dan Walsh's documentation on Docker Security at `http://opensource.com/business/15/3/docker-security-tuning`

10
Getting Help and Tips and Tricks

In this chapter, we will see 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 execution driver of Docker
- 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 the free node, such as `#docker` and `#docker-dev`, to discuss Docker in general and dev-related stuff respectively. Similarly, Docker has two mailing lists:

- The Docker user list available at `https://groups.google.com/forum/#!forum/docker-user`
- The Docker dev list available at `https://groups.google.com/forum/#!forum/docker-dev`

While working on Docker, if you find any bugs, you can report them on GitHub at `https://github.com/docker/docker/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 a YouTube channel, which are great learning resources and can be found at `https://forums.docker.com/` and `https://www.youtube.com/user/dockerrun` respectively.

There are many Docker meet up groups around the world, where you meet like-minded individuals and learn by sharing experiences at `https://www.docker.com/community/meetups/`.

In this chapter, I'll also put down a few tips and tricks, which will help you to work better with Docker.

Starting Docker in debug mode

We can start Docker in debug mode to debug logs.

Getting ready

Install Docker on the system.

How to do it...

1. Start the Docker daemon with the debug option `-D`. To start from the command line, you can run the following command:

   ```
   $ docker -d -D
   ```

2. You can also add the `--debug/-D` option in the Docker configuration file to start in debug mode.

How it works...

The preceding command would start the Docker in the 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:

```
[info] POST /v1.15/containers/create
[99430521] +job create()
......
......
```

Building a Docker binary from the source

Sometimes it becomes necessary to build a Docker binary from the source for testing a patch. It is very easy to build the Docker binary from the source.

Getting ready

1. Download the Docker source code using `git`:

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

2. Install `make` on Fedora:

   ```
   $ yum install -y make
   ```

3. Make sure Docker is running on the host on which you are building the code and you can access it through the Docker client, as the build we are discussing here happens inside a container.

How to do it…

1. Go inside the cloned directory:

   ```
   $ cd docker
   ```

2. Run the `make` command:

   ```
   $ sudo make
   ```

How it works…

This will create a container and compile the code inside that from the master branch. Once finished, it will spit out the binary inside `bundles/<version>/binary`.

There's more…

- Similar to the source code, you can build the documentation as well:

  ```
  $ sudo make docs
  ```

- You can also run tests with the following command:

  ```
  $ sudo make test
  ```

See also

- Look at the documentation on the Docker website `https://docs.docker.com/contributing/devenvironment/`

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 `yum -y update`. Let's see how we can do that in this recipe.

Getting ready

Get a Dockerfile to build the image.

How to do it...

1. While building the image, pass the `--no-cache` option as follows:

```
$ docker build -t test --no-cache - < Dockerfile
```

How it works...

The `--no-cache` option will discard any cached layer and build one Dockerfile by following the instructions.

There's more...

Sometimes, we also want to discard the cache after only a few instructions. In such cases, we can add any arbitrary command which doesn't affect the image, such as creation or setting up an environment variable.

Building your own bridge for container communication

As we know, when the Docker daemon starts, it creates a bridge called `docker0` and all the containers would get the IP from it. Sometimes we might want to customize those settings. Let's see how we can do that in this recipe.

Getting ready

I am assuming you already have a Docker set up. On the Docker host, stop the Docker daemon. On Fedora, use the following command:

```
$ systemctl stop docker
```

How to do it...

1. To remove the default `docker0` bridge, use the following command:

   ```
   $ sudo ip link set dev docker0 down
   $ sudo brctl delbr docker0
   ```

2. To create the custom bridge, use the following command:

   ```
   $ sudo brctl addbr br0
   $ sudo ip addr add 192.168.2.1/24 dev br0
   $ sudo ip link set dev bridge0 up
   ```

3. Update the Docker configuration file to start with the bridge we created earlier. On Fedora, you can update the configuration file as follows:

   ```
   $ sed -i '/^OPTIONS/ s/$/ --bridge br0/' /etc/sysconfig/docker
   ```

4. To start the Docker daemon, use the following command:

   ```
   $ systemctl start docker
   ```

How it works...

The preceding steps will create a new bridge and it will assign the IP from 192.168.2.0 subnet to the containers.

