docker

QUALITY THOUGHT

INFOSYSTEMS (INDIA) PVT. LTD.

Table of Contents

The basic idea

The basic idea behind Docker is to pack an application with all of its dependencies (let it be binaries, libraries, configuration files, scripts, jars, and so on) into a single, standardized unit for software development and deployment. Docker containers wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, and system libraries-anything you can install on a server. This guarantees that it will always run in the same way, no matter what environment it will be deployed in. With Docker, you can build a Node.js or Java project (but you are of course not limited to those two) without having to install Node.js or Java on your host machine. Once you're done with it, you can just destroy the Docker image, and it's as though nothing ever happened. It's not a programming language or a framework; rather, think of it as a tool that helps solve common problems such as installing, distributing, and managing the software. It allows programmers and DevOps to build, ship, and run their code anywhere.

You may think that Docker is a virtualization engine, but it's far from it as we will explain in a while.

Containerization versus virtualization

To fully understand what Docker really is, first we need to understand the difference between traditional virtualization and containerization. Let's compare those two technologies now.

Traditional virtualization

A traditional virtual machine, which represents the hardware-level virtualization, is basically a complete operating system running on top of the host operating system. There are two types of virtualization hypervisor: **Type 1** and **Type 2**. Type 1 hypervisors provide server virtualization on bare metal hardware—there is no traditional end user's operating system. Type 2 hypervisors, on the other hand, are commonly used as a desktop virtualization—you run the virtualization engine on top of your own operating system. There are a lot of use cases that would take advantage of using virtualization—the biggest asset is that you can run many virtual machines with totally different operating systems on a single host.

Virtual machines are fully isolated, hence very secure. But nothing comes without a price. There are many drawbacks—they contain all the features that an operating system needs to have: device drivers, core system libraries, and so on. They are heavyweight, usually resource-hungry, and not so easy to set up—virtual machines require full installation. They require more computing resources to execute. To successfully run an application on a virtual machine, the hypervisor needs to first import the virtual machine and then power it up, and this takes time. Furthermore, their performance gets substantially degraded. As a result, only a few virtual machines can be provisioned and made available to work on a single machine.

Containerization

The Docker software runs in an isolated environment called a **Docker container**. A Docker container is not a virtual machine in the popular sense. It represents operating system virtualization. While each virtual machine image runs on an independent guest OS, the Docker images run within the same operating system kernel. A container has its own filesystem and environment variables. It's self-sufficient. Because of the containers run within the same kernel, they utilize fewer system resources. The base container can be, and usually is, very lightweight. It's worth knowing that Docker containers are isolated not only from the underlying operating system, but from each other as well. There is no overhead related to a classic virtualization hypervisor and a guest operating system. This allows achieving almost bare metal, near native performance. The boot time of a **dockerized** application is usually very fast due to the low overhead of containers. It is also possible to speed up the roll-out of hundreds of application containers in seconds and to reduce the time taken provisioning your software.

Be aware that containers cannot substitute virtual machines for all use cases. A thoughtful evaluation is still required to determine what is best for your application. Both solutions have their advantages. On one hand we have the fully isolated, secure virtual machine with average performance and on the other hand, we have the containers that are missing some of the key features (such as total isolation), but are equipped with high performance that can be provisioned swiftly. Let's see what other benefits you will get when using Docker containerization.

As you can see, Docker is quite different from the traditional virtualization engines. Be aware that containers are not substitutes for virtual machines for all use cases. A thoughtful evaluation is still required to determine what is best for your application. Both solutions have their advantages. On one hand we have the fully isolated, secure virtual machine with average performance, and on the other hand, we have containers that are missing some of the key features (such as total isolation), but are equipped with high performance and can be provisioned swiftly.

Understanding Docker

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

Docker 1.13 has implemented management commands that have started to sort commands into useful categories. This helps understand what commands are associated with what actions; that is, what commands pertain to containers, images, nodes, and and so on. This is more than likely going to be the new standard moving forward with versions beyond Docker 1.13. Now you will still see the following output in Docker 1.13:

```
$ docker
Usage: docker COMMAND
A self-sufficient runtime for containers
Options:
```

--config string Location of client config files (default "/home/spg14/.docker")
-D, --debug Enable debug mode
--help Print usage
-H, --host list Daemon socket(s) to connect to (default [])
-l, --log-level string Set the logging level ("debug", "info", "warn", "error", "fatal") (default "info")
--tls Use TLS; implied by --tlsverify
--tlscacert string Trust certs signed only by this CA (default "/home/spg14/.docker/ca.pem")
--tlscert string Path to TLS certificate file (default "/home/spg14/.docker/cert.pem")
--tlskey string Path to TLS key file (default "/home/spg14/.docker/key.pem")
--tlsverify Use TLS and verify the remote
-v, --version Print version information and quit

Management commands:container: Manage containersimage: Manage imagesnetwork: Manage networksnode: Manage Swarm nodesplugin: Manage pluginssecret: Manage Docker secretsservice Manage servicesstack Manage Docker stacksswarm Manage Swarmsystem Manage Dockervolume Manage volumes

Commands:
attach Attach to a running container
build Build an image from a Dockerfile
commit Create a new image from a container's changes
cp Copy files/folders between a container and the local filesystem
create Create a new container
diff Inspect changes on a container's filesystem
events Get real time events from the server
exec Run a command in a running container
export Export a container's filesystem as a tar archive
history Show the history of an image
images List images
import Import the contents from a tarball to create a filesystem image
info Display system-wide information
inspect Return low-level information on Docker objects
kill Kill one or more running containers
load Load an image from a tar archive or STDIN
login Log in to a Docker registry
logout Log out from a Docker registry
logs Fetch the logs of a container
pause Pause all processes within one or more containers
port List port mappings or a specific mapping for the container
ps List containers
pull Pull an image or a repository from a registry

```
push Push an image or a repository to a registry
rename Rename a container
restart Restart one or more containers
rm Remove one or more containers
rmi Remove one or more images
run Run a command in a new container
save Save one or more images to a tar archive (streamed to STDOUT by default)
search Search the Docker Hub for images
start Start one or more stopped containers
stats Display a live stream of container(s) resource usage statistics
stop Stop one or more running containers
tag Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
top Display the running processes of a container
unpause Unpause all processes within one or more containers
update Update configuration of one or more containers
version Show the Docker version information
wait Block until one or more containers stop, then print their exit codes
Run 'docker COMMAND --help' for more information on a command.
```

Docker 1.13 allows you to hide the legacy commands and see what the "future" is going to look like:

```
export DOCKER_HIDE_LEGACY_COMMANDS=1
```

```
docker
Usage: docker COMMAND
A self-sufficient runtime for containers
Options:
--config string Location of client config files (default "/home/spg14/.docker")
-D, --debug Enable debug mode
--help Print usage
-H, --host list Daemon socket(s) to connect to (default [])
-l, --log-level string Set the logging level ("debug", "info", "warn", "error", "fatal") (default "info")
--tls Use TLS; implied by --tlsverify
--tlscacert string Trust certs signed only by this CA (default "/home/spg14/.docker/ca.pem")
--tlscert string Path to TLS certificate file (default "/home/spg14/.docker/cert.pem")
--tlskey string Path to TLS key file (default "/home/spg14/.docker/key.pem")
--tlsverify Use TLS and verify the remote
-v, --version Print version information and quit
```

Management Commands:
container Manage containers
image Manage images
network Manage networks
node Manage Swarm nodes
plugin Manage plugins
secret Manage Docker secrets
service Manage services
stack Manage Docker stacks
swarm Manage Swarm
system Manage Docker
volume Manage volumes

Commands:
build Build an image from a Dockerfile
login Log in to a Docker registry
logout Log out from a Docker registry
run Run a command in a new container
search Search the Docker Hub for images
version Show the Docker version information
Run 'docker COMMAND --help' for more information on a command.

You can now view the management commands sub-commands by executing them in a terminal window.

$ docker image
Usage: docker image COMMAND
Manage images

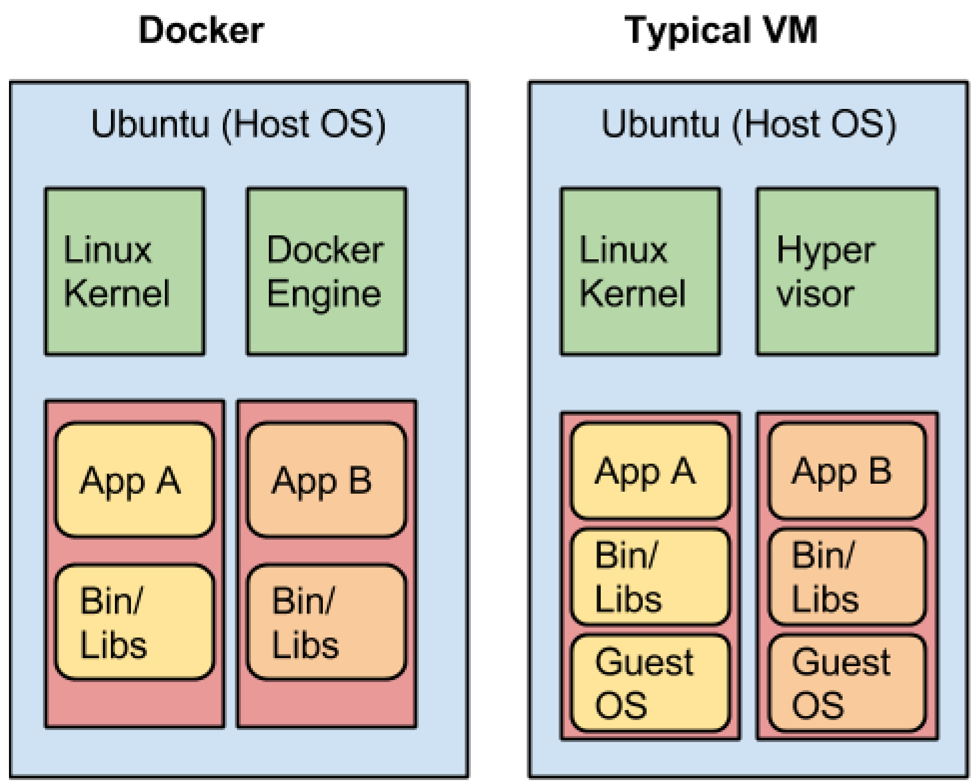
Options:
--help Print usage

Commands:
build Build an image from a Dockerfile
history Show the history of an image
import Import the contents from a tarball to create a filesystem image
inspect Display detailed information on one or more images
load Load an image from a tar archive or STDIN
ls List images
prune Remove unused images
pull Pull an image or a repository from a registry
push Push an image or a repository to a registry
rm Remove one or more images
save Save one or more images to a tar archive (streamed to STDOUT by default)
tag Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
Run 'docker image COMMAND --help' for more information on a command.

Difference between Docker and typical VMs

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

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



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

Dockerfile

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

```
FROM ubuntu:latest
MAINTAINER info@qualitythought.in
RUN apt-get update && apt-get install -y apache2
ADD 000-default.conf /etc/apache2/sites-available/
RUN chown root:root /etc/apache2/sites-available/000-default.conf
EXPOSE 80
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

These are the typical items you would find in a basic Dockerfile. The first line states the image we want to start off with when we build the container. In this example, we will be using Ubuntu; the item after the colon can be called if you want a specific version of it. In this case, I am just going to say use the latest version of Ubuntu; but you will also specify trusty, precise, raring, and so on. The second line is the line that is relevant to the maintainer of Dockerfile. This is for people to contact you if they have any questions or find any errors in your file. Typically, most people just include their name and e-mail address. The maintainer instruction can still be used, but it is recommended to use the LABEL instruction as it provides more flexibility.

The next line is a typical line you will see while pulling updates and packages in an Ubuntu environment. You might think they should be separate and wonder why they should be put on the same line separated by &&. Well, in the Dockerfile, it helps by only having to run one process to encompass the entire line. If you were to split it into separate lines, it would have to run one process, finish the process, then start the next process, and finish it. With this, it helps speed up the process by pairing the processes together. They still run one after another, but with more efficiency.

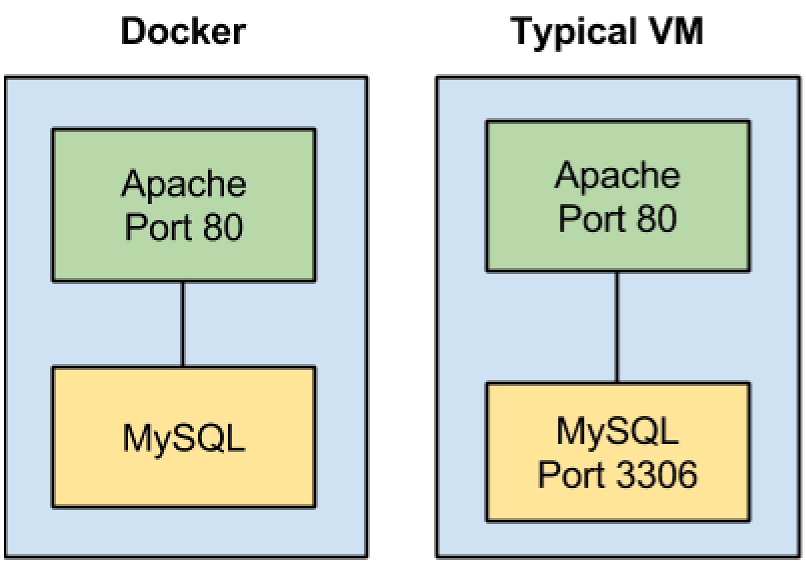
The next two lines complement each other. The first adds your custom configurations to the path you specified and changes the owner and group owner to the root user. The EXPOSE line will expose the ports to anything external to the container and to the host it is running on. (This will, by default, expose the container externally beyond the host, unless the firewall is enabled and protecting it.) The last line is the command that is run when the container is launched. This particular command in a Dockerfile should only be used once. If it is used more than once, the last CMD in the Dockerfile will be launched upon the container that is running. This also helps emphasize the one process per container rule.

The idea is to spread out the processes so that each process runs in its own container, thus the value of the containers will become more understandable. Essentially, something that runs in the foreground, such as the earlier command to keep the Apache running in the foreground. If we were to use CMD ["service apache2 start"], the container would start and then immediately stop. There is nothing to keep the container running. You can also have other instructions, such as ENV to specify the environmental variables that users can pass upon runtime. These are typically used and are useful while using shell scripts to perform actions such as specifying a database to be created in MySQL or setting permission databases. We will be covering these types of items in a later chapter, so don't worry about looking them up right now.

There are two new instructions you can utilize inside your Dockerfile since the last edition of this book. Those are SHELL and HEALTHCHECK. The SHELL instructions will allow you to override the default shell that is used to run commands. This is very useful when it comes to Windows. Windows has both the traditional command prompt as well as powershell. By default Windows will use the command prompt to execute commands you call from inside your Dockerfile. By using the SHELL instruction you can override this and use the very useful powershell on Windows. The HEALTHCHECK directive tells Docker how to perform a health check on a container to ensure it's operating as required. If it isn't the container will exit instead of being stuck in an infinite loop. This instruction allows you to specify number of retires, duration timeout, and a duration internal to perform the health check before exiting. With HEALTHCHECK there can only be one specified per Dockerfile.

Docker Networking/Linking

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



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

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

Docker installers/installation

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

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

Types of Installers

With the various types of installers listed earlier, there are different ways Docker actually operates on the operating system. Docker natively runs on Linux; so if you are using Linux, then it's pretty straightforward how Docker runs right on your system. However, if you are using Windows or Mac OS X, then it operates a little differently, since it relies on using Linux.

With the newer Mac and Windows operating systems the installers have changed. They no longer are packaged as the Docker toolbox. They do still contain all the same packages; such as the Docker CLI, Docker Machine, and Docker Compose. However as we will soon learn they don't rely on needing something such as Virtualbox to launch the Linux instance required. How they accomplish this is by utilizing the built-in virtualization that comes with the newer versions of Mac and Windows operating systems. You will want to check on the Docker documentation to ensure that your system can handle the newer installers.

If you are not running a newer operating system on Mac or Windows then you will need to use the Docker Toolbox. With these operating systems, they need Linux in some sort of way, thus enters the virtual machine needed to run the Linux part that Docker operates on, which is called boot2docker. The installers for both Windows and Mac OS X are bundled with the boot2docker package alongside the virtual machine software that, by default, is the Oracle VirtualBox

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

Docker Toolbox piece	**Mac OS X**	**Windows**
Docker Client	Yes	Yes
Docker Machine	Yes	Yes
Docker Compose	Yes	No
Docker Kitematic	Yes	Yes
VirtualBox	Yes	Yes

Docker Machine - the new boot2docker

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

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

Note: Docker Machine is not required in the newer installers. It is only required if you want to manage multiple Docker hosts.

Installing the Docker Machine

The Docker Engine is built on top of the Linux kernel and it extensively leverages Linux kernel features such as namespaces and cgroups. Due to the burgeoning popularity of Docker, it is now being packaged by all the major Linux distributions so that they can retain their loyal users as well as attract new users. You can install the Docker Engine using the corresponding packaging tool of the Linux distribution, for example, using the apt-get command for Debian and Ubuntu, and the yum command for Red Hat, Fedora, and CentOS. Alternatively, you can use the fully automated install script, which will do all the hard work for you behind the scenes.

If you are a Mac or Microsoft Windows user, you can run Docker on Linux emulations (VMs). There are multiple solutions available to run Docker using Linux VM, which is explained in a later subsection

Installing Docker on Ubuntu

https://docs.docker.com/engine/installation/linux/ubuntu/

Installing Docker on RedHat

https://docs.docker.com/engine/installation/linux/rhel/

Understanding the Docker setup

It is important to understand the Docker components and their versions, storage, and the execution drivers, the file locations, and so on. Incidentally, the quest for understanding the Docker setup will also reveal whether the installation was successful or not. You can accomplish this using two Docker subcommands: `docker version` and `docker info`.

Let's start our Docker journey with the `docker version` subcommand, as shown here:

```
$ sudo docker version
Client:
 Version:      17.03.0-ce
 API version:  1.26
 Go version:   go1.7.5
 Git commit:   60ccb22
 Built:        Thu Feb 23 10:57:47 2017
 OS/Arch:      linux/amd64

Server:
 Version:      17.03.0-ce
 API version:  1.26 (minimum version 1.12)
 Go version:   go1.7.5
 Git commit:   60ccb22
 Built:        Thu Feb 23 10:57:47 2017
 OS/Arch:      linux/amd64
 Experimental: false
```

Although the docker version subcommand lists many lines of text, as a Docker user you should know what these following output lines mean:

- The client version
- The client API version
- The server version
- The server API version

Here, both the client and server are of community edition 17.03.0 and the client API and the server API of version 1.26.

If we dissect the internals of the docker version subcommand, then it will first list the client-related information that is stored locally. Subsequently, it will make a REST API call to the server over HTTP to obtain server-related details.

Learn more about the Docker environment using the docker info subcommand:

ubuntu4docker — ubuntu@ubuntu-xenial: ~ — ssh ◂ vagrant s...

