Docker Cookbook

Docker is a Linux container engine that allows you to create consistent, stable, and production-quality environments with containers.

You will start by installing Docker and understanding and working with containers and images. You then proceed to learn about network and data management for containers. The book explores the RESTful APIs provided by Docker to perform different actions such as image/container operations. Finally, the book explores logs and troubleshooting Docker to solve issues and bottlenecks. This book will also help you understand Docker use cases, orchestration, security, ecosystems, and hosting platforms to make your applications easy to deploy, build, and collaborate on.

What this book will do for you...

- Install and set up Docker on different environments
- Work with Docker images and containers to Dockerize applications
- Create services with Docker to enable the swift development and deployment of applications
- Make optimum use of Docker in a testing environment do complete CI/CD
- Plan efficient working with Docker APIs, orchestration, and hosting platform such as Project Atomic
- Learn the various use cases available for Docker
- Troubleshoot, maintain, and optimize your Docker services effectively

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

Quick answers to common problems

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

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Introduction and Installation'
- A synopsis of the book’s content
- More information on Docker Cookbook
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/.
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.
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.
Introduction and Installation

The following diagram depicts such a setup:


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:

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.

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 11654  2656 ?  S  07:37 0:00 sh container.sh
root  80 0.0  0.0 9084  840 ?  S+ 07:43 0:00 grep container
```

On the host:

```
[root@dockerhost ~]# ps aux | grep container
root 29778 0.0 0.0 11654 2660 pts/3  S  07:37 0:00 sh container.sh
root 29912 0.0 0.0 113604 2160 pts/4  S+ 07:45 0:00 grep --color=auto container
```

**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 *vethcfla*. 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
2: eth0: <BROADCAST,MULTICAST,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 fe80::242: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:fe69:9799/64 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 state UP group default
    link/ether f6:c4:52:c4:68:ba brd ff:ff:ff:ff:ff:ff
    inet6 fe80::f4c4:52ff:fe46:68ba/64 scope link
      valid_lft forever preferred_lft forever
```

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
```

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     MEMORY        CPU ID       COMMAND
root@dockerhost:~# docker ps
CONTAINER ID   IMAGE                COMMAND                  CREATED             STATUS     MEMORY        CPU ID       COMMAND
```

For more details, visit [https://docs.docker.com/articles/runmetrics/](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:

   $ docker -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:

   $ docker -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.
Introduction and Installation

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

<table>
<thead>
<tr>
<th>REPOSITORY</th>
<th>IMAGE ID</th>
<th>CREATED</th>
<th>VIRTUAL SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>docker.io/mysql</td>
<td>56f320b6ba0d</td>
<td>12 days ago</td>
<td>202.5 MB</td>
</tr>
<tr>
<td>docker.io/fedora</td>
<td>93be8892dfb8</td>
<td>12 days ago</td>
<td>241.3 MB</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>COMMAND</th>
<th>CREATED</th>
<th>STATUS</th>
<th>boom</th>
<th>NAMES</th>
</tr>
</thead>
<tbody>
<tr>
<td>e4289e6f0e20</td>
<td>docker.io/fedora:latest</td>
<td>docker run</td>
<td>3 seconds ago</td>
<td>Up 2 seconds</td>
<td>721</td>
<td>721</td>
</tr>
</tbody>
</table>

### 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 diagram](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.

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:

```bash
$ 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:
   ```bash
   $ 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

<table>
<thead>
<tr>
<th>NAME</th>
<th>ACTIVE</th>
<th>DRIVER</th>
<th>STATE</th>
<th>URL</th>
<th>SWARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>cookbook</td>
<td>*</td>
<td>google</td>
<td>Running</td>
<td>tcp://104.154.84.152:2376</td>
<td></td>
</tr>
</tbody>
</table>

   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

<table>
<thead>
<tr>
<th>NAME</th>
<th>ACTIVE</th>
<th>DRIVER</th>
<th>STATE</th>
<th>URL</th>
<th>SWARM</th>
</tr>
</thead>
</table>

   Here, dev is the machine name. By default, the latest deployed machine becomes primary.
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/
Where to buy this book

You can buy Docker Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.