There's more...

You can even add an interface to the bridge.

See also

- The documentation on the Docker website at `https://docs.docker.com/articles/networking/`

Changing the default execution driver of Docker

As we know, libcontainer is the default execution driver. There is legacy support for LXC userspace tools (`https://linuxcontainers.org/`). Keep in mind that LXC is not the primary development environment.

Getting ready

Install Docker on the system.

How to do it…

1. Start the Docker daemon with the `-e lxc` option, as follows:

```
$ docker -d -e lxc
```

You can also add this option in the configuration file of Docker, depending on the distribution.

How it works…

Docker uses LXC tools to access kernel features, such as Namespaces and Cgroups to run containers.

See also

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#docker-exec-driver-option`

Selecting the logging driver for containers

With the release of Docker 1.6, a new feature has been added to select the logging driver while starting the Docker daemon. Currently, three types of logging drivers are supported:

- none
- json-file (default)
- syslog

Getting ready

Install Docker 1.6 or above on the system.

How to do it...

1. Start the Docker daemon with the required logging driver as follows:

```
$ docker -d --log-driver=none
$ docker -d --log-driver=syslog
```

You can also add this option in the configuration file of Docker, depending on the distribution.

The `docker logs` command will just support the default logging driver JSON file.

How it works...

Depending on the log driver configuration, Docker daemon selects the corresponding logging driver.

There's more...

There is work in progress to add `journald` as one of the logging drivers. It will be available from Docker 1.7 at `http://www.projectatomic.io/blog/2015/04/logging-docker-container-output-to-journald/`.

See also

- The documentation on the Docker website `http://docs.docker.com/reference/run/#logging-drivers-log-driver`

Getting real-time Docker events for containers

As we will be running many containers in production, it will 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 or containers.

Getting ready

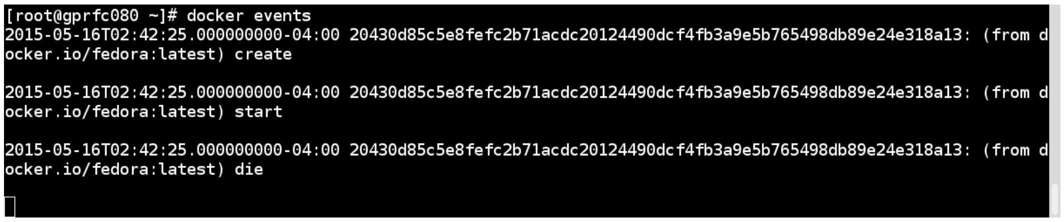
Make sure the Docker daemon is running on the host and you can connect through the Docker client.

How to do it...

1. Start the Docker events logging with the following command:

```
$ docker events
```

2. From the other terminal, do some container/image-related operation and you will see a result similar to the following screenshot on the first terminal:



```
[root@gprfc080 ~]# docker events
2015-05-16T02:42:25.000000000-04:00 20430d85c5e8fefc2b71acdc20124490dcf4fb3a9e5b765498db89e24e318a13: (from d
ocker.io/fedora:latest) create

2015-05-16T02:42:25.000000000-04:00 20430d85c5e8fefc2b71acdc20124490dcf4fb3a9e5b765498db89e24e318a13: (from d
ocker.io/fedora:latest) start

2015-05-16T02:42:25.000000000-04:00 20430d85c5e8fefc2b71acdc20124490dcf4fb3a9e5b765498db89e24e318a13: (from d
ocker.io/fedora:latest) die
```

After the events collection started, I created a container to just echo something. As you can see from the preceding screenshot, a container got created, started, and died.

How it works...

With Docker events, Docker starts listing different events.

There's more...

- You can use the `--since` or `--until` option with Docker events to narrow down results for a 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 CentOS, use the following command:

    ```
    $ docker events --filter 'image=docker.io/centos:centos7'
    ```

- To list events from the specific container, use the following command:

```
docker events --filter
'container=b3619441cb444b87b4d79a8c30616ca70da4b5aa8fdc5d8a4
8d23a2082052174'
```

See also

- The documentation on the Docker website `https://docs.docker.com/reference/commandline/cli/#events`

Index