```
$ sudo docker info
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
Server Version: 17.03.0-ce
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 0
 Dirperm1 Supported: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Init Binary: docker-init
containerd version: 977c511eda0925a723debdc94d09459af49d082a
runc version: a01dafd48bc1c7cc12bdb01206f9fea7dd6feb70
init version: 949e6fa
Security Options:
 apparmor
 seccomp
  Profile: default
Kernel Version: 4.4.0-66-generic
Operating System: Ubuntu 16.04.2 LTS
OSType: linux
Architecture: x86_64
CPUs: 2
Total Memory: 992.2 MiB
Name: ubuntu-xenial
ID: GMHP:5H3Z:CLSD:ZJMY:3KTP:627O:BNFN:GSCX:QUOJ:CNGE:GIH3:SPIO
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Registry: https://index.docker.io/v1/
WARNING: No swap limit support
Experimental: false
Insecure Registries:
 127.0.0.0/8
Live Restore Enabled: false
$
```

As you can see, in the output of a freshly installed Docker Engine, the number of Containers and Images is invariably nil. The Storage Driver has been set up as aufs, and the directory has been given the /var/lib/docker/aufs location. The runtime has been set to runc. This command also lists details, such as Logging Driver, Cgroups Driver, Kernel Version, Operating System, CPUs, and Total Memory

Client-server communication

On Linux installations, Docker is usually programmed to carry out the server-client communication using the Unix socket (/var/run/docker.sock). Docker also has an IANA-registered port, which is 2375. However, for security reasons, this port is not enabled by default.

Downloading the first Docker image

Having installed the Docker Engine successfully, the next logical step is to download the images from the Docker Registry. The Docker Registry is an application repository that hosts various applications, ranging from basic Linux images to advanced applications. The docker pull subcommand is used to download any number of images from the registry. In this section, we will download a sample hello-world image using the following command:

```
$ sudo docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
78445dd45222: Pull complete
Digest: sha256:c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Status: Downloaded newer image for hello-world:latest
```

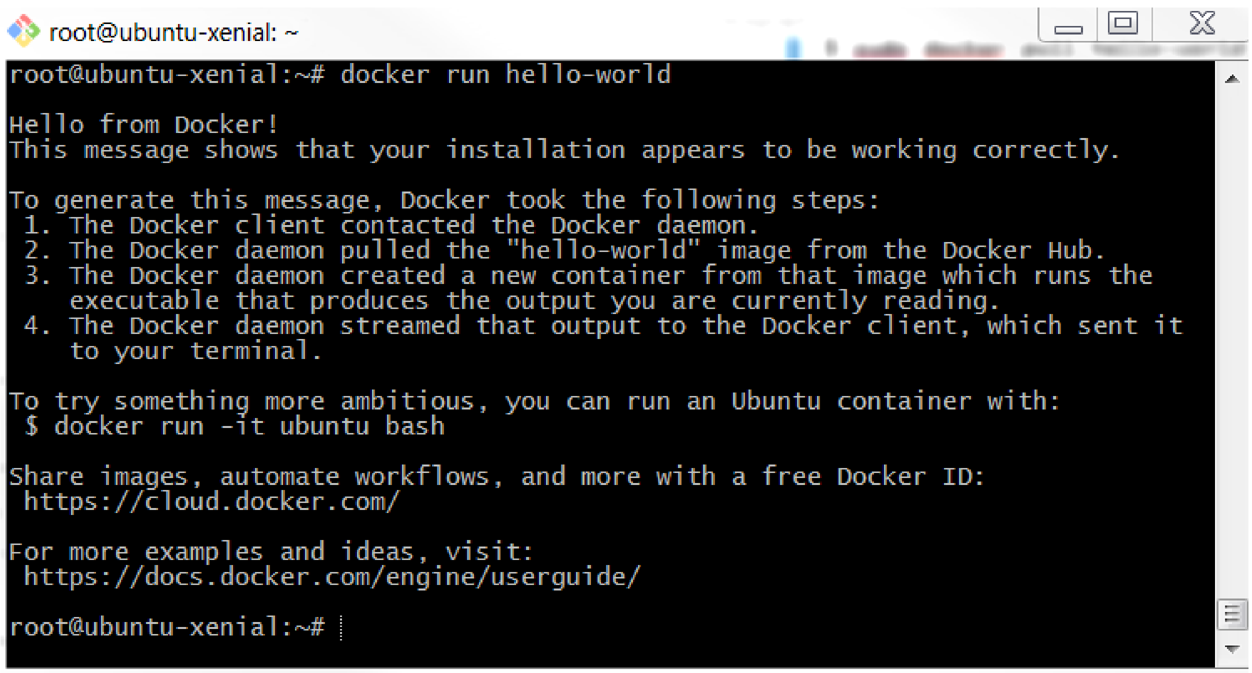
Once the images have been downloaded, they can be verified using the docker images subcommand, as shown here:

```
$ sudo docker images
REPOSITORY    TAG     IMAGE ID       CREATED     VIRTUAL SIZE
```

hello-world latest 48b5124b2768 6 weeks ago 1.84 kB

Running the first Docker container

Now you can start your first Docker container as shown here:



Cool, isn't it? You have set up your first Docker container in no time. In the preceding example, the docker runsubcommand has been used to create a container from the hello-world image.

Troubleshooting Docker containers

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

sudo service docker status

If the Docker service is up-and-running, the Active column (the third from the top) will list the status of the Docker service as active (running), as shown next:

```
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
   Active: active (running) since Thu 2017-02-23 10:52:39 UTC; 2 days ago
     Docs: https://docs.docker.com
 Main PID: 29327 (dockerd)
    Tasks: 22
   Memory: 31.6M
      CPU: 1min 18.943s
   CGroup: /system.slice/docker.service
           ├─29327 /usr/bin/dockerd -H fd://
           └─29336 docker-containerd -l unix:///var/run/docker/libcontainerd/docker-containerd
```

However, if the Active column shows inactive or maintenance as the status, your Docker service is not running. In such cases, restart the Docker service, as shown here:

sudo service docker restart

If you are still experiencing issues with the Docker setup, then you must extract the Docker log, using the journalctl -u docker command, for further investigation.

Handling Docker Containers

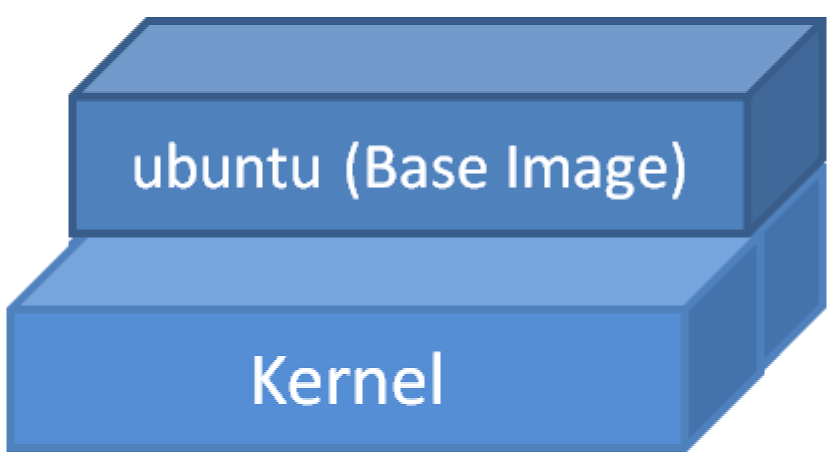
In the previous chapter, we explained the stimulating and sustainable concepts that clearly articulated Docker's way of crafting futuristic and flexible application-aware containers. We discussed all the relevant details of bringing Docker containers into multiple environments (on-premise as well as off-premise). You can easily replicate these Docker capabilities in your own environments to get a rewarding experience. Now, the next logical step for us is to understand container life cycle aspects in a decisive manner. You are to learn the optimal utilization of your own containers as well as those of other third-party containers in an efficient and risk-free way. Containers are to be found, accessed, assessed, and leveraged toward bigger and better distributed applications.

Clarifying Docker terms

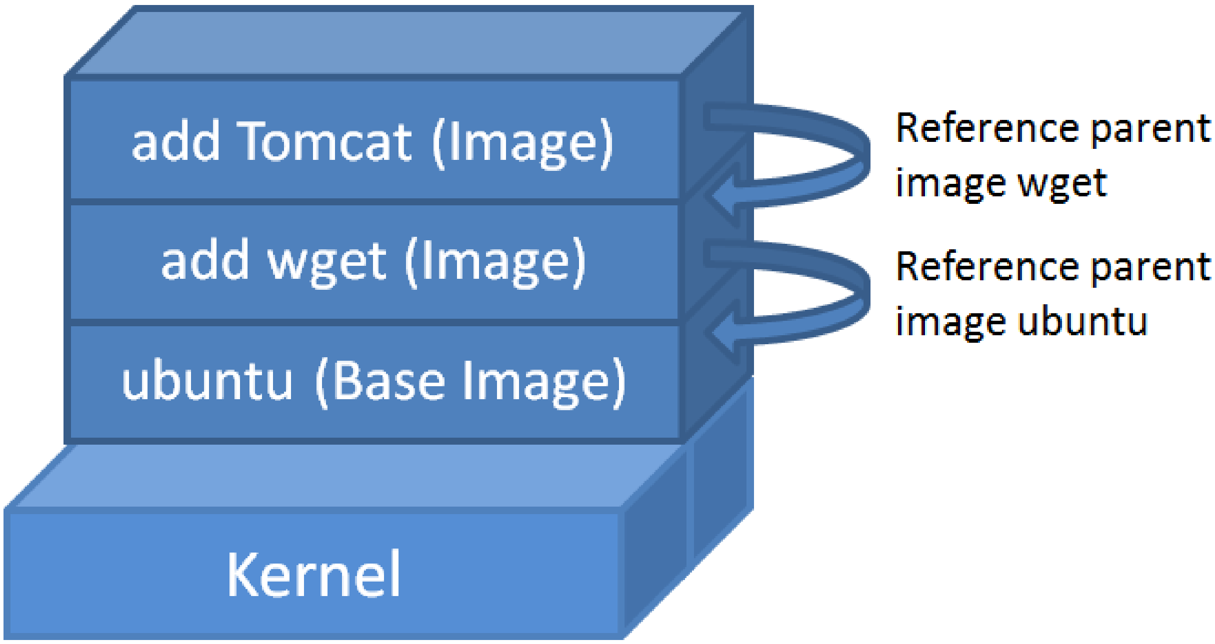
Docker images

A **Docker image** is a collection of all the files that make up an executable software application. This collection includes the application plus all the libraries, binaries, and other dependencies such as the deployment descriptor, just needed to run the application everywhere without hitch or hurdle. These files in the Docker image are read-only and hence the content of the image cannot be altered. If you choose to alter the content of your image, the only option Docker allows is to add another layer with the new changes. In other words, a Docker image is made up of layers, which you can review using the `docker history` subcommand

The Docker image architecture effectively leverages this layering concept to seamlessly add additional capabilities to the existing images in order to meet varying business requirements and also increase the reuse of images. In other words, capabilities can be added to the existing images by adding additional layers on top of that image and deriving a new image. Docker images have a parent-child relationship and the bottom-most image is called the **base image**. The base image is a special image that doesn't have any parent



In the preceding diagram, ubuntu is a base image and it does not have any parent image.



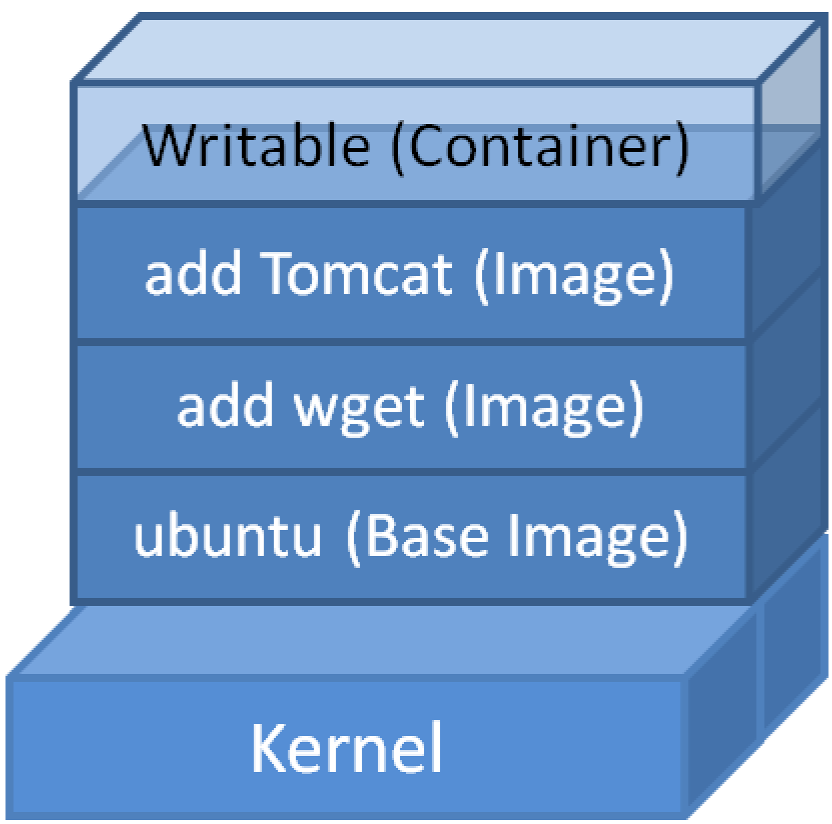
As you can see in the preceding figure, everything starts with a base image and here in this example, it is ubuntu. Further on, the wget capability is added to the image as a layer and the wget image is referencing the ubuntu image as its parent. In the next layer, an instance of the Tomcat application server is added and it refers the wget image as its parent. Each addition that is made to the original base image is stored in a separate layer (a kind of hierarchy gets generated here to retain the original identity). Precisely speaking, any Docker image has to originate from a base image and an image gets continuously enriched in its functionality by getting fresh modules, and this is accomplished by adding an additional module as a new layer on the existing Docker image one by one, as vividly illustrated in the preceding figure.

The Docker platform provides a simple way for building new images or extending existing images. You can also download the Docker images that other people have already created and deposited in the Docker image repositories (private or public). Every image has a unique ID

Docker containers

Docker images are a read-only template of the application stack bundle and they don't have any state associated with them. The Docker container is spun off from the Docker image and it adds a read-write layer on top of the static image layers. If we try to draw a comparison with the object-oriented programming paradigm, Docker images are typically classes, whereas Docker containers are objects (instances of the classes).

The Docker image defines the behavior of the Docker container such as what process to run when the container is started. In the previous chapter, when you invoked `docker run hello-world`, the Docker Engine launched a new container from the `hello-world` Docker image and it went on to output quite a lot of information on the screen. From this example, it is quite evident that Docker images are the basic building block for Docker containers and Docker images prescribe the behavior of Docker containers.



As clearly illustrated in the preceding figure, when the container is spun-off, a writeable (read-write) layer is added on top of the image in order to maintain the application state. There could be several read-only images beneath the container layer (writeable).

Docker Registry

A **Docker Registry** is a place where Docker images can be stored in order to be publicly or privately found, accessed, and used by software developers worldwide for quickly crafting fresh and composite applications without any risks. Because all the stored images will have gone through multiple validations, verifications, and refinements, the quality of those images is really high. You can dispatch your Docker image to the registry so that it is registered and deposited using the `docker push` subcommand. You can download Docker images from the registry using the `docker pull` subcommand.

Docker Registry could be hosted by a third party as a public or private registry, like one of the following registries:

- Docker Hub
- Quay
- Google Container Registry
- AWS Container Registry

Working with Docker images

Now, there is a need for a closer observation of the output of the `docker pull` subcommand, which is the de facto command to download Docker images. Now, in this section, we will use the `busybox` image, one of the smallest but a very handy Docker image, to dive deep into Docker image handling:

```
$ sudo docker pull busybox
Using default tag: latest
latest: Pulling from library/busybox
8ddc19f16526: Pull complete
Digest: sha256:a59906e33509d14c036c8678d687bd4eec81ed7c4b8ce907b888c607f6a1e0e6
Status: Downloaded newer image for busybox:latest
```

If you pay close attention to the output of the docker pull subcommand, you will notice the Using default tag: latest text. The Docker image management capability (the local image storage on your Docker host or on a Docker image registry) enables storing multiple variants of the Docker image. In other words, you could use tags to version your images.

By default, Docker always uses the image that is tagged as latest. Each image variant can be directly identified by qualifying it with an appropriate tag. An image can be tag-qualified by adding a colon (:) between the tag and the repository name (<repository>:<tag>). For demonstration, we will pull the 1.24 tagged version of busybox as shown here:

```
$ sudo docker pull busybox:1.24
1.24: Pulling from library/busybox

385e281300cc: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:8ea3273d79b47a8b6d018be398c17590a4b5ec604515f416c5b797db9dde3ad8
Status: Downloaded newer image for busybox:1.24
```

We are able to pull a specific version of busybox; in this case, it is busybox:1.24. The docker pull command also supports the -a option to download all available image variants. Use this option cautiously because you might end up filling up your disk space.

So far, we downloaded a few Docker images from the repository, and now they are locally available in the Docker host. You can find out the images that are available on the Docker host by running the docker images subcommand:

```
$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
hello-world         latest              c54a2cc56cbb        3 weeks ago
1.848 kB
busybox             latest              2b8fd9751c4c        4 weeks ago
1.093 MB
busybox             1.24                47bcc53f74dc        4 months ago
1.113 MB
```

Evidently, we have three items in the preceding list and to gain a better understanding of these, we need to comprehend the information that is printed out by the docker images subcommand. Here is a list of the possible categories:

- REPOSITORY: This is the name of the repository or image. In the preceding example, the repository names are hello-world and busybox.
- TAG: This is the tag associated with the image, for example 1.24 and latest. One or more tags can be associated with one image.
- IMAGE ID: Every image is associated with a unique ID. The image ID is represented using a 64 hex digit long random number. By default, the docker images subcommand will only show 12 hex digits. You can display all the 64 hex digits using the --no-trunc flag (for example: sudo docker images --no-trunc).
- CREATED: This indicates the time when the image was created.
- SIZE: This category highlights the virtual size of the image.

The Docker Hub

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

The good folks in the Docker community have built a repository of images and they have made it publicly available at a default location, index.docker.io. This default location is called the **Docker Hub**. The docker pullsubcommand is programmed to look for images at this location. Thus, when you pull a busybox image, it is effortlessly downloaded from the default registry. This mechanism helps in speeding up the spinning of Docker containers. The Docker Hub is the official repository that contains all the painstakingly curated images that are created and deposited by the worldwide Docker development community. This so-called cure is implemented for ensuring that all the images stored in the Docker Hub are secure and safe through a host of quarantine tasks. There are additional mechanisms, such as creating the image digest and having content trust, which gives you the ability to verify both the integrity and the publisher of all the data received from a registry over any channel.

There are proven verification and validation methods for cleaning up any knowingly or unknowingly introduced malware, adware, viruses, and so on, from these Docker images. The digital signature is a prominent mechanism of the utmost integrity of the Docker images. Nonetheless, if the official image has been either corrupted or tampered with, then the Docker Engine will issue a warning and then continue to run the image.

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

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

Searching Docker images

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

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

sudo docker search mysql

The search on mysql will list many mysql images, but we will limit it to just five lines by piping it with the head -10 command, as follows:

```
$ sudo docker search mysql | head -10
NAME                       DESCRIPTION                                       STARS     OFFICIAL   AUTOMATED
mysql                      MySQL is a widely used, open-source relati...    2759      [OK]
mysql/mysql-server         Optimized MySQL Server Docker images. Crea...    178                  [OK]
centurylink/mysql          Image containing mysql. Optimized to be li...    46                   [OK]
sameersbn/mysql                                                              36                   [OK]
appcontainers/mysql        CentOS/Debian Based Customizable MySQL Con...    8                    [OK]
marvambass/mysql           MySQL Server based on Ubuntu 14.04               6                    [OK]
alterway/mysql             Docker Mysql                                      2                    [OK]
drupaldocker/mysql         MySQL for Drupal                                  2                    [OK]
azukiapp/mysql             Docker image to run MySQL by Azuki - http:...    2                    [OK]
$
```

As you can see in the preceding search output excerpts, the images are ordered based on their star rating. The search result also indicates whether the image is an official image (curated and hosted by Docker Inc) or not. The search result also indicates whether the image is built using the automation framework provided by Docker Inc. The mysqlimage curated and hosted by Docker Inc has a 2759 star rating, which indicated that this is the most popular mysql image. We strongly recommend that you use the images that are officially hosted by Docker Inc for security reasons, otherwise make sure that the images are provided by trusted and well-known sources. The next image in the list is mysql-server, made available by the third party, mysql, with a 178 star rating. Docker containers are the standard building blocks of distributed applications.

Working with an interactive container

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

In the following example, we are going to launch an interactive container using the ubuntu:16.04 image and /bin/bash as the command:

```
$ sudo docker run -i -t ubuntu:16.04 /bin/bash
```

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

```
Unable to find image 'ubuntu:16.04' locally
16.04: Pulling from library/ubuntu
```

As soon as the download is completed, the container will get launched along with the ubuntu:16.04 image. It will also launch a Bash shell within the container, because we have specified /bin/bash as the command to be executed. This will land us in a Bash prompt, as shown here:

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

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

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

```
root@742718c21816:/# hostname
742718c21816
root@742718c21816:/# id
uid=0(root) gid=0(root) groups=0(root)
root@742718c21816:/# echo $PS1
[e]0;u@h: wa]${debian_chroot:+($debian_chroot)}u@h:w$
```

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

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

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

```
sudo docker ps
CONTAINER ID       IMAGE            COMMAND          CREATED
STATUS           PORTS           NAMES
742718c21816      ubuntu:16.04      "/bin/bash"       About a
minute ago   Up About a minute        jolly_lovelace
```

The docker ps subcommand will list out the following details:

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

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

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

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

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

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

Tracking changes inside containers

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

```
$ sudo docker run -i -t ubuntu:16.04 /bin/bash
```

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

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

Now, we can create three empty files using the touch command, as follows. The first ls -l command will show that there are no files in the directory and the second ls -l command will show that there are three empty files:

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

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

```
$ sudo docker diff d5ad60f174d3
```

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

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

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

Controlling Docker containers

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

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

```
$ sudo docker run -i -t ubuntu:16.04 /bin/bash
root@da1c0f7daa2a:/#
```

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

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

Now, if we switch to the screen/Terminal where we were running the container, we will notice that the container is being terminated. If you observe a little more keenly, then you will also notice the exit text next to the container prompt. This happened due to the SIGTERM handling mechanism of the Bash shell, as shown here:

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

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

```
$ sudo docker ps -a
CONTAINER ID          IMAGE             COMMAND
CREATED               STATUS            PORTS
NAMES
da1c0f7daa2a        ubuntu:16.04          "/bin/bash"
20 minutes ago        Exited (0) 10 minutes ago
desperate_engelbart
$
```

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

Let's start the previously stopped container using the docker start subcommand by specifying the container ID as an argument, as follows:

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

Let's start the previously stopped container using the docker start subcommand by specifying the container ID as an argument, as follows:

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

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

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

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

```
$ sudo docker ps
CONTAINER ID          IMAGE               COMMAND
CREATED               STATUS              PORTS
NAMES
da1c0f7daa2a        ubuntu:16.04        "/bin/bash"        25 minutes ago        Up 3
minutes
desperate_engelbart
$
```

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

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

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

```
$ sudo docker run -i -t ubuntu:16.04 /bin/bash
root@c439077aa80a:/# while true; do date; sleep 5; done
Thu Oct  2 03:11:19 UTC 2016
Thu Oct  2 03:11:24 UTC 2016
Thu Oct  2 03:11:29 UTC 2016
Thu Oct  2 03:11:34 UTC 2016
Thu Oct  2 03:11:59 UTC 2016
Thu Oct  2 03:12:04 UTC 2016
Thu Oct  2 03:12:09 UTC 2016
Thu Oct  2 03:12:14 UTC 2016
Thu Oct  2 03:12:19 UTC 2016
Thu Oct  2 03:12:24 UTC 2016
Thu Oct  2 03:12:29 UTC 2016
Thu Oct  2 03:12:34 UTC 2016
```

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

```
Thu Oct  2 03:11:34 UTC 2016
Thu Oct  2 03:11:59 UTC 2016
```

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

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

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

```
$ sudo docker ps
```

```
CONTAINER ID          IMAGE          COMMAND
CREATED               STATUS         PORTS          NAMES
c439077aa80a          ubuntu:16.04   "/bin/bash"
47 seconds ago        Up 46 seconds (Paused)
ecstatic_torvalds
```

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

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

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

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

Housekeeping containers

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

During a container startup, we can instruct the Docker Engine to clean up the container as soon as it reaches the stopped state. For this purpose, the docker run subcommand supports a --rm option (for example, sudo docker run -i -t --rm ubuntu:16.04 /bin/bash).

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

```
$ sudo docker ps -a
CONTAINER ID       IMAGE       COMMAND      CREATED
STATUS            PORTS
NAMES
7473f2568add      ubuntu:16.04  "/bin/bash"  5 seconds ago
Exited (0) 3 seconds ago
jolly_wilson

$ sudo docker rm 7473f2568add
7473f2568add
$
```

Two Docker subcommands, that is, docker rm and docker ps, can be combined together for automatically deleting all the containers that are not currently running, as shown in the following command:

```
$ sudo docker rm $(sudo docker ps -aq)
```

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

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

Perhaps we could avoid the preceding error by filtering the containers that are in the Exited state using the filter (-f) option of the docker ps subcommand, as shown here:

```
$ sudo docker rm $(sudo docker ps -aq -f state=exited)
```

eeling frustrated at typing such a long and complicated chain of commands? Here is the good news for you. The docker container prune subcommand comes in handy to remove all stopped containers. This functionality is introduced in Docker version 1.13 and here is a sample run of the docker container prune subcommand:

```
$ sudo docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
9b1aaaf108d3922d1a503fe01e9024302f0434a3b387c450d3b302020966a13e
d43c75065c6147501a7bc62f418fe501eeabaddd8617d77a4b28b5807dfeaa89
1614c44092f1c358cbb248a49430e70b674b52b32b8a193da9bba9b7136d1640

Total reclaimed space: 0 B
```

Building images from containers

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

Let's take ubuntu:16.04 as our base image, install the wget application, and then convert the running container to an image by performing the following steps:

1. Launch an ubuntu:16.04 container using the docker run subcommand, as shown here:

```
$ sudo docker run -i -t ubuntu:16.04 /bin/bash
```

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

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

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

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

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

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

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

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

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

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

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

```
sudo docker commit 472c96295678 \
    learningdocker/ubuntu_wget
sha256:a530f0a0238654fa741813fac39bba2cc14457aee079a7ae1f
e1c64dc7e1ac25
```

We committed our image using the learningdocker/ubuntu_wget name.

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

```
$ sudo docker images
REPOSITORY              TAG          IMAGE ID
CREATED                  VIRTUAL SIZE
learningdocker/ubuntu_wget  latest       a530f0a02386
48 seconds ago            221.3 MB
busybox                  latest        e72ac664f4f0
2 days ago               2.433 MB
ubuntu                   16.04        6b4e8a7373fe
2 days ago               194.8 MB
```

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

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

Launching a container as a daemon

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

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

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

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

```
$ sudo docker logs \
0137d98ee363b44f22a48246ac5d460c65b67e4d7955aab6cbb0379ac421269b
Sat Oct  4 17:41:04 UTC 2016
Sat Oct  4 17:41:09 UTC 2016
Sat Oct  4 17:41:14 UTC 2016
Sat Oct  4 17:41:19 UTC 2016
```

Building Images

In the previous chapter, we explained the image and container handling, and its housekeeping techniques and tips in detail. In addition to that, we described the standard procedure for installing a software package on a Docker container and then converting the container into an image for future usage and maneuvering. This chapter is quite different from the previous ones and is included to clearly describe how Docker images are built using Dockerfile, which is the standard way for building highly usable Docker images. Leveraging Dockerfile is the most competent way of building powerful images for the software development community.

Docker's integrated image building system

Docker images are the fundamental building blocks of containers. These images could be very basic operating environments, such as busybox or ubuntu, as we found while experimenting with Docker in earlier chapters. Alternatively, the images can craft advanced application stacks for the enterprise and cloud IT environments. As we discussed in the previous chapter, we can craft an image manually by launching a container from a base image, install all the required applications, make the necessary configuration file changes, and then commit the container as an image.

As a better alternative, we can resort to the automated approach of crafting the images using Dockerfile, which is a text-based build script that contains special instructions in a sequence for building the correct and relevant images from the base images. The sequential instructions inside Dockerfile can include selecting the base image, installing the required application, adding the configuration and the data files, and automatically running the services as well as exposing those services to the external world. Thus, the Dockerfile-based automated build system has simplified the image-building process remarkably. It also offers a great deal of flexibility in organizing the build instructions and in visualizing the complete build process.

The Docker Engine tightly integrates this build process with the help of the docker build subcommand. In the client-server paradigm of Docker, the Docker server (or daemon) is responsible for the complete build process, and the Docker command-line interface is responsible for transferring the build context, including transferring Dockerfileto the daemon.

In order to have a sneak peak into the Dockerfile integrated build system, we will introduce you to a basic Dockerfile in this section. Then, we will explain the steps for converting that Dockerfile into an image, and then launch a container from that image. Our Dockerfile is made up of two instructions, as shown here:

```
$ cat Dockerfile
FROM busybox:latest
CMD echo Hello World!!
```

We will discuss these two instructions as follows:

- The first instruction is for choosing the base image selection. In this example, we select the busybox:latest image.
- The second instruction is for carrying out the CMD command, which instructs the container to execute echo Hello World!!.

Now, let's proceed towards generating a Docker image using the preceding Dockerfile by calling docker build along with the path of Dockerfile. In our example, we will invoke the docker build subcommand from the directory where we have stored Dockerfile, and the path will be specified by the following command:

```
$ sudo docker build .
```

After issuing the preceding command, the build process will begin by sending the build context to the daemon and then display the text shown here:

```
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM busybox:latest
```

The build process will continue and after completing itself, will display the following:

```
Successfully built 0a2abe57c325
```

In the preceding example, the image was built with the 0a2abe57c325 image ID. Let's use this image to launch a container using the docker run subcommand, as follows:

```
$ sudo docker run 0a2abe57c325
Hello World!!
```

With very little effort, we have been able to craft an image with busybox as the base image, and we have been able to extend that image to produce Hello World!!. This is a simple application, but the enterprise-scale images can also be realized using the same technology.

Now, let's look at the image details using the docker images subcommand, as shown here:

```
$ sudo docker images
REPOSITORY    TAG       IMAGE ID     CREATED      SIZE
<none><none>        0a2abe57c325  2 hours ago   1.11 MB
```

Here, you may be surprised to see that the image (REPOSITORY) and TAG name have been listed as <none>. This is because we did not specify any image or any TAG name when we built this image. You could specify an image name and optionally a TAG name using the docker tag subcommand, as shown here:

```
$ sudo docker tag 0a2abe57c325 busyboxplus
```

The alternative approach is to build the image with an image name during the build time using the -t option for the docker build subcommand, as shown here:

```
$ sudo docker build -t busyboxplus .
```

Since there is no change to the instructions in Dockerfile, the Docker Engine will efficiently reuse the old image that has the 0a2abe57c325 ID and update the image name to busyboxplus. By default, the build system applies latest as the tag name. This behavior can be modified by specifying the tag name after the image name by having a : separator placed between them. This means that, <image name>:<tag name> is the correct syntax for modifying behaviors, wherein <image name> is the name of the image and <tag name> is the name of the tag.

Once again, let's look at the image details using the docker images subcommand, and you will notice that the image (repository) name is busyboxplus and the tag name is latest:

```
$ sudo docker images
REPOSITORY    TAG      IMAGE ID      CREATED      VIRTUAL SIZE
busyboxplus   latest   0a2abe57c325  2 hours ago
2.433 MB
```

A quick overview of the Dockerfile's syntax

In this section, we will explain the syntax or the format of Dockerfile. A Dockerfile is made up of instructions, comments, parser directives, and empty lines, as shown here:

```
# Comment

INSTRUCTION arguments
```

The instruction line of Dockerfile is made up of two components, where the instruction line begins with the INSTRUCTION itself, which is followed by the arguments for the instruction. The INSTRUCTION can be written in any case, in other words, it is case-insensitive. However, the standard practice or the convention is to use **uppercase** in order to differentiate it from the arguments. Let's relook at the content of Dockerfile in our previous example:

```
FROM busybox:latest
CMD echo Hello World!!
```

Here, FROM is an instruction that has taken busybox:latest as an argument and CMD is an instruction that has taken echo Hello World!! as an argument.

The comment line

The comment line in Dockerfile must begin with the # symbol. The # symbol after an instruction is considered as an argument. If the # symbol is preceded by a whitespace, then the docker build system will consider this as an unknown instruction and skip the line. Now, understand the preceding cases with the help of an example to get a better understanding of the comment line:

- A valid Dockerfile comment line always begins with a # symbol as the first character of the line:

```
# This is my first Dockerfile comment
```

- The # symbol can be part of an argument:

```
    CMD echo ### Welcome to Docker ###
```

- If the # symbol is preceded by a whitespace, then it is considered as an unknown instruction by the build system:

```
 # this is an invalid comment line
```

The docker build system ignores any empty line in the Dockerfile and hence, the author of Dockerfile is encouraged to add comments and empty lines to substantially improve the readability of Dockerfile.

The parser directives

As the name implies, the parser directives instruct the Dockerfile parser to handle the content of Dockerfileas specified in the directives. The parser directives are optional and must be at the top of a Dockerfile. Currently, escape is the only supported directive.

We use the escape character to escape characters in a line or to extend a single line to multiple lines. On UNIX-like platforms, \ is the escape character, whereas, on Windows, \ is a directory path separator and ` is the escape character. By default, the Dockerfile parser considers \ is the escape character and you could override this on Windows using the escape parser directive, as shown here:

```
# escape=`
```

The Dockerfile build instructions

So far, we have looked at the integrated build system, the Dockerfile syntax, and a sample life cycle, wherein we discussed how a sample Dockerfile is leveraged for generating an image and how a container gets spun off from that image. In this section, we will introduce the Dockerfile instructions, their syntax, and a few befitting examples.

The FROM instruction

The FROM instruction is the most important one and is the first valid instruction of a Dockerfile. It sets the base image for the build process. Subsequent instructions will use this base image and build on top of it. The Docker build system lets you flexibly use the images built by anyone. You can also extend them by adding more precise and practical features. By default, the Docker build system looks for the images in the Docker host. However, if the image is not found in the Docker host, then the Docker build system will pull the image from the publicly available Docker Hub Registry. The Docker build system will return an error if it cannot find the specified image in the Docker host and the Docker Hub Registry.

The FROM instruction has the following syntax:

```
FROM <image>[:<tag>|@<digest>]
```

In the preceding code statement, note the following:

- <image>: This is the name of the image that will be used as the base image.
- <tag> or <digest>: Both <tag> and <digest> are optional attributes and you can qualify a particular Docker image version using either a tag attribute or a digest attribute. The latest tag is assumed by default if both tag and digest are not present.

Here is an example of the FROM instruction with the centos image name:

```
FROM centos
```

In the preceding example, the Docker build system will implicitly default to the latest tag because neither a tag nor a digest is explicitly added to the image name. Here is another example of the FROM instruction with the ubuntu image name and the 16.04 tag qualifier:

```
FROM ubuntu:16.04
```

Next is a classic example of the FROM instruction with the ubuntu image name and the digest qualifier:

```
FROM
ubuntu@sha256:8e2324f2288c26e1393b63e680ee7844202391414dbd48497e9a4fd997cd3cbf
```

Docker allows multiple FROM instructions in a single Dockerfile in order to create multiple images. The Docker build system will pull all the images specified in the FROM instruction. Docker does not provide any mechanism for naming the individual images that are generated with the help of multiple FROM instructions. We strongly discourage using multiple FROM instructions in a single Dockerfile, as damaging conflicts could arise.

The MAINTAINER instruction

The MAINTAINER instruction is an informational instruction of a Dockerfile. This instruction capability enables the authors to set the details in an image. Docker does not place any restrictions on placing the MAINTAINERinstruction in Dockerfile. However, it is strongly recommended that you place it after the FROM instruction.

The following is the syntax of the MAINTAINER instruction, where <author's detail> can be in any text. However, it is strongly recommended that you use the image, author's name, and e-mail address as shown in this code syntax:

```
MAINTAINER <author's detail>
```

Here is an example of the MAINTAINER instruction with the author's name and e-mail address:

```
MAINTAINER QT <qtdevops@gmail.com>
```

The COPY instruction

The COPY instruction enables you to copy the files from the Docker host to the filesystem of the new image. The following is the syntax of the COPY instruction:

```
COPY <src> ... <dst>
```

The preceding code terms are explained here:

- <src>: This is the source directory, the file in the build context, or the directory from where the docker buildsubcommand was invoked.
- ...: This indicates that multiple source files can either be specified directly or be specified by wildcards.
- <dst>: This is the destination path for the new image into which the source file or directory will get copied. If multiple files have been specified, then the destination path must be a directory and it must end with a slash (/).

Using an absolute path for the destination directory or a file is recommended. In the absence of an absolute path, the COPY instruction will assume that the destination path will start from the root (/). The COPY instruction is powerful enough for creating a new directory and for overwriting the filesystem in the newly created image.

In the following example, we will copy the html directory from the source build context to /var/www/html, which is in the image filesystem, using the COPY instruction, as shown here:

```
COPY html /var/www/html
```

Here is another example of the multiple files (httpd.conf and magic) that will be copied from the source build context to /etc/httpd/conf/, which is in the image filesystem:

```
COPY httpd.conf magic /etc/httpd/conf/
```

The ADD instruction

The ADD instruction is similar to the COPY instruction. However, in addition to the functionality supported by the COPY instruction, the ADD instruction can handle the TAR files and remote URLs. We can annotate the ADD instruction as COPY on steroids.

The following is the syntax of the ADD instruction:

```
ADD <src> ... <dst>
```

The arguments of the ADD instruction are very similar to those of the COPY instruction, as shown here:

- <src>: This is either the source directory or the file that is in the build context or in the directory from where the docker build subcommand will be invoked. However, the noteworthy difference is that the source can either be a TAR file stored in the build context or be a remote URL.
- ...: This indicates that multiple source files can either be specified directly or be specified using wildcards.
- <dst>: This is the destination path for the new image into which the source file or directory will be copied.

Here is an example for demonstrating the procedure for copying multiple source files to the various destination directories in the target image filesystem. In this example, we have taken a

TAR file (web-page-config.tar) in the source build context with the http daemon configuration file and the files for the web pages are stored in the appropriate directory structure, as shown here:

```
$ tar tf web-page-config.tar
etc/httpd/conf/httpd.conf
var/www/html/index.html
var/www/html/aboutus.html
var/www/html/images/welcome.gif
var/www/html/images/banner.gif
```

The next line in the Dockerfile content has an ADD instruction for copying the TAR file (web-page-config.tar) to the target image and extracting the TAR file from the root directory (/) of the target image, as shown here:

```
ADD web-page-config.tar /
```

Thus, the TAR option of the ADD instruction can be used for copying multiple files to the target image.

The ENV instruction

The ENV instruction sets an environment variable in the new image. An environment variable is a key-value pair, which can be accessed by any script or application. Linux applications use the environment variables a lot for a starting configuration.

The following line forms the syntax of the ENV instruction:

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

Here, the code terms indicate the following:

- <key>: This is the environment variable
- <value>: This is the value that is to be set for the environment variable

The following lines give two examples for the ENV instruction, where, in the first line, DEBUG_LVL has been set to 3 and on the second line, APACHE_LOG_DIR has been set to /var/log/apache:

```
ENV DEBUG_LVL 3
ENV APACHE_LOG_DIR /var/log/apache
```

The ARG instruction

The ARG instruction lets you define variables that can be passed during the Docker image build time. The Docker build subcommand supports the --build-arg flag to pass a value to the variables defined using the ARGinstruction. If you specify a build argument that was not defined in your Dockerfile, the build would fail. In other words, the build argument variables must be defined in the Dockerfile to be passed during the Docker image build time.

The syntax of the ARG instruction is as follows:

```
ARG <variable>[=<default value>]
```

Here, the code terms mean the following:

- <variable>: This is the build argument variable
- <default value>: This is the default value you could optionally specify to the build argument variable

Here is an example for the ARG instruction:

```
ARG usr
ARG uid=1000
```

Here is an example of the --build-arg flag of the docker build subcommand:

```
docker build --build-arg usr=app --build-arg uid=100 .
```

The environment variables

The environment variables declared using the ENV or ARG instruction can be used in the ADD, COPY, ENV, EXPOSE, LABEL, USER, WORKDIR, VOLUME, STOPSIGNAL, and ONBUILD instructions.

Here is an example of the environment variable usage:

```
ARG BUILD_VERSION
LABEL com.example.app.build_version=${BUILD_VERSION}
```

The USER instruction

The USER instruction sets the startup user ID or username in the new image. By default, the containers will be launched with root as the user ID or UID. Essentially, the USER instruction will modify the default user ID from root to the one specified in this instruction.

The syntax of the USER instruction is as follows:

```
USER <UID>|<UName>
```

The USER instructions accept either <UID> or <UName> as its argument:

- <UID>: This is a numerical user ID
- <UName>: This is a valid username

The following is an example for setting the default user ID at the time of startup to 73. Here, 73 is the numerical ID of the user:

```
USER 73
```

Though it is recommended that you have a valid user ID to match with the /etc/passwd file, the user ID can contain any random numerical value. However, the username must match with a valid username in the /etc/passwd file, otherwise, the docker run subcommand will fail and it will display the following error message:

```
finalize namespace setup user get supplementary groups Unable to find user
```

The WORKDIR instruction

The WORKDIR instruction changes the current working directory from / to the path specified by this instruction. The ensuing instructions, such as RUN, CMD, and ENTRYPOINT will also work on the directory set by the WORKDIR instruction.

The following line gives the appropriate syntax for the WORKDIR instruction:

```
WORKDIR <dirpath>
```

Here, <dirpath> is the path for the working directory to set in. The path can be either absolute or relative. In the case of a relative path, it will be relative to the previous path set by the WORKDIR instruction. If the specified directory is not found in the target image filesystem, then the directory will be created.

The following line is a clear example of the WORKDIR instruction in a Dockerfile:

```
WORKDIR /var/log
```

The VOLUME instruction

The VOLUME instruction creates a directory in the image filesystem, which can later be used for mounting volumes from the Docker host or the other containers.

The VOLUME instruction has two types of syntax, as shown here:

- The first type is either exec or JSON array (all values must be within double-quotes ("")):

```
VOLUME ["<mountpoint>"]
```

- The second type is the shell, as shown here:

```
VOLUME <mountpoint>
```

In the preceding lines, <mountpoint> is the mount point that has to be created in the new image.

The EXPOSE instruction

The EXPOSE instruction opens up a container network port for communicating between the container and the external world.

The syntax of the EXPOSE instruction is as follows:

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

Here, the code terms mean the following:

- <port>: This is the network port that has to be exposed to the outside world.
- <proto>: This is an optional field provided for a specific transport protocol, such as TCP and UDP. If no transport protocol has been specified, then TCP is assumed to be the transport protocol.

The EXPOSE instruction allows you to specify multiple ports in a single line.

The following is an example of the EXPOSE instruction inside a Dockerfile exposing the 7373 port number as a UDP port and the 8080 port number as a TCP port. As mentioned earlier, if the transport protocol has not been specified, then the TCP transport is assumed to be the transport protocol:

```
EXPOSE 7373/udp 8080
```

The LABEL instruction

The LABEL instruction enables you to add key-value pairs as metadata to your Docker images. These metadata can be further leveraged to provide meaningful Docker image management and orchestration.

The syntax of the LABEL instruction is as follows:

```
LABEL <key-1>=<val-1><key-2>=<val-2> ... <key-n>=<val-n>
```

The LABEL instruction can have one or more key-value pairs. Though a Dockerfile can have more than one LABEL instruction, it is recommended that you use a single LABEL instruction with multiple key-value pairs.

Here is an example of the LABEL instruction:

```
LABEL version="2.0"
    release-date="2016-08-05"
```

The preceding label keys are very simple and this could result in naming conflicts. Hence Docker recommends using namespaces to label keys using the reverse domain notation. There is a community project called **Label Schema** that provides shared namespace. The shared namespace acts as a glue between the image creators and tool builders to provide standardized Docker image management and orchestration. Here is an example of the LABEL instruction using Label Schema:

```
LABEL org.label-schema.schema-version="1.0"
    org.label-schema.version="2.0"
    org.label-schema.description="Learning Docker Example"
```

The RUN instruction

The RUN instruction is the real workhorse during the build, and it can run any command. The general recommendation is to execute the multiple commands using one RUN instruction. This reduces the layers in the resulting Docker image because the Docker system inherently creates a layer for each time an instruction is called in Dockerfile.

The RUN instruction has two types of syntax:

- The first is the shell type, as shown here:

```
RUN <command>
```

Here, <command> is the shell command that has to be executed during the build time. If this type of syntax is to be used, then the command is always executed using /bin/sh -c.

- The second syntax type is either exec or the JSON array, as shown here:

```
RUN ["<exec>", "<arg-1>", ..., "<arg-n>"]
```

Here, the code terms mean the following:

- <exec>: This is the executable to run during the build time
- <arg-1>, ..., <arg-n>: These are the variable numbers (zero or more) of arguments for the executable

Unlike the first type of syntax, this type does not invoke /bin/sh -c. Hence, the types of shell processing, such as the variable substitution ($USER) and the wildcard substitution (*, ?) do not happen in this type. If shell processing is critical for you, then you are encouraged to use the shell type. However, if you still prefer the exec (JSON array type) type, then use your preferred shell as the executable and supply the command as an argument.

Consider the example, RUN ["bash", "-c", "rm", "-rf", "/tmp/abc"].

Now, let's look at a few examples of the RUN instruction. In the first example, we will use the RUN instruction for adding a greeting line to the .bashrc file in the target image filesystem, as shown here:

```
RUN echo "echo Welcome to Docker!" >> /root/.bashrc
```

The second example is a Dockerfile, which has the instructions for crafting an Apache2 application image on top of the Ubuntu 14.04 base image. The following steps will explain the Dockerfile instructions line by line:

1. We are going to build an image using ubuntu:14.04 as the base image, using the FROM instruction, as shown here:

```
###############################################
# Dockerfile to build an Apache2 image
###############################################
# Base image is Ubuntu
FROM ubuntu:14.04
```

2. Set the author's details using the MAINTAINER instruction, as shown here:

```
# Author: Dr. Peter
MAINTAINER Dr. Peter <peterindia@gmail.com>
```

3. Using one RUN instruction, we will synchronize the apt repository source list, install the apache2 package, and then clean the retrieved files, as shown here:

```
# Install apache2 package
RUN apt-get update && \
  apt-get install -y apache2 && \
  apt-get clean
```

The CMD instruction

The CMD instruction can run any command (or application), which is similar to the RUN instruction. However, the major difference between these two is the time of execution. The command supplied through the RUN instruction is executed during the build time, whereas the command specified by the CMD instruction is executed when the container is launched from the newly created image. Thus, the CMD instruction provides a default execution for this container. However, it can be overridden by the docker run subcommand arguments. When the application terminates, the container will also terminate along with the application and vice versa.

The CMD instruction has three types of syntax, as shown here:

- The first syntax type is the shell type, as shown here:

```
CMD <command>
```

Here, <command> is the shell command, which has to be executed during the launch of the container. If this type of syntax is used, then the command is always executed using /bin/sh -c.

- The second type of syntax is exec or the JSON array, as shown here:

```
CMD ["<exec>", "<arg-1>", ..., "<arg-n>"]
```

Here, the code terms mean the following:

- <exec>: This is the executable, which is to be run during the launch of the container
- <arg-1>, ..., <arg-n>: These are the variable numbers (zero or more) of arguments for the executable

- The third type of syntax is also exec or the JSON array, which is similar to the previous type. However, this type is used for setting the default parameters to the ENTRYPOINT instruction, as shown here:

```
CMD ["<arg-1>", ..., "<arg-n>"]
```

Here, the code terms mean the following:

<arg-1>, ..., <arg-n>: These are the variable numbers (zero or more) of arguments for the ENTRYPOINTinstruction, which will be explained in the next section.

Syntactically, you can add more than one CMD instruction in Dockerfile. However, the build system will ignore all the CMD instructions except for the last one. In other words, in the case of multiple CMD instructions, only the last CMD instruction will be effective.

Here, in this example, let's craft an image using Dockerfile with the CMD instruction for providing a default execution and then launching a container using the crafted image. The following is Dockerfile with a CMDinstruction to echo a text:

```
########################################################
# Dockerfile to demonstrate the behavior of CMD
########################################################
# Build from base image busybox:latest
FROM busybox:latest
# Author: Dr. Peter
MAINTAINER Dr. Peter <peterindia@gmail.com>
# Set command for CMD
CMD ["echo", "Dockerfile CMD demo"]
```

Now, let's build a Docker image using the docker build subcommand and cmd-demo as the image name. The docker build system will read the instruction from the Dockerfile that is stored in the current directory (.), and craft the image accordingly, as shown here:

```
$ sudo docker build -t cmd-demo .
```

Having built the image, we can launch the container using the docker run subcommand, as shown here:

```
$ sudo docker run cmd-demo
Dockerfile CMD demo
```

We have given a default execution for our container and our container has faithfully echoed Dockerfile CMD demo. However, this default execution can be easily overridden by passing another command as an argument to the docker run subcommand, as shown in the following example:

```
$ sudo docker run cmd-demo echo Override CMD demo
Override CMD demo
```

The ENTRYPOINT instruction

The ENTRYPOINT instruction will help in crafting an image for running an application (entry point) during the complete life cycle of the container, which would have been spun out of the image. When the entry point application is terminated, the container would also be terminated along with the application and vice versa. Thus, the ENTRYPOINT instruction would make the container function like an executable. Functionally, ENTRYPOINT is akin to the CMD instruction, but the major difference between the two is that the entry point application is launched using the ENTRYPOINT instruction, which cannot be overridden using the docker run subcommand arguments. However, these docker run subcommand arguments will be passed as additional arguments to the entry point application. Having said this, Docker provides a mechanism for overriding the entry point application through the --entrypoint option in the docker run subcommand. The --entrypoint option can accept only words as its argument and hence, it has limited functionality.

Syntactically, the ENTRYPOINT instruction is very similar to the RUN and CMD instructions, and it has two types of syntax, as shown here:

- The first type of syntax is the shell type, as shown here:

```
ENTRYPOINT <command>
```

Here, <command> is the shell command, which is executed during the launch of the container. If this type of syntax is used, then the command is always executed using /bin/sh -c.

- The second type of syntax is exec or the JSON array, as shown here:

```
ENTRYPOINT ["<exec>", "<arg-1>", ..., "<arg-n>"]
```

Here, the code terms mean the following:

- <exec>: This is the executable, which has to be run during the launch of the container
- <arg-1>, ..., <arg-n>: These are the variable numbers (zero or more) of arguments for the executable

Syntactically, you can have more than one ENTRYPOINT instruction in a Dockerfile. However, the build system will ignore all the ENTRYPOINT instructions except the last one. In other words, in the case of multiple ENTRYPOINT instructions, only the last ENTRYPOINT instruction will be effective.

In order to gain a better understanding of the ENTRYPOINT instruction, let's craft an image using Dockerfilewith the ENTRYPOINT instruction and then launch a container using the crafted image. The following is Dockerfile with an ENTRYPOINT instruction to echo a text:

```
###########################################################
# Dockerfile to demonstrate the behavior of ENTRYPOINT
###########################################################
# Build from base image busybox:latest
FROM busybox:latest
# Author: Dr. Peter
MAINTAINER Dr. Peter <peterindia@gmail.com>
# Set entrypoint command
ENTRYPOINT ["echo", "Dockerfile ENTRYPOINT demo"]
```

Now, let's build a Docker image using the docker build as the subcommand and entrypoint-demo as the image name. The docker build system would read the instruction from Dockerfile stored in the current directory (.) and craft the image, as shown here:

```
$ sudo docker build -t entrypoint-demo .
```

Having built the image, we can launch the container using the docker run subcommand:

```
$ sudo docker run entrypoint-demo
Dockerfile ENTRYPOINT demo
```

Here, the container will run like an executable by echoing the Dockerfile ENTRYPOINT demo string and then it will exit immediately. If we pass any additional arguments to the docker run subcommand, then the additional argument would be passed to the ENTRYPOINT command. The following is the demonstration of launching the same image with the additional arguments given to the docker run subcommand:

```
$ sudo docker run entrypoint-demo with additional arguments
Dockerfile ENTRYPOINT demo with additional arguments
```

Now, let's see an example where we override the build time entry point application with the --entrypoint option and then launch a shell (/bin/sh) in the docker run subcommand, as shown here:

```
$ sudo docker run -it --entrypoint="/bin/sh" entrypoint-demo
/ #
```

The HEALTHCHECK instruction

Any Docker container is designed to run just one process/application/service as a best practice and also to be uniquely compatible with the fast-evolving **Microservices Architecture** (**MSA**). The life cycle of a container is tightly bound to the process running inside the container. When the process running inside the container crashes or dies for any reason, the Docker Engine will move the container to the stop state. There is a possibility that the application running inside the container might be in an unhealthy state and such a state must be externalized for effective container management. Here the HEALTHCHECK instruction comes in handy to monitor the health of the containerized application by running a health monitoring command (or tool) at a prescribed interval.

The syntax of the HEALTHCHECK instruction is as follows:

```
HEALTHCHECK [<options>] CMD <command>
```

Here, the code terms mean the following:

- <command>: The HEALTHCHECK command is to be executed at a prescribed interval. If the command exit status is 0, the container is considered to be in the healthy state. If the command exit status is 1, the container is considered to be in the unhealthy state.
- <options>: By default, the HEALTHCHECK command is invoked every 30 seconds, the command timeout is 30 seconds, and the command is retried three times before the container is declared unhealthy. Optionally, you can modify the default interval, timeout, and retries values using the following options:
  - --interval=<DURATION> [default: 30s]
  - --timeout=<DURATION> [default: 30s]
  - --retries=<N> [default: 3]

Here is an example of the HEALTHCHECK instruction:

```
HEALTHCHECK --interval=5m --timeout=3s
  CMD curl -f http://localhost/ || exit 1
```

If there is more than one HEALTHCHECK instruction in a Dockerfile, only the last HEALTHCHECK instruction will take effect. So you can override the health check defined in the base image. For any reason, if you choose to disable the health check defined in the base image, you could resort to the NONE option of the HEALTHCHECKinstructions, as shown here:

```
HEALTHCHECK NONE
```

The ONBUILD instruction

The ONBUILD instruction registers a build instruction to an image and this gets triggered when another image is built using this image as its base image. Any build instruction can be registered as a trigger and those instructions will be triggered immediately after the FROM instruction in the downstream Dockerfile. Thus, the ONBUILDinstruction can be used for deferring the execution of the build instruction from the base image to the target image.

The syntax of the ONBUILD instruction is as follows:

```
ONBUILD <INSTRUCTION>
```

Here, <INSTRUCTION> is another Dockerfile build instruction, which will be triggered later. The ONBUILDinstruction does not allow the chaining of another ONBUILD instruction. In addition, it does not allow the FROMand MAINTAINER instruction as an ONBUILD trigger.

Here is an example of the ONBUILD instruction:

```
ONBUILD ADD config /etc/appconfig
```

The STOPSIGNAL instruction

The STOPSIGNAL instruction enables you to configure an exit signal for your container. It has the following syntax:

```
STOPSIGNAL <signal>
```

Here, <signal> is either a valid signal name, such as SIGKILL, or a valid unsigned signal number.

The SHELL instruction

The SHELL instruction allows us to override the default shell, that is, sh on Linux and cmd on Windows.

The syntax of the SHELL instruction is as follows:

```
SHELL ["<shell>", "<arg-1>", ..., "<arg-n>"]
```

Here, the code terms mean the following:

- <shell>: The shell to be used during container runtime

- <arg-1>, ..., <arg-n>: These are the variable numbers (zero or more) of the arguments for the shell

The .dockerignore file

In the **Docker's integrated image building system** section, you learned that the docker build process will send the complete build context to the daemon. In a practical environment, the docker build context will contain many other working files and directories, which would never be built into the image. Nevertheless, the docker buildsystem will still send those files to the daemon. So, you may be wondering how you can optimize the build process by not sending these working files to the daemon. Well, the folks behind Docker too have thought about that and have given a very simple solution, using a .dockerignore file.

The .dockerignore file is a newline-separated TEXT file, wherein you can provide the files and the directories which are to be excluded from the build process. The exclusion list in the file can have both the fully specified file/directory name and the wildcards.

The following snippet is a sample .dockerignore file through which the build system has been instructed to exclude the .git directory and all the files that have the .tmp extension:

```
.git
*.tmp
```

A brief on the Docker image management

As we saw in the previous chapter and earlier in this chapter, there are many ways of getting a handle on a Docker image. You could download a full setup application stack from the public repository using the docker pullsubcommand. Otherwise, you could craft your own application stack either manually using the docker commitsubcommand or automatically using Dockerfile and the docker build subcommand combination.

The Docker images are positioned as the key building blocks of the containerized applications that in turn enable the realization of distributed applications, which will be deployed on the cloud servers. The Docker images are built in layers, that is, the images can be built on top of other images. The original image is called the **parent image** and the one that is generated is called the **child image**. The base image is a bundle, which comprises an application's common dependencies. Each change that is made to the original image is stored as a separate layer. Each time you commit to a Docker image, you will create a new layer on the Docker image and each change that is made to the original image will be stored as a separate layer. As the reusability of the layers is facilitated, making new Docker images becomes simple and fast. You can create a new Docker image by changing a single line in Dockerfile and you do not need to rebuild the whole stack.

Now that you learned about layers in the Docker image, you may be wondering how one could visualize these layers in a Docker image. Well, the docker history subcommand is an excellent and handy tool for visualizing the image layers.

Here, let's see a practical example for understanding layering in the Docker images better. For this purpose, let's follow these steps:

1. Here, we have Dockerfile with the instructions for automatically building the Apache2 application image on top of the Ubuntu 14.04 base image. The RUN section of the previously crafted and used Dockerfile of this chapter will be reused in this section, as shown here:

```
###########################################
# Dockerfile to build an Apache2 image
###########################################
# Base image is Ubuntu
FROM ubuntu:14.04
# Author: Dr. Peter
MAINTAINER Dr. Peter <peterindia@gmail.com>
# Install apache2 package
RUN apt-get update &&
   apt-get install -y apache2 &&
   apt-get clean
```

2. Now, craft an image from the preceding Dockerfile using the docker build subcommand, as shown here:

```
$ sudo docker build -t apache2 .
```

3. Finally, let's visualize the layers in the Docker image using the docker history subcommand:

```
$  sudo docker history apache2
```

The preceding subcommand will produce a detailed report on each layer of apache2 Docker image, as shown here:

```
IMAGE        CREATED      CREATED BY     SIZE
      aa83b67feeba   2 minutes ago    /bin/sh -c apt-get
      update &&  apt-get inst        35.19 MB
      c7877665c770    3 minutes ago    /bin/sh -c #(nop)
      MAINTAINER Dr. Peter <peter     0 B
      9cbaf023786c    6 days ago     /bin/sh -c #(nop)
      CMD [/bin/bash]                0 B
      03db2b23cf03    6 days ago     /bin/sh -c apt-get
      update && apt-get dist-upg      0 B
      8f321fc43180    6 days ago    /bin/sh -c sed -i
      's/^#s*(deb.*universe)$/     1.895 kB
      6a459d727ebb    6 days ago     /bin/sh -c rm -rf
      /var/lib/apt/lists/*           0 B
      2dcbbf65536c    6 days ago     /bin/sh -c echo
      '#!/bin/sh' >  /usr/sbin/polic  194.5 kB
      97fd97495e49    6 days ago     /bin/sh -c #(nop)
      ADD file:84c5e0e741a0235ef8     192.6 MB
      511136ea3c5a    16 months ago   0 B
```

Here, the apache2 image is made up of ten image layers. The top two layers, that is, the layers with the aa83b67feeba and c7877665c770 image IDs are the result of the RUN and MAINTAINER instructions in our Dockerfile. The remaining eight layers of the image will be pulled from the repository by the FROMinstruction in our Dockerfile.

An undisputable truth is that a set of best practices always plays an indispensable role in elevating any new technology. There is a well-written section listing all the best practices for crafting a Dockerfile. We found it incredible and hence, we wanted to share them for your benefit. You can find them at https://docs.docker.com/articles/dockerfile_best-practices/.

Publishing Images

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

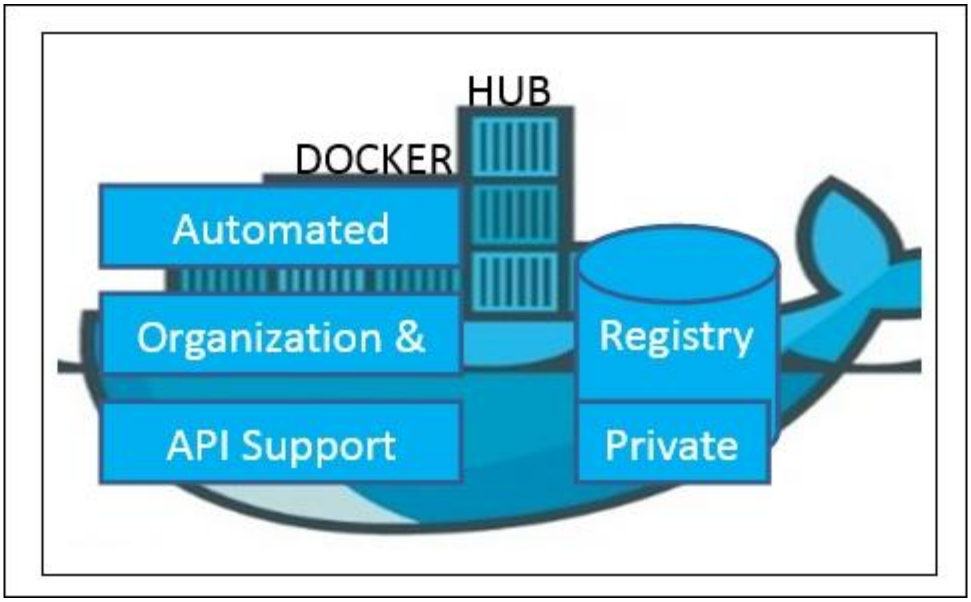
The following topics are covered in this chapter:

- Understanding Docker Hub
- Pushing images to Docker Hub
- Automatic building of images
- Private repositories on Docker Hub
- Creating organizations on Docker Hub

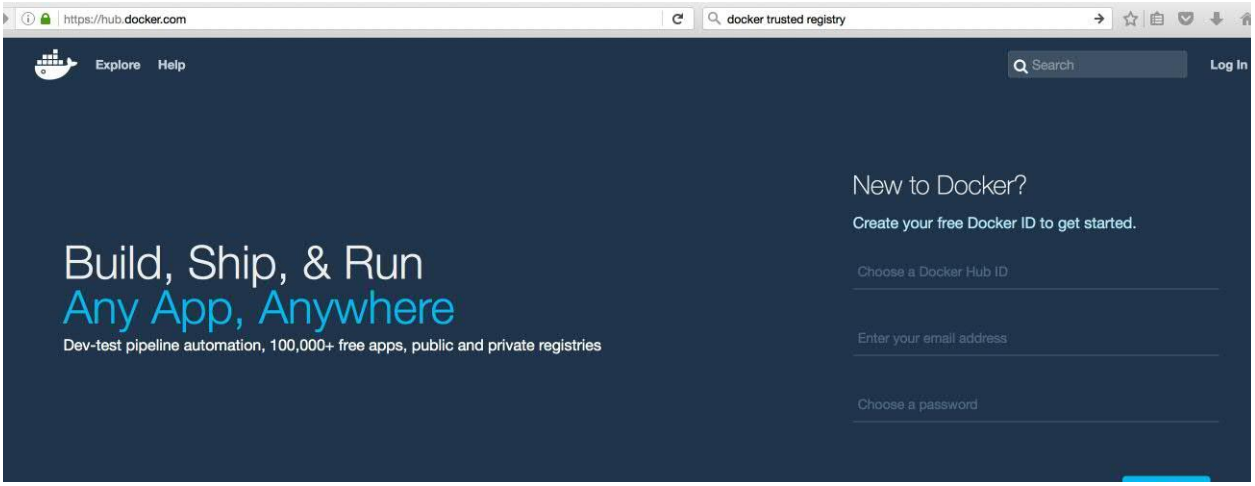
Understanding Docker Hub

Docker Hub is the central place used for keeping the Docker images either in a public or private repository. Docker Hub provides features, such as a repository for Docker images, user authentications, automated image builds, integration with GitHub or Bitbucket, and managing organizations and groups. The Docker Registry component of Docker Hub manages the repository for Docker images. Also, you can protect your repositories using Docker Security Scanning, which is free as of now. This feature was first enabled in IBM container repositories.

Docker Registry is a storage system used to store images. Automated build is a feature of Docker Hub, which is not open source yet at the time of writing this book. The following diagram shows the typical features:



In order to work with Docker Hub, you have to register with Docker Hub, and create an account using the link available at https://hub.docker.com/. You can update the Docker Hub ID, e-mail address, and password fields, as shown in the following screenshot:



After completing the sign up process, you need to complete the verification received in an e-mail.

Docker Hub also supports command-line access to Docker Hub using an Ubuntu Terminal:

$ sudo docker login

Log in with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one. Enter your username and password in the Terminal:

Username: <yourusername>
Password:

After a successful login, the output is as follows:

Login Succeeded

Pushing images to Docker Hub

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

1. Create a Docker image on the local machine by doing one of the following:

- Using the docker commit subcommand
- Using the docker commit subcommand with Dockerfile

2. Pushing this created image to Docker Hub
3. Deleting the image from Docker Hub

We will use the ubuntu base image, run the container, add a new directory and a new file, and then create a new image

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

```
$ sudo docker run -i --name="containerforhub" -t ubuntu /bin/bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
952132ac251a: Pull complete
Digest: sha256:f4691c96e6bbaa99d99ebafd9af1b68ace2aa2128ae95a60369c506dd6e6f6ab
Status: Downloaded newer image for ubuntu:latest
root@1068a1fae7da:/#
```

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

```
root@1068a1fae7da:/# mkdir mynewdir
root@1068a1fae7da:/# cd mynewdir
root@1068a1fae7da:/mynewdir# echo 'this is my new container to make image and then push to hub' > mynewfile
root@1068a1fae7da:/mynewdir# cat mynewfile
this is my new container to make image and then push to hub
root@1068a1fae7da:/mynewdir#
```

Let's build the new image with the docker commit command from the container, which has just been created.

```
$ sudo docker commit -m="NewImage for second edition" containerforhub qthought/imageforhub2
sha256:619a25519578b0525b4c098e3d349288de35986c1f3510958b6246fa5d3a3f56
```

Now, we have a new Docker image available on the local machine with the qthought/imageforhub2 name. At this point, a new image with mynewdir and mynewfile is created locally:

```
$ sudo docker images -a
REPOSITORY           TAG        IMAGE ID        CREATED             SIZE
qthought/imageforhub2   latest      619a25519578
2 minutes ago          126.6 MB
```

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

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

```
$ sudo docker push qthought/imageforhub2
The push refers to a repository [docker.io/qthought/imageforhub2]
0ed7a0595d8a: Pushed
0cad5e07ba33: Mounted from library/ubuntu
48373480614b: Mounted from library/ubuntu
latest:                                                              digest:
sha256:cd5a86d1b26ad156b0c74b0b7de449ddb1eb51db7e8ae9274307d27f810280c9   size:
1564
```

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

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

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

We will also delete the `qthought/imageforhub2` image:

```
$ sudo docker rmi qthought/imageforhub2
Untagged: qthought/imageforhub2:latest
Untagged:
vinoddandy/imageforhub2@sha256:cd5a86d1b26ad156b0c74b0b7de449ddb1eb51db7e8a
e9274307d27f810280c9
Deleted:
sha256:619a25519578b0525b4c098e3d349288de35986c1f3510958b6246fa5d3a3f56
```

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

```
$ sudo docker run -i --name="newcontainerforhub" -t \ qthought/imageforhub2 /bin/bash
Unable to find image 'qthought/imageforhub2:latest' locally
latest: Pulling from qthought/imageforhub2

952132ac251a: Already exists
82659f8f1b76: Already exists
Digest:
sha256:cd5a86d1b26ad156b0c74b0b7de449ddb1eb51db7e8ae9274307d27f810280c9
Status: Downloaded newer image for qthought /imageforhub2:latest

root@9dc6df728ae9:/# cat /mynewdir/mynewfile
this is my new container to make image and then push to hub
root@9dc6df728ae9::/#
```

We'll again create this image, but now using the Dockerfile process. So, let's create the Docker image using the Dockerfile

The Dockerfile on the local machine is as follows:

```
###############################################
# Dockerfile to build a new image
###############################################
# Base image is Ubuntu
FROM ubuntu:16.04
# Author: Dr. Peter
MAINTAINER Dr. Peter <peterindia@gmail.com>
# create 'mynewdir' and 'mynewfile'
RUN mkdir mynewdir
RUN touch /mynewdir/mynewfile
# Write the message in file
  RUN echo 'this is my new container to make image and then push to hub'
>/mynewdir/mynewfile
```

Now we'll build the image locally using the following command:

```
$ sudo docker build -t="qthought/dockerfileimageforhub1" .
Sending build context to Docker daemon 16.74 MB
Step 1 : FROM ubuntu:16.04
16.04: Pulling from library/ubuntu
862a3e9af0ae: Pull complete
```

```
7a1f7116d1e3: Pull complete
Digest: sha256:5b5d48912298181c3c80086e7d3982029b288678fccabf2265899199c24d7f89
Status: Downloaded newer image for ubuntu:16.04
---> 4a725d3b3b1c
Step 2 : MAINTAINER Dr. Peter <peterindia@gmail.com>
---> Running in 5be5edc9b970
---> 348692986c9b
Removing intermediate container 5be5edc9b970
Step 3 : RUN mkdir mynewdir
---> Running in ac2fc73d75f3
---> 21585ffffab5
Removing intermediate container ac2fc73d75f3
Step 4 : RUN touch /mynewdir/mynewfile
---> Running in c64c98954dd3
---> a6304b678ea0
Removing intermediate container c64c98954dd3
Step 5 : RUN echo 'this is my new container to make image and then push to hub' >
/mynewdir/mynewfile
---> Running in 7f6d087e29fa
---> 061944a9ba54
Removing intermediate container 7f6d087e29fa
Successfully built 061944a9ba54
```

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

```
$ sudo docker run -i --name="dockerfilecontainerforhub" -t
qthought/dockerfileimageforhub1 /bin/bash

root@236bfb39fd48:/# cat /mynewdir/mynewfile
this is my new container to make image and then push to hub
```

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

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

Automating the build process for images

---

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

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

1. We first connect Docker Hub to our GitHub account.

2. Log in to Docker Hub from https://hub.docker.com/login/, click on **Create**, and then navigate to **Create Automated Build**

4. We'll now select **Link Accounts** and Once GitHub is selected, we will select **Public and Private (Recommended)**

3. After clicking on **Select**, your GitHub repository will now be shown.
4. Now, provide the GitHub credentials to link your GitHub account with Docker Hub and select **Sign in**:
5. So, whenever the Dockerfile is updated in GitHub, the automated build gets triggered and a new image will be stored in the Docker Hub Registry. We can always check the build history. We can change the Dockerfile on the local machine and push it to GitHub

Private repositories on Docker Hub

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

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

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

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

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

In the automated build section, we built the qthought/dockerfileimageforhub1 image. Let's tag the 224affbf9a65 image ID to our locally created registry image. This tagging of the image is needed for unique identification inside the local repository. This image registry may have multiple variants in the repository, so this tag will help you identify the particular image:

```
$ sudo docker tag 224affbf9a65 \ localhost:5000/qthought/dockerfileimageforhub1
```

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

```
$ sudo docker push localhost:5000/qthought/dockerfile
imageforhub1
The push refers to a repository  [localhost:5000/qthought/dockerfileimageforhub1
] (len: 1)
Sending image list
Pushing repository localhost:5000/qthought/dockerfileimageforhub1 (1  tags)
511136ea3c5a: Image successfully pushed
d497ad3926c8: Image successfully pushed
---------------------------------------------------
224affbf9a65: Image successfully pushed
Pushing tag for rev [224affbf9a65] on
{http://localhost:5000/v1/repositories/qthought/dockerfileimageforhub1/tags/latest}
```

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

Organizations and teams on Docker Hub

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

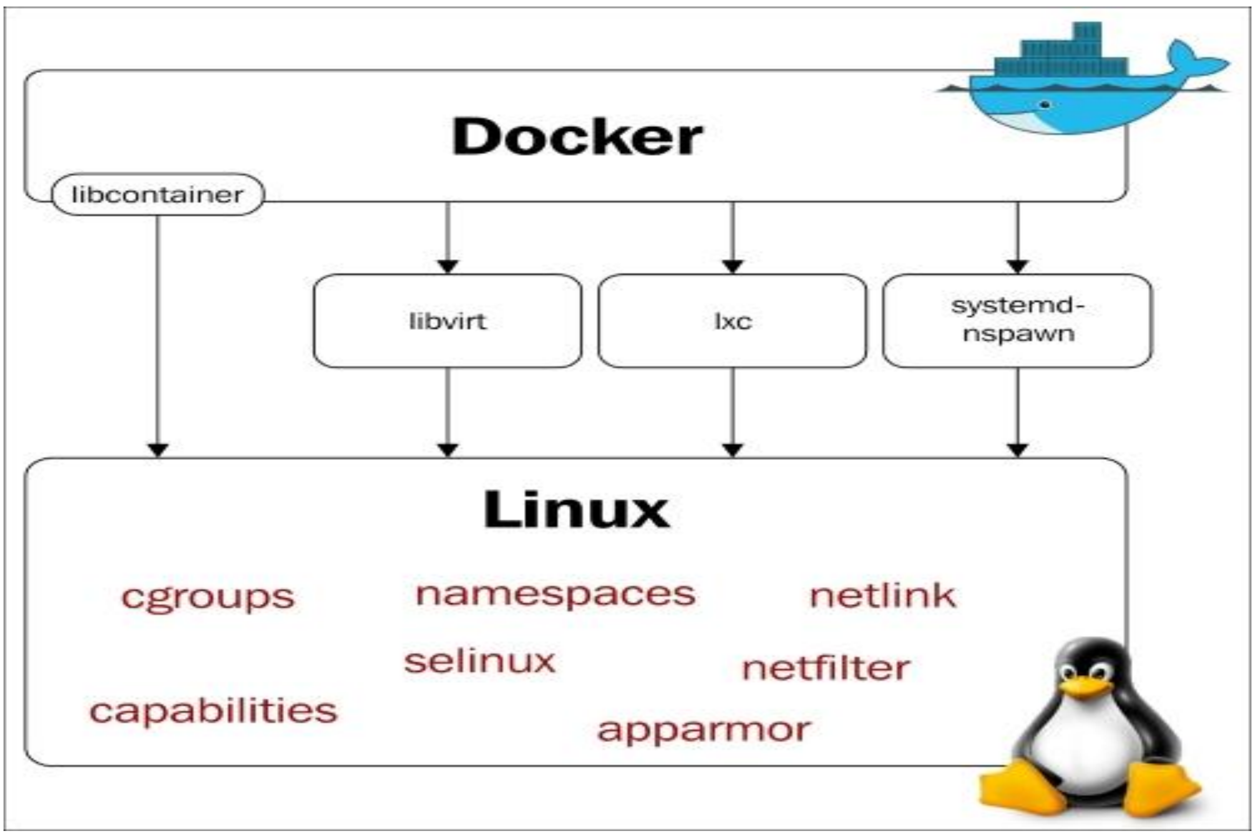
The first step is to create an organization on Docker Hub at https://hub.docker.com/organizations/add/

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

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

How Docker Works ?

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:



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 mntnamespace 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 lscgroupcommand 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
```

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.

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 groupadd 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
```

Docker Networking Primer

In this chapter, you will learn about the essential components of Docker networking and how to build and run simple container examples.

This chapter covers the following topics:

- Networking and Docker
- The docker0 bridge networking
- Docker OVS networking
- Unix domain networks
- Linking Docker containers
- What's new in Docker networking

A brief overview of container networking

Networking is a critical infrastructure component of enterprise and cloud IT. Especially, as computing becomes extremely distributed, networking becomes indispensable. Typically, a Docker host comprises multiple Docker containers and hence the networking has become a crucial component for realizing composite containerized applications. Docker containers also need to interact and collaborate with local as well as remote ones to come out with distributed applications. Precisely speaking, different and distributed containers need to be publicly found, network-accessible, and composable to bring forth business-centric and process-aware applications.

One of the key strengths of the Docker containerization paradigm is the ability to network seamlessly without much effort from the user. The earlier version of Docker supported just the bridge network; later, Docker acquired the SDN startup SocketPlane to add additional networking capabilities. Since then, Docker's networking capability has grown leaps and bounds and a separate set of subcommands, namely docker network connect, docker network create, docker network disconnect, docker network inspect, docker network ls, and docker network rm, were introduced to handle the nitty-gritty of the Docker networking. By default, during installation, the Docker Engine creates three networks for you, which you can list using the docker network ls subcommand, as shown here:

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
daa55dd5830a        bridge              bridge              local
3e99b1085979        host                host                local
9b06957b4a00        none                null                local
$ 
```

As you can see in the preceding screenshot, during the Docker setup, the Docker Engine creates the bridge, host, and none (null) networks. When Docker spins up a new container, by default, it creates a network stack for the container and attaches to the default bridge network. However, optionally, you could attach the container to the host or none network or the user-defined network using the --net option of the docker run subcommand. If you choose the host network, the container gets attached to the host network stack and shares the host's IP addresses and ports. The none network mode creates a network stack with just the Loopback (lo) interface. We can confirm this using the docker run --rm --net=none busybox ip addr command, as shown here:

```
$ docker run --rm  --net=none busybox ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    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
$ 
```

Evidently, as you can see in the preceding screenshot, the container has got just a Loopback interface. Since this container has got just a Loopback interface, the container cannot communicate with other containers or the external world.

The bridge network is the default network interface that Docker Engine assigns to a container if the network is not configured using the --net option of the docker run subcommand. To have a better understanding of the bridge network, let's begin by inspecting it using the docker network inspect subcommand, as shown here:

```
$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "daa55dd5830a4d5ad2cfa68085644baea2651a1a6ed8664ed8ef0a74b18f6bc5",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",        2
                    "Gateway": "172.17.0.1"           3
                }
            ]
        },
        "Internal": false,
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",      1
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
$
```

Here, in the preceding screenshot, we have highlighted three paramount insights. You can find the relevant description of what happens during the Docker installation process:

- docker0: Docker creates an Ethernet bridge interface inside the Linux kernel with the docker0 name on the Docker host. This interface is used as a bridge to pass the

Ethernet frames between containers and also between containers and an external network.

- Subnet: Docker also selects a private IP subnet from the address range of 172.17.0.0 to 172.17.255.255 and keeps it revered for its containers. In the preceding screenshot, Docker has selected the 172.17.0.0/16 subnet for the containers.
- Gateway: The docker0 interface is the gateway for the bridge network and Docker, from the IP subnet range selected earlier, assigns an IP address to docker0. Here, in the preceding example, 172.17.0.1 is assigned to the gateway.

We can cross-check the gateway address by listing the docker0 interface using the ip addr show Linux command:

```
$ ip addr show docker0
```

The third line of the output shows the assigned IP address and its network prefix:

```
inet 172.17.0.1/16 scope global docker0
```

Apparently, from the preceding text, 172.17.0.1 is the IP address assigned to docker0, the Ethernet bridge interface, which is also listed as the gateway address in the output of the docker network inspect bridge command.

Now that we have a clear understanding of the bridge creation and the subnet/gateway address selection process, let's explore the container networking in the bridge mode a bit more in detail. In the bridge network mode, the Docker Engine creates a network stack with a Loopback (lo) interface and an Ethernet (eth0) interface during the launch of the container. We can quickly examine this by running t

```
$ docker run -it busybox ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    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
201: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:3/64 scope link tentative
       valid_lft forever preferred_lft forever
$ 
```

he docker run --rm busybox ip addr command:

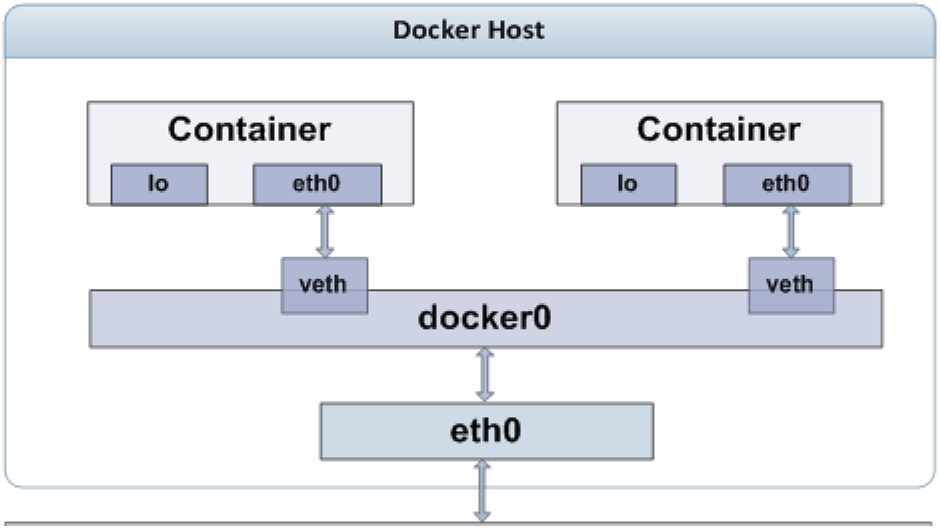
Evidently, the preceding output of the ip addr command shows that the Docker Engine has created a network stack for the container with two network interfaces, which are as follows:

- The first interface is the lo (Loopback) interface, for which the Docker Engine assigned the 127.0.0.1 Loopback address. The Loopback interface is used for local communication within a container.
- The second interface is an eth0 (Ethernet) interface, for which the Docker Engine assigned the 172.17.0.3 IP address. Obviously, this address also falls within the same IP address range of the docker0 Ethernet bridge interface. Besides, the address assigned to the eth0 interface is used for intra-container communication and host-to-container communication.

Note

The ip addr and/or ifconfig commands are not supported by all Docker images, including ubuntu:14.04 and ubuntu:16.04. The docker inspect subcommand is the reliable way to find the IP address of the container.

Earlier, we mentioned that docker0, the Ethernet bridge interface, acts as a conduit to pass the Ethernet frames between containers and also between containers and the external world. However, we have not yet clarified how the containers connect with the docker0 bridge. The following diagram unravels some of the mystery around this connection:



As depicted here, the container's eth0 interface is connected to the docker0 bridge using veth. The eth0 and veth interfaces belong to a special type of Linux network interface called a **Virtual Ethernet** (**veth**) Interface. The vethinterface always comes in a pair, and they are like a water pipe wherein the data send from one veth interface will come out of the other interface and vice versa. The Docker Engine assigns one of the veth interfaces to the container with the eth0 name and assigns the container IP address to that interface. The other veth interface of the pair is bound to the docker0 bridge interface. This ensures the seamless flow of data between the Docker host and the containers.

Docker assigns private IP addresses to the container, which is not reachable from outside of the Docker host. However, the container IP address comes in handy for debugging within the Docker host. As we noted earlier, many Docker images do not support the ip addr or ifconfig commands, besides we may not directly have access to the container prompt to run any of these commands. Fortunately, Docker provides a docker inspect subcommand, which is as handy as a Swiss Army knife, to dive deep into the low-level details of the Docker container or image. The docker inspect subcommand reports quite a lot of details including the IP address and the gateway address. For the practical purpose, here you can either select a running container or temporarily launch a container, as follows:

```
$ sudo docker run -itd ubuntu:16.04
```

Here, let's assume the container ID is 4b0b567b6019 and run the docker inspect subcommand, as shown here:

```
$ sudo docker inspect 4b0b567b6019
```

This command generates quite a lot of information about the container. Here, we show some excerpts of the container's network configuration from the output of the docker inspect subcommand:

```
"Networks": {
   "bridge": {
      "IPAMConfig": null,
      "Links": null,
      "Aliases": null,
      "NetworkID": "ID removed for readability",
      "EndpointID": "ID removed for readability",
      "Gateway": "172.17.0.1",
      "IPAddress": "172.17.0.3",
      "IPPrefixLen": 16,
      "IPv6Gateway": "",
      "GlobalIPv6Address": "",
      "GlobalIPv6PrefixLen": 0,
      "MacAddress": "02:42:ac:11:00:03"
   }
}
```

Here are the details of some of the important fields in the network configuration:

- Gateway: This is the gateway address of the container, which is the address of the bridge interface as well
- IPAddress: This is the IP address assigned to the container
- IPPrefixLen: This is the IP prefix length, another way of representing the subnet mask

Without doubt, the docker inspect subcommand is quite convenient to find the minute details of a container or an image. However, it's a tiresome job to go through the intimidating details and to find the right information that we are keenly looking for. Perhaps, you can narrow it down to the right information, using the grep command. Alternatively, even better, the docker inspect subcommand helps you pick the right field from the JSON array using the --format option of the docker inspect subcommand.

Notably, in the following example, we use the --format option of the docker inspect subcommand to retrieve just the IP address of the container. The IP address is accessible through the .NetworkSettings.IPAddress field of the JSON array:

> In addition to the none, host, and bridge networking modes, Docker also supports the overlay, macvlan, and ipvlan network modes.

Networking and Docker

Each Docker container has its own network stack, and this is due to the Linux kernel NET namespace, where a new NET namespace for each container is instantiated and cannot be seen from outside the container or from other containers.

Docker networking is powered by the following network components and services.

Linux bridges

These are L2/MAC learning switches built into the kernel and are to be used for forwarding.

Open vSwitch

This is an advanced bridge that is programmable and supports tunneling.

NAT

Network address translators are immediate entities that translate IP addresses and ports (SNAT, DNAT, and so on).

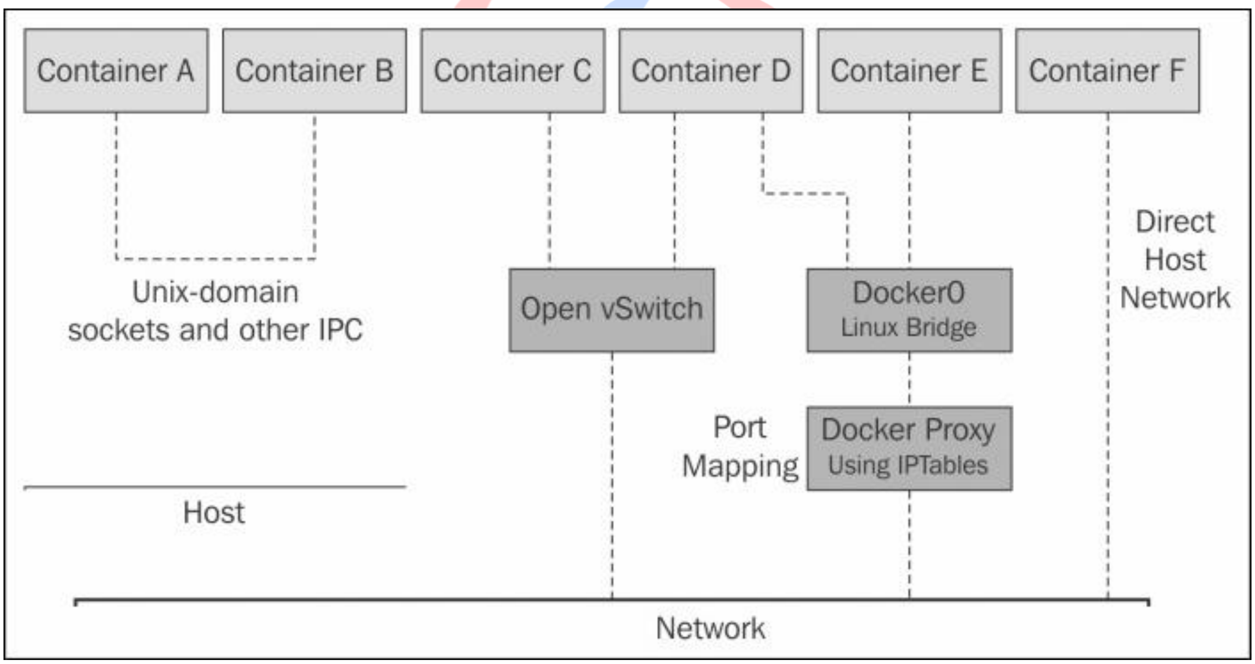
IPtables

This is a policy engine in the kernel used for managing packet forwarding, firewall, and NAT features.

AppArmor/SELinux

Firewall policies for each application can be defined with these.

Various networking components can be used to work with Docker, providing new ways to access and use Docker-based services. As a result, we see a lot of libraries that follow a different approach to networking. Some of the prominent ones are Docker Compose, Weave, Kubernetes, Pipework, libnetwork, and so on. The following figure depicts the root ideas of Docker networking:



The docker0 bridge

The docker0 bridge is the heart of default networking. When the Docker service is started, a Linux bridge is created on the host machine. The interfaces on the containers talk to the bridge, and the bridge proxies to the external world. Multiple containers on the same host can talk to each other through the Linux bridge.

docker0 can be configured via the --net flag and has, in general, four modes:

- --net default
- --net=none
- --net=container:$container2
- --net=host

The --net default mode

In this mode, the default bridge is used as the bridge for containers to connect to each other.

The --net=none mode

With this mode, the container created is truly isolated and cannot connect to the network.

The --net=container:$container2 mode

With this flag, the container created shares its network namespace with the container called $container2.

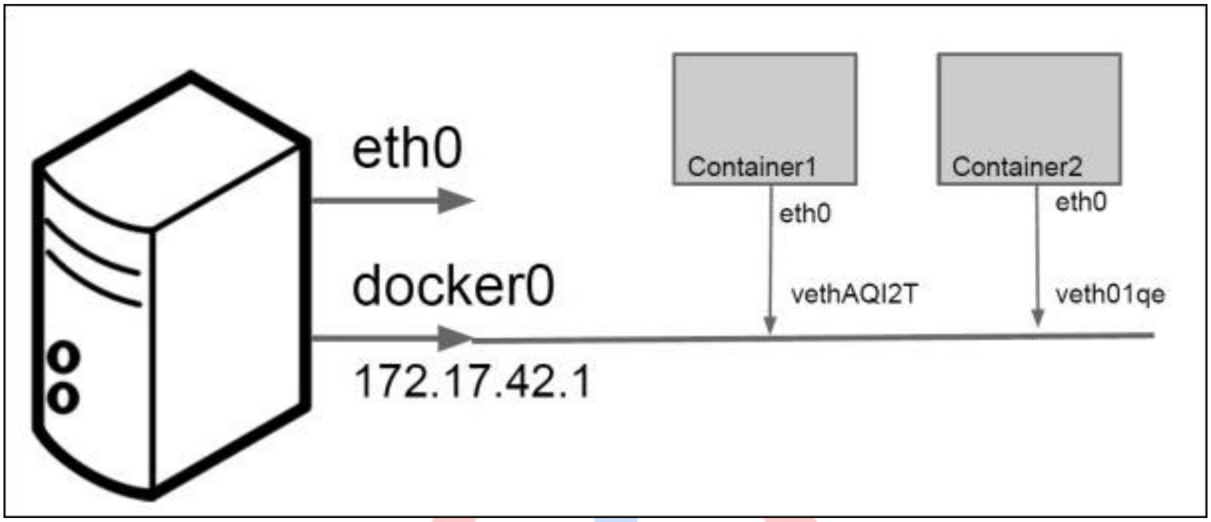
The --net=host mode

With this mode, the container created shares its network namespace with the host.

Port mapping in Docker container

In this section, we look at how container ports are mapped to host ports. This mapping can either be done implicitly by Docker Engine or can be specified.

If we create two containers called **Container1** and **Container2**, both of them are assigned an IP address from a private IP address space and also connected to the **docker0** bridge, as shown in the following figure:



Both the preceding containers will be able to ping each other as well as reach the external world.

For external access, their port will be mapped to a host port.

As mentioned in the previous section, containers use network namespaces. When the first container is created, a new network namespace is created for the container. A vEthernet link is created between the container and the Linux bridge. Traffic sent from eth0 of the container reaches the bridge through the vEthernet interface and gets switched thereafter. The following code can be used to show a list of Linux bridges:

```
# show linux bridges
$ sudo brctl show
```

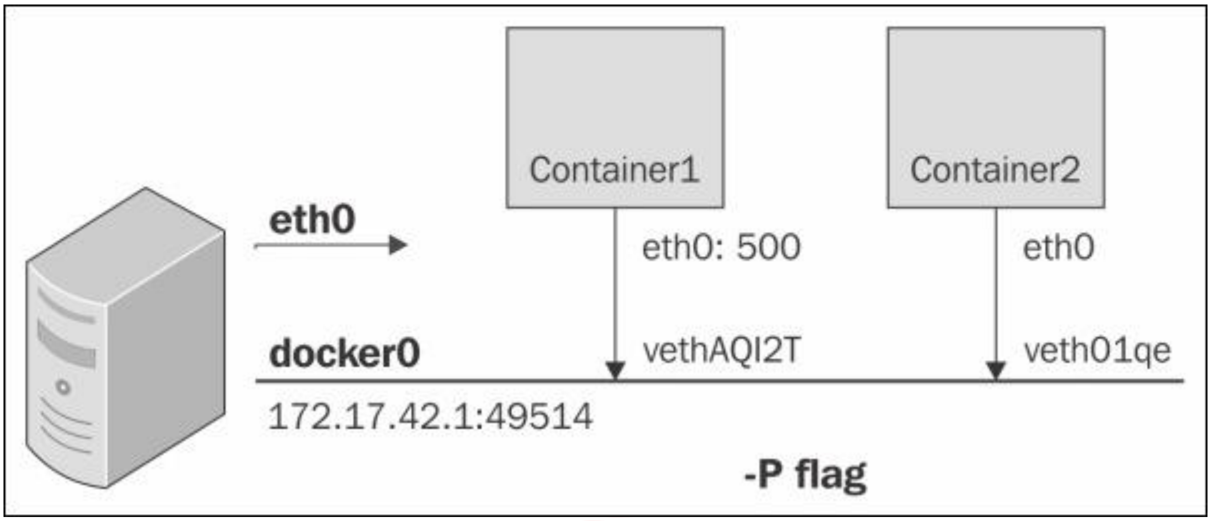
The output will be similar to the one shown as follows, with a bridge name and the veth interfaces on the containers it is mapped to:

```
bridge name     bridge id       STP enabled    interfaces
docker0     8000.56847afe9799      no       veth44cb727
                                         veth98c3700
```

How does the container connect to the external world? The iptables nat table on the host is used to masquerade all external connections, as shown here:

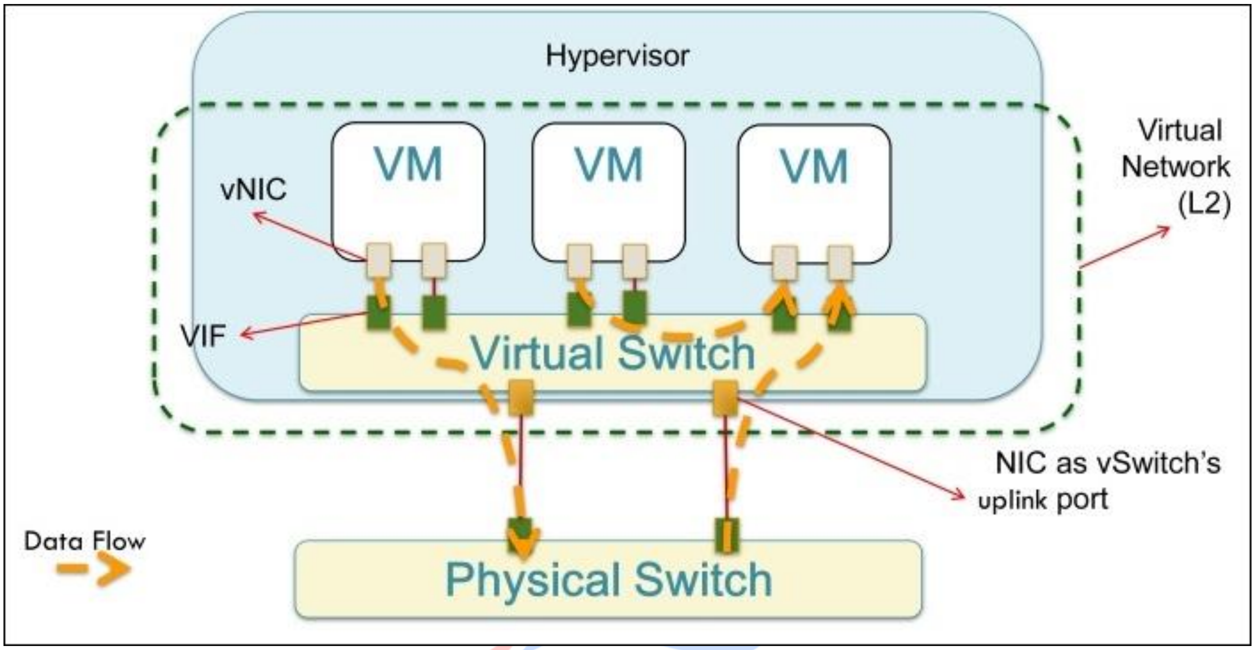
```
$ sudo iptables -t nat -L –n
...
Chain POSTROUTING (policy ACCEPT) target prot opt
source destination MASQUERADE all -- 172.17.0.0/16
!172.17.0.0/16
...
```

How to reach containers from the outside world? The port mapping is again done using the iptables nat option on the host machine.

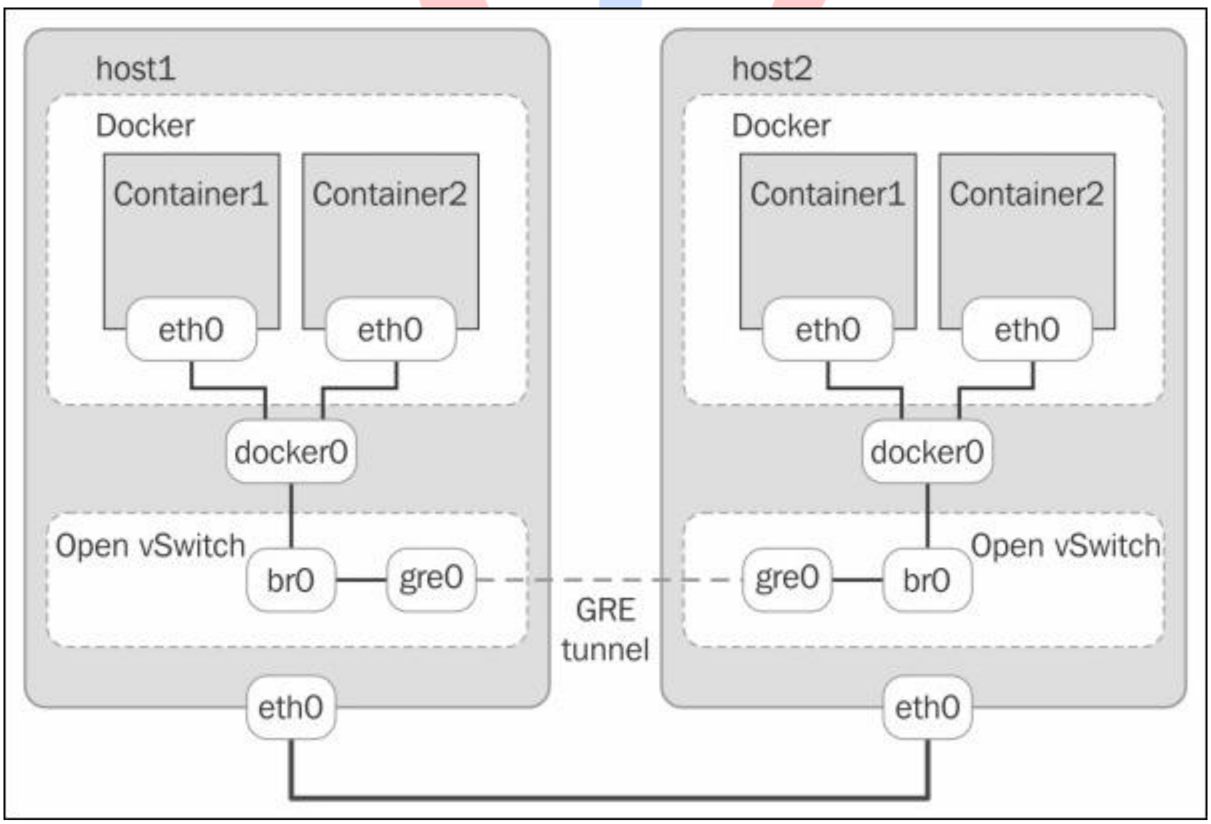


Docker OVS

Open vSwitch is a powerful network abstraction. The following figure shows how OVS interacts with the **VMs**, **Hypervisor**, and the **Physical Switch**. Every **VM** has a **vNIC** associated with it. Every **vNIC** is connected through a **VIF** (also called a **virtual interface**) with the **Virtual Switch**:



OVS uses tunnelling mechanisms such as GRE, VXLAN, or STT to create virtual overlays instead of using physical networking topologies and Ethernet components. The following figure shows how OVS can be configured for the containers to communicate between multiple hosts using GRE tunnels:



Linking Docker containers

In this section, we introduce the concept of linking two containers. Docker creates a tunnel between the containers, which doesn't need to expose any ports externally on the container. It uses environment variables as one of the mechanisms for passing information from the parent container to the child container.

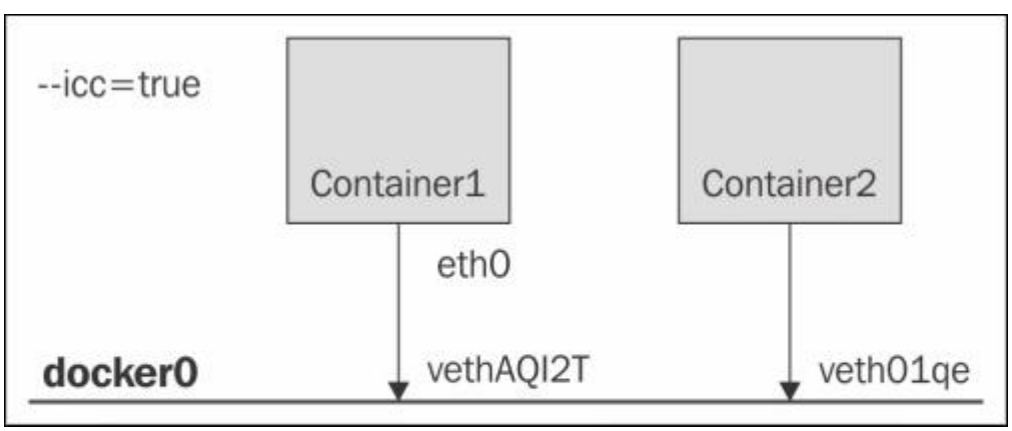
In addition to the environment variable env, Docker also adds a host entry for the source container to the /etc/hosts file. The following is an example of the host file:

```
$ docker run -t -i --name c2 --rm --link c1:c1alias training/webapp /bin/bash
root@<container_id>:/opt/webapp# cat /etc/hosts
172.17.0.1  aed84ee21bde
...
172.17.0.2  c1alaias 6e5cdeb2d300 c1
```

There are two entries:

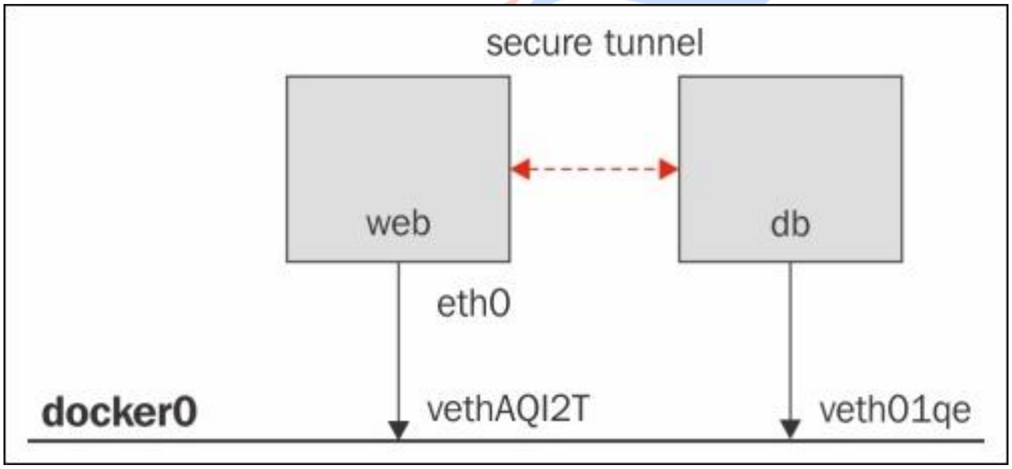
- The first is an entry for the container c2 that uses the Docker container ID as a host name
- The second entry, 172.17.0.2 c1alaias 6e5cdeb2d300 c1, uses the link alias to reference the IP address of the c1 container

The following figure shows two containers **Container 1** and **Container 2** connected using veth pairs to the docker0bridge with --icc=true. This means these two containers can access each other through the bridge:



Links

Links provide service discovery for Docker. They allow containers to discover and securely communicate with each other by using the flag -link name:alias. Inter-container communication can be disabled with the daemon flag -icc=false. With this flag set to false, **Container 1** cannot access **Container 2** unless explicitly allowed via a link. This is a huge advantage for securing your containers. When two containers are linked together, Docker creates a parent-child relationship between them, as shown in the following figure:



From the outside, it looks like this:

```
# start the database
$  sudo docker run -dp 3306:3306 --name todomvcdb \
-v /data/mysql:/var/lib/mysql cpswan/todomvc.mysql

# start the app server
$  sudo docker run -dp 4567:4567 --name todomvcapp \
--link todomvcdb:db cpswan/todomvc.sinatra
```

On the inside, it looks like this:

```
$   dburl = "mysql://root:pa55Word@" + \ ENV["DB_PORT_3306_TCP_ADDR"] + "/todomvc"
```

```
$ DataMapper.setup(:default, dburl)
```

What's new in Docker networking?

Docker networking is at a very nascent stage, and there are many interesting contributions from the developer community, such as Pipework, Weave, Clocker, and Kubernetes. Each of them reflects a different aspect of Docker networking. We will learn about them in later chapters. Docker, Inc. has also established a new project where networking will be standardized. It is called **libnetwork**.

libnetwork implements the **container network model** (**CNM**), which formalizes the steps required to provide networking for containers while providing an abstraction that can be used to support multiple network drivers. The CNM is built on three main components—sandbox, endpoint, and network.

Sandbox

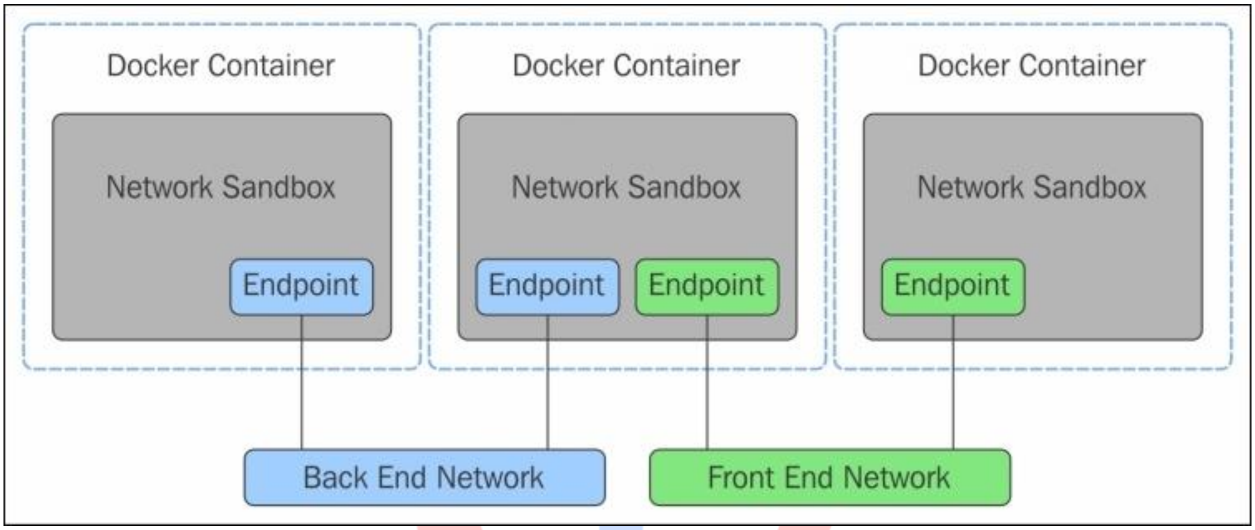
A sandbox contains the configuration of a container's network stack. This includes management of the container's interfaces, routing table, and DNS settings. An implementation of a sandbox could be a Linux network namespace, a FreeBSD jail, or other similar concept. A sandbox may contain many endpoints from multiple networks.

Endpoint

An endpoint connects a sandbox to a network. An implementation of an endpoint could be a veth pair, an Open vSwitch internal port, or something similar. An endpoint can belong to only one network but may only belong to one sandbox.

Network

A network is a group of endpoints that are able to communicate with each other directly. An implementation of a network could be a Linux bridge, a VLAN, and so on. Networks consist of many endpoints, as shown in the following diagram:



The Docker CNM model

The CNM provides the following contract between networks and containers:

- All containers on the same network can communicate freely with each other
- Multiple networks are the way to segment traffic between containers and should be supported by all drivers
- Multiple endpoints per container are the way to join a container to multiple networks
- An endpoint is added to a network sandbox to provide it with network connectivity

Envisaging container as a service

We laid a good foundation of the fundamentals of the Docker technology. In this section, we are going to focus on crafting an image with the HTTP service, launch the HTTP service inside the container using the crafted image, and then, demonstrate the connectivity to the HTTP service running inside the container.

Building an HTTP server image

In this section, we are going to craft a Docker image in order to install Apache2 on top of the Ubuntu 16.04 base image, and configure an Apache HTTP server to run as an executable, using the `ENTRYPOINT` instruction.

In this example, we are going to extend this `Dockerfile` by setting the Apache log path and setting Apache2 as the default execution application, using the `ENTRYPOINT` instruction. The following is a detailed explanation of the content of `Dockerfile`.

We are going to build an image using `ubuntu:16.04` as the base image, using the `FROM` instruction, as shown in the `Dockerfile` snippet:

```
###########################################
# Dockerfile to build an apache2 image
###########################################
# Base image is Ubuntu
FROM ubuntu:16.04
```

Set the author's detail using the `MAINTAINER` instruction:

```
# Author: Dr. Peter
MAINTAINER Dr. Peter <peterindia@gmail.com>
```

Using one `RUN` instruction, we will synchronize the APT repository source list, install the `apache2` package, and then clean the retrieved files:

```
# Install apache2 package
RUN apt-get update && \
    apt-get install -y apache2 && \
```

```
apt-get clean
```

Set the Apache log directory path using the ENV instruction:

```
# Set the log directory PATH
ENV APACHE_LOG_DIR /var/log/apache2
```

Now, the final instruction is to launch the apache2 server using the ENTRYPOINT instruction:

```
# Launch apache2 server in the foreground
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

In the preceding line, you might be surprised to see the FOREGROUND argument. This is one of the key differences between the traditional and the container paradigm. In the traditional paradigm, the server applications are usually launched in the background either as a service or a daemon because the host system is a general-purpose system. However, in the container paradigm, it is imperative to launch an application in the foreground because the images are crafted for a sole purpose.

Having prescribed the image building instruction in the Dockerfile, let's now move to the next logical step of building the image using the docker build subcommand by naming the image as apache2, as shown here:

```
$ sudo docker build -t apache2 .
```

Let's now do a quick verification of the images using the docker images subcommand:

```
$ sudo docker images
```

As we have seen in the previous chapters, the docker images command displays the details of all the images in the Docker host. However, in order to illustrate precisely the images created using the docker build subcommand, we highlight the details of apache2:latest (the target image) and ubuntu:16.04 (the base image) from the complete image list, as shown in the following output snippet:

```
apache2     latest    1b34e47c273d    About a minute ago   265.5 MB
ubuntu      16.04     f753707788c5    3 weeks ago          127.2 MB
```

Having built the HTTP server image, let's now move on to the next session to learn how to run the HTTP service.

Running the HTTP server image as a service

In this section, we are going to launch a container using the Apache HTTP server image, we crafted in the previous section. Here, we launch the container in the detached mode (similar to the UNIX daemon process) using the -d option of the docker run subcommand:

```
$ sudo docker run -d apache2
9d4d3566e55c0b8829086e9be2040751017989a47b5411c9c4f170ab865afcef
```

Having launched the container, let's run the docker logs subcommand to see whether our Docker container generates any output on its stdin (standard input) or stderr (standard error):

```
$ sudo docker logs \
9d4d3566e55c0b8829086e9be2040751017989a47b5411c9c4f170ab865afcef
```

As we have not fully configured the Apache HTTP server, you will find the following warning, as the output of the docker logs subcommand:

```
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.17.0.13. Set the 'ServerName' directive globally to suppress this message
```

From the preceding warning message, it is quite evident that the IP address assigned to this container is 172.17.0.13.

```
$ sudo docker inspect \
--format='{{.NetworkSettings.IPAddress}}'
9d4d3566e55c0b8829086e9be2040751017989a47b5411c9c4f170ab865afcef
```

```
172.17.0.13
```

Having found the IP address of the container as 172.17.0.13, let's quickly run a web request on this IP address from the shell prompt of the Docker host, using the wget command. Here, we choose to run the wget command with -qO -in order to run in the quiet mode and also display the retrieved HTML file on the screen:

```
$ wget -qO - 172.17.0.13
```

Here, we are showcasing just the first five lines of the retrieved HTML file:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<!--
   Modified from the Debian original for Ubuntu
   Last updated: 2014-03-19
```

Exposing container services

So far, we successfully launched an HTTP service and accessed the service from the Docker host as well as another container within the same Docker host

Handling Docker Containers, the container is able to successfully install the wget package by making a connection to the publicly available APT repository over the Internet. Nonetheless, the outside world cannot access the service offered by a container by default. At the outset, this might seem like a limitation in the Docker technology. However, the fact is, the containers are isolated from the outside world by design.

Docker achieves network isolation for the containers by the IP address assignment criteria, as enumerated here:

- Assigning a private IP address to the container, which is not reachable from an external network
- Assigning an IP address to the container outside the host's IP network

Consequently, the Docker container is not reachable even from the systems that are connected to the same IP network as the Docker host. This assignment scheme also provides protection from an IP address conflict that might otherwise arise.

Now, you might wonder how to make the services run inside a container that is accessible to the outside world, in other words, exposing container services. Well, Docker bridges this connectivity gap in a classy manner by leveraging the Linux iptables functionality under the hood.

At the frontend, Docker provides two different building blocks for bridging this connectivity gap for its users. One of the building blocks is to bind the container port using the -p (publish a container's port to the host interface) option of the docker run subcommand. Another alternative is to use the combination of the EXPOSE instruction of Dockerfileand the -P (publish all exposed ports to the host interfaces) option of the docker run subcommand.

Publishing a container's port – the -p option

Docker enables you to publish a service offered inside a container by binding the container's port to the host interface. The -p option of the docker run subcommand enables you to bind a container port to a user-specified or autogenerated port of the Docker host. Thus, any communication destined for the IP address and the port of the Docker host will be forwarded to the port of the container. The -p option, actually, supports the following four formats of arguments:

- <hostPort>:<containerPort>
- <containerPort>
- <ip>:<hostPort>:<containerPort>

- <ip>::<containerPort>

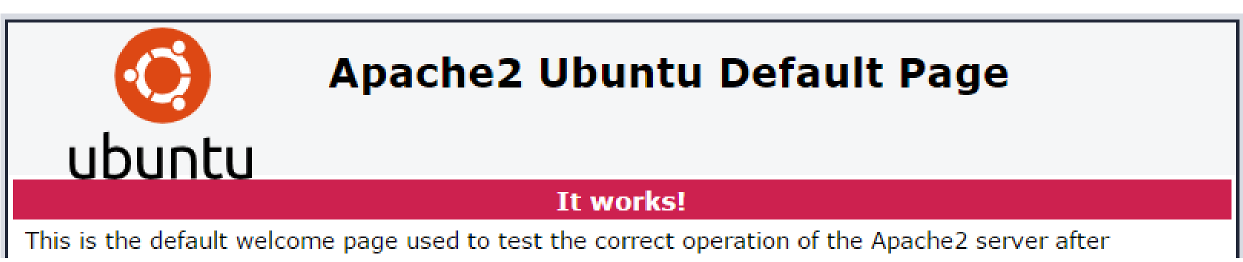
Here, <ip> is the IP address of the Docker host, <hostPort> is the Docker host port number, and <containerPort>is the port number of the container. Here, in this section, we present you with the -p <hostPort>:<containerPort> format and introduce other formats in the succeeding sections.

In order to understand the port binding process better, let's reuse the apache2 HTTP server image that we crafted previously and spin up a container using a -p option of the docker run subcommand. The 80 port is the published port of the HTTP service, and as the default behavior, our apache2 HTTP server is also available on port 80. Here, in order to demonstrate this capability, we are going to bind port 80 of the container to port 80 of the Docker host, using the -p <hostPort>:<containerPort> option of the docker run subcommand, as shown in the following command:

```
$ sudo docker run -d -p 80:80 apache2
baddba8afa98725ec85ad953557cd0614b4d0254f45436f9cb440f3f9eeae134
```

Now that we have successfully launched the container, we can connect to our HTTP server using any web browser from any external system (provided it has a network connectivity) to reach our Docker host.

So far, we have not added any web pages to our apache2 HTTP server image. Hence, when we connect from a web browser, we will get the following screen, which is nothing but the default page that comes along with the Ubuntu Apache2 package:



NAT for containers

In the previous section, we saw how a -p 80:80 option did the magic, didn't we? Well, in reality, under the hood, the Docker Engine achieves this seamless connectivity by automatically configuring the **Network Address Translation** (**NAT**) rule in the Linux iptables configuration files.

To illustrate the automatic configuration of the NAT rule in Linux iptables, let's query the Docker hosts iptables for its NAT entries, as follows:

```
$ sudo iptables -t nat -L -n
```

The ensuing text is an excerpt from the iptables NAT entry, which is automatically added by the Docker Engine:

```
Chain DOCKER (2 references)
target     prot opt source          destination
DNAT       tcp  --  0.0.0.0/0       0.0.0.0/0         tcp dpt:80 to:172.17.0.14:80
```

From the preceding excerpt, it is quite evident that the Docker Engine has effectively added a DNAT rule. The following are the details of the DNAT rule:

- The tcp keyword signifies that this DNAT rule applies only to the TCP transport protocol.
- The first 0.0.0.0/0 address is a meta IP address of the source address. This address indicates that the connection can originate from any IP address.
- The second 0.0.0.0/0 address is a meta IP address of the destination address on the Docker host. This address indicates that the connection can be made to any valid IP address in the Docker host.
- Finally, dpt:80 to:172.17.0.14:80 is the forwarding instruction used to forward any TCP activity on port 80 of the Docker host to be forwarded to the 172.17.0.17 IP address, the IP address of our container and port 80.

Note

Therefore, any TCP packet that the Docker host receives on port 80 will be forwarded to port 80 of the container.

Retrieving the container port

The Docker Engine provides at least three different options to retrieve the container's port binding details. Here, let's first explore the options, and then, move on to dissect the retrieved information. The options are as follows:

- The docker ps subcommand always displays the port binding details of a container, as shown here:

```
$ sudo docker ps
CONTAINER ID       IMAGE       COMMAND
CREATED            STATUS      PORTS
NAMES
baddba8afa98       apache2:latest
"/usr/sbin/apache2ct
26 seconds ago      Up 25 seconds
0.0.0.0:80->80/tcp
furious_carson
```

- The docker inspect subcommand is another alternative; however, you have to skim through quite a lot of details. Run the following command:

```
$ sudo docker inspect baddba8afa98
```

- The docker inspect subcommand displays the port binding related information in three JSON objects, as shown here:

- The ExposedPorts object enumerates all ports that are exposed through the EXPOSE instruction in Dockerfile, as well as the container ports that are mapped using the -p option in the docker run subcommand. Since we didn't add the EXPOSEinstruction in our Dockerfile, what we have is just the container port that was mapped using -p 80:80 as an argument to the docker run subcommand:

```
"ExposedPorts": {
          "80/tcp": {}
     },
```

- The PortBindings object is part of the HostConfig object, and this object lists out all the port binding done through the -p option in the docker run subcommand. This object will never list the ports exposed through the EXPOSE instruction in the Dockerfile:

```
"PortBindings": {
  "80/tcp": [
    {
      "HostIp": "",
      "HostPort": "80"
    }
  ]
},
```

- The Ports object of the NetworkSettings object has the same level of details, as the preceding PortBindings object. However, this object encompasses all ports that are exposed through the EXPOSE instruction in Dockerfile, as well as the container ports that are mapped using the -p option in the docker run subcommand:

Of course, the specific port field can be filtered using the --format option of the docker inspect subcommand.

The docker port subcommand enables you to retrieve the port binding on the Docker host by specifying the container's port number:

```
$ sudo docker port baddba8afa98 80
```

```
0.0.0.0:80
```

Evidently, in all the preceding output excerpts, the information that stands out is the 0.0.0.0 IP address and the 80 port number. The 0.0.0.0 IP address is a meta address, which represents all the IP addresses configured on the Docker host. In effect, the 80 container's port is bound to all the valid IP addresses on the Docker host. Therefore, the HTTP service is accessible through any of the valid IP addresses configured on the Docker host.

Binding a container to a specific IP address

Until now, with the method that you learned, the containers always get bound to all the IP addresses configured on the Docker host. However, you may want to offer different services on different IP addresses. In other words, a specific IP address and port would be configured to offer a particular service. We can achieve this in Docker using the -p <ip>:<hostPort>:<containerPort> option of the docker run subcommand, as shown in the following example:

```
$ sudo docker run -d -p 198.51.100.73:80:80 apache2
92f107537bebd48e8917ea4f4788bf3f57064c8c996fc23ea0fd8ea49b4f3335
```

Here, the IP address must be a valid IP address on the Docker host. If the specified IP address is not a valid IP address on the Docker host, the container launch will fail with an error message, as follows:

```
2014/11/09 10:22:10 Error response from daemon: Cannot start container
99db8d30b284c0a0826d68044c42c370875d2c3cad0b87001b858ba78e9de53b:
Error starting user land proxy: listen tcp 10.110.73.34:49153: bind:cannot assign requested address
```

Now, let's quickly review the port mapping as well the NAT entry for the preceding example:

- The following text is an excerpt from the output of the docker ps subcommand that shows the details of this container:

```
92f107537beb    apache2:latest    "/usr/sbin/apache2ct
```

```
About a minute ago   Up About a minute   198.51.100.73:80->80/tcp
boring_ptolemy
```

- The following text is an excerpt from the output of the iptables -n nat -L -n command that shows the DNAT entry created for this container:

```
DNAT    tcp -- 0.0.0.0/0     198.51.100.73     tcp dpt:80
to:172.17.0.15:80
```

After reviewing both the output of the docker run subcommand and the DNAT entry of iptables, you will realize how elegantly the Docker Engine has configured the service offered by the container on the 198.51.100.73 IP address and 80 port of the Docker host.

Autogenerating the Docker host port

The Docker containers are innately lightweight and due to their lightweight nature, you can run multiple containers with the same or different service on a single Docker host. Particularly, autoscaling of the same service across several containers based on the demand is the need of the IT infrastructure today. Here, in this section, you will be informed about the challenge in spinning up multiple containers with the same service and also the Docker's way of addressing this challenge.

Earlier in this chapter, we launched a container using Apache2 HTTP server by binding it to port 80 of the Docker host. Now, if we attempt to launch one more container with the same port 80 binding, the container would fail to start with an error message, as you can see in the following example:

```
$ sudo docker run -d -p 80:80 apache2
6f01f485ab3ce81d45dc6369316659aed17eb341e9ad0229f66060a8ba4a2d0e
2014/11/03 23:28:07 Error response from daemon: Cannot start container
6f01f485ab3ce81d45dc6369316659aed17eb341e9ad0229f66060a8ba4a2d0e:
Bind for 0.0.0.0:80 failed: port is already allocated
```

Obviously, in the preceding example, the container failed to start because the previous container is already mapped to 0.0.0.0 (all the IP addresses of the Docker host) and port 80. In the TCP/IP communication model, the combination of the IP address, port, and the transport protocols (TCP, UDP, and so on) has to be unique.

We could have overcome this issue by manually choosing the Docker host port number (for instance, -p 81:80 or -p 8081:80). Though this is an excellent solution, it does not scale well for autoscaling scenarios. Instead, if we give the control to Docker, it would autogenerate the port number on the Docker host. This port number generation is achieved by underspecifying the Docker host port number, using the -p <containerPort> option of the docker run subcommand, as shown in the following example:

```
$ sudo docker run -d -p 80 apache2
ea3e0d1b18cff40ffcddd2bf077647dc94bceffad967b86c1a343bd33187d7a8
```

Having successfully started the new container with the autogenerated port, let's review the port mapping as well the NAT entry for the preceding example:

- The following text is an excerpt from the output of the docker ps subcommand that shows the details of this container:

```
ea3e0d1b18cf      apache2:latest     "/usr/sbin/apache2ct
5 minutes ago     Up 5 minutes       0.0.0.0:49158->80/tcp
nostalgic_morse
```

- The following text is an excerpt from the output of the iptables -n nat -L -n command that shows the DNAT entry created for this container:

```
DNAT    tcp -- 0.0.0.0/0     0.0.0.0/0     tcp dpt:49158
to:172.17.0.18:80
```

After reviewing both the output of the docker run subcommand and the DNAT entry of iptables, what stands out is the 49158 port number. The 49158 port number is niftily autogenerated by the Docker Engine on the Docker host, with the help of the underlying operating system. Besides, the 0.0.0.0 meta IP address implies that the service offered by the container is accessible from outside, through any of the valid IP addresses configured on the Docker host.

You may have a use case where you want to autogenerate the port number. However, if you still want to restrict the service to a particular IP address of the Docker host, you can use the -p <IP>::<containerPort> option of the docker run subcommand, as shown in the following example:

```
$ sudo docker run -d -p 198.51.100.73::80 apache2
6b5de258b3b82da0290f29946436d7ae307c8b72f22239956e453356532ec2a7
```

In the preceding two scenarios, the Docker Engine autogenerated the port number on the Docker host and exposed it to the outside world. The general norm of network communication is to expose any service through a predefined port number so that anybody knows the IP address, and the port number can easily access the offered service. Whereas, here the port numbers are autogenerated and as a result, the outside world cannot directly reach the offered service. So, the primary purpose of this method of container creation is to achieve autoscaling, and the container created in this fashion would be interfaced with a proxy or load balance service on a predefined port.

Connecting to the HTTP service

In the preceding section, indecently, from the warning message, we find out that the IP address of the container is 172.17.0.13. On a fully configured HTTP server container, no such warning is available, so let's still run the docker inspect subcommand to retrieve the IP address using the container ID:

Sharing Data with Containers

The Docker technology has three different ways of providing persistent storage:

- The first and recommended approach is to use volumes that are created using Docker's volume management.
- The second method is to mount a directory from the Docker host to a specified location inside the container.
- The other alternative is to use a data-only container. The data-only container is a specially crafted container that is used to share data with one or more containers.

Data volume

Data volume is the fundamental building block of data sharing in the Docker environment. Before getting into the details of data sharing, it is imperative to get a good understanding of the data volume concept. Until now, all the files that we create in an image or a container is part and parcel of the union filesystem. The container's union filesystem perishes along with the container. In other words, when the container is removed, its filesystem is also automatically removed. However, the enterprise-grade applications must persist data and the container's filesystem will not render itself for such a requirement.

The Docker ecosystem, however, elegantly addresses this issue with the data volume concept. Data volume is essentially a part of the Docker host filesystem and it simply gets mounted inside the container. Since data volume is not a part of the container's filesystem, it has a life cycle independent of the container.

A data volume can be inscribed in a Docker image using the `VOLUME` instruction of the `Dockerfile`. Also, it can be prescribed during the launch of a container using the `-v` option of the `docker run` subcommand. Here, in the following example, the implication of the `VOLUME` instruction in the `Dockerfile` is illustrated in detail in the following steps:

1. Create a very simple `Dockerfile` with the instruction of the base image (`ubuntu:16.04`) and the data volume (`/MountPointDemo`):

```
FROM ubuntu:16.04
VOLUME /MountPointDemo
```

2. Build the image with the `mount-point-demo` name using the `docker build` subcommand:

```
$ sudo docker build -t mount-point-demo .
```

3. Having built the image, let's quickly inspect the image for our data volume using the `docker inspect` subcommand:

```
$ sudo docker inspect mount-point-demo
 [
     {
         "Id": "sha256:<64 bit hex id>",
         "RepoTags": [
            "mount-point-demo:latest"
         ],
... TRUNCATED OUTPUT ...
         "Volumes": {
            "/MountPointDemo": {}
         },
... TRUNCATED OUTPUT ...
```

Evidently, in the preceding output, data volume is inscribed in the image itself.

4. Now, let's launch an interactive container using the `docker run` subcommand from the earlier crafted image, as shown in the following command:

```
$ sudo docker run --rm -it mount-point-demo
```

5. From the container's prompt, let's check the presence of data volume using the `ls -ld` command:

```
root@8d22f73b5b46:/# ls -ld /MountPointDemo
drwxr-xr-x 2 root root 4096 Nov 18 19:22
/MountPointDemo
```

6. As mentioned earlier, data volume is part of the Docker host filesystem and it gets mounted, as shown in the following command:

```
root@8d22f73b5b46:/# mount | grep MountPointDemo
/dev/xvda2 on /MountPointDemo type ext3
(rw,noatime,nobarrier,errors=remount-ro,data=ordered)
```

7. In this section, we inspected the image to find out about the data volume declaration in the image. Now that we have launched the container, let's inspect the container's data volume using the docker inspect subcommand with the container ID as its argument in a different Terminal. We created a few containers previously and for this purpose, let's take the 8d22f73b5b46container ID directly from the container's prompt:

```
$ sudo docker inspect -f
 '{{json .Mounts}}' 8d22f73b5b46
    [
       {
          "Propagation": "",
          "RW": true,
          "Mode": "",
          "Driver": "local",
          "Destination": "/MountPointDemo",
          "Source":
"/var/lib/docker/volumes/720e2a2478e70a7cb49ab7385b8be627d4b6ec52e6bb33063e4144
355d59592a/_data",
"Name": "720e2a2478e70a7cb49ab7385b8be627d4b6ec52e6bb33063e4144355d59592a"
       }
    ]
```

Apparently, here, data volume is mapped to a directory in the Docker host, and the directory is mounted in the read-write mode. This directory, also called as volume, is created by the Docker Engine automatically during the launch of the container. Since version 1.9 of Docker, the volumes are managed through a top-level volume management command, which we will dive and dig further down into tell all in the next section.

So far, we have seen the implication of the VOLUME instruction in the Dockerfile, and how Docker manages data volume. Like the VOLUME instruction of the Dockerfile, we can use the -v <container mount point path> option of the docker run subcommand, as shown in the following command:

```
$ sudo docker run -v /MountPointDemo -it ubuntu:16.04
```

Having launched the container, we encourage you to try the ls -ld /MountPointDemo and mount commands in the newly launched container, and then also, inspect the container, as shown in the preceding step 5.

In both the scenarios described here, the Docker Engine automatically creates the volume under the /var/lib/docker/volumes/ directory and mounts it to the container. When a container is removed using the docker rm subcommand, the Docker Engine does not remove the volume that was automatically created during the launch of the container. This behavior is innately designed to preserve the state of the container's application that was stored in the volume filesystem. If you want to remove the volume that was automatically created by the Docker Engine, you can do so while removing the container by providing a -v option to the docker rm subcommand, on an already stopped container:

```
$ sudo docker rm -v 8d22f73b5b46
```

If the container is still running, then you can remove the container as well as the autogenerated directory by adding a -foption to the previous command:

```
$ sudo docker rm -fv 8d22f73b5b46
```

We have taken you through the techniques and tips to autogenerate a directory in the Docker host and mount it to the data volume in the container. However, with the -v option of the docker run subcommand, a user-defined directory can be mounted to the data volume. In such cases, the Docker Engine will not autogenerate any directory.

The volume management command

Docker has introduced a top-level volume management command from version 1.9 in order to manage the persistent filesystem effectively. The volume management command is capable of managing data volumes that are part of the Docker host. In addition to that, it also helps us to extend the Docker persistent capability using pluggable volume drivers (Flocker, GlusterFS, and so on). You can find the list of supported plugins at https://docs.docker.com/engine/extend/legacy_plugins/.

The docker volume command supports four subcommands as listed here:

- create: This creates a new volume
- inspect: This displays detailed information about one or more volumes
- ls: This lists the volumes in the Docker host

- rm: This removes a volume

Let's quickly explore the volume management command through a few examples. You can create a volume using the docker volume create subcommand, as shown here:

```
$ sudo docker volume create
50957995c7304e7d398429585d36213bb87781c53550b72a6a27c755c7a99639
```

The preceding command will create a volume by autogenerating a 64-hex digit string as the volume name. However, it is more effective to name the volume with a meaningful name for easy identification. You can name a volume using the --nameoption of the docker volume create subcommand:

```
$ sudo docker volume create --name example
example
```

Now, that we have created two volumes with and without a volume name, let's use the docker volume ls subcommand to display them:

```
$ sudo docker volume ls
DRIVER          VOLUME NAME
local        50957995c7304e7d398429585d36213bb87781c53550b72a6a27c755c7a99639
local        example
```

Having listed out the volumes, let's run the docker volume inspect subcommand into the details of the volumes we have created earlier:

```
$ sudo docker volume inspect example
[
   {
      "Name": "example",
      "Driver": "local",
      "Mountpoint":
      "/var/lib/docker/volumes/example/_data",
      "Labels": {},
      "Scope": "local"
   }
]
```

The docker volume rm subcommand enables you to remove the volumes you don't need anymore:

```
$ sudo docker volume rm example
```

example

Sharing host data

Docker does not provide any mechanism to mount the host directory or file during the build time in order to ensure the Docker images to be portable. The only provision Docker provides is to mount the host directory or file to a container's data volume during the container's launch. Docker exposes the host directory or file mounting facility through the -v option of the docker run subcommand. The -v option has five different formats, enumerated as follows:

- -v <container mount path>
- -v <host path>:<container mount path>
- -v <host path>:<container mount path>:<read write mode>
- -v <volume name>:<container mount path>
- -v <volume name>:<container mount path>:<read write mode>

The <host path> format is an absolute path in the Docker host, <container mount path> is an absolute path in the container filesystem, <volume name> is the name of the volume created using the docker volume create subcommand, and <read write mode> can be either the read-only (ro) or read-write (rw) mode. The first -v <container mount path> format has already been explained in the **Data volume** section in this chapter, as a method to create a mount point during the launch of the container launch. The second and third formats enable us to mount a file or directory from the Docker host to the container mount point. The fourth and fifth formats allow us to mount volumes created using the docker volume create subcommand.

In the first example, we will demonstrate how to share a directory between the Docker host and the container, and in the second example, we will demonstrate file sharing.

Here, in the first example, we mount a directory from the Docker host to a container, perform a few basic file operations on the container, and verify these operations from the Docker host, as detailed in the following steps:

1. First, let's launch an interactive container with the -v option of the docker run subcommand to mount /tmp/hostdirof the Docker host directory to /MountPoint of the container:

**$ sudo docker run -v /tmp/hostdir:/MountPoint **

```
    -it ubuntu:16.04
```

Note:

If /tmp/hostdir is not found on the Docker host, the Docker Engine will create the directory per se. However, the problem is that the system-generated directory cannot be deleted using the -v option of the docker rm subcommand.

2. Having successfully launched the container, we can check the presence of /MountPoint using the ls command:

```
root@4a018d99c133:/# ls -ld /MountPoint
drwxr-xr-x 2 root root 4096 Nov 23 18:28
/MountPoint
```

3. Now, we can proceed to check the mount details using the mount command:

```
root@4a018d99c133:/# mount | grep MountPoint
/dev/xvda2 on /MountPoint type ext3
(rw,noatime,nobarrier,errors=
remount-ro,data=ordered)
```

4. Here, we are going to validate /MountPoint, change to the /MountPoint directory using the cd command, create a few files using the touch command, and list the files using the ls command, as shown in the following script:

```
root@4a018d99c133:/# cd /MountPoint/
root@4a018d99c133:/MountPoint# touch {a,b,c}
root@4a018d99c133:/MountPoint# ls -l
total 0
-rw-r--r-- 1 root root 0 Nov 23 18:39 a
-rw-r--r-- 1 root root 0 Nov 23 18:39 b
```

```
-rw-r--r-- 1 root root 0 Nov 23 18:39 c
```

5. It might be worth the effort to verify the files in the /tmp/hostdir Docker host directory using the ls command on a new Terminal, as our container is running in an interactive mode on the existing Terminal:

```
$ sudo  ls -l /tmp/hostdir/
total 0
-rw-r--r-- 1 root root 0 Nov 23 12:39 a
-rw-r--r-- 1 root root 0 Nov 23 12:39 b
-rw-r--r-- 1 root root 0 Nov 23 12:39 c
```

Here, we can see the same set of files, as we saw in step 4. However, you might have noticed the difference in the timestamp of the files. This time difference is due to the time zone difference between the Docker host and the container.

6. Finally, let's run the docker inspect subcommand with the 4a018d99c133 container ID as an argument to see whether the directory mapping is set up between the Docker host and the container mount point, as shown in the following command:

```
$ sudo docker inspect \
          --format='{{json .Mounts}}' 4a018d99c133
[{"Source":"/tmp/hostdir",
"Destination":"/MountPoint","Mode":"",
"RW":true,"Propagation":"rprivate"}]
```

Apparently, in the preceding output of the docker inspect subcommand, the /tmp/hostdir directory of the Docker host is mounted on the /MountPoint mount point of the container.

For the second example, we will mount a file from the Docker host to a container, update the file from the container, and verify those operations from the Docker host, as detailed in the following steps:

1. In order to mount a file from the Docker host to the container, the file must preexist in the Docker host. Otherwise, the Docker Engine will create a new directory with the

specified name and mount it as a directory. We can start by creating a file on the Docker host using the touch command:

```
$ touch /tmp/hostfile.txt
```

2. Launch an interactive container with the -v option of the docker run subcommand to mount the /tmp/hostfile.txtDocker host file to the container as /tmp/mntfile.txt:

```
$ sudo docker run -v /tmp/hostfile.txt:/mntfile.txt \
        -it ubuntu:16.04
```

3. Having successfully launched the container, now let's check the presence of /mntfile.txt using the ls command:

```
root@d23a15527eeb:/# ls -l /mntfile.txt
-rw-rw-r-- 1 1000 1000 0 Nov 23 19:33 /mntfile.txt
```

4. Then, proceed to check the mount details using the mount command:

```
root@d23a15527eeb:/# mount | grep mntfile
/dev/xvda2 on /mntfile.txt type ext3
(rw,noatime,nobarrier,errors=remount-ro,data=ordered)
```

5. Then, update some text to /mntfile.txt using the echo command:

```
root@d23a15527eeb:/# echo "Writing from Container"
> mntfile.txt
```

6. Meanwhile, switch to a different Terminal in the Docker host, and print the /tmp/hostfile.txt Docker host file using the cat command:

```
$ cat /tmp/hostfile.txt
Writing from Container
```

7. Finally, run the docker inspect subcommand with the d23a15527eeb container ID as it's argument to see the file mapping between the Docker host and the container mount point:

```
$ sudo docker inspect \
          --format='{{json .Mounts}}' d23a15527eeb
[{"Source":"/tmp/hostfile.txt",
"Destination":"/mntfile.txt",
"Mode":"","RW":true,"Propagation":"rprivate"}]
```

From the preceding output, it is evident that the /tmp/hostfile.txt file from the Docker host is mounted as /mntfile.txt inside the container.

For the last example, we will create a Docker volume and mount a named data volume to a container. In this example, we are not going to run the verification steps as we did in the previous two examples. However, you are encouraged to run the verification steps we laid out in the first example.

1. Create a named data volume using the docker volume create subcommand, as shown here:

```
$ docker volume create --name namedvol
```

2. Now, launch an interactive container with the -v option of the docker run subcommand to mount namedvol a named data value to /MountPoint of the container:

```
$ sudo docker run -v namedvol:/MountPoint \
            -it ubuntu:16.04
```

Note: During the launch of the container, Docker Engine creates namedvol if it is not created already.

3. Having successfully launched the container, you can repeat the verification steps 2 to 6 of the first example and you will find the same output pattern in this example as well.

The practicality of host data sharing

In the previous chapter, we launched an HTTP service in a Docker container. However, if you remember correctly, the log file for the HTTP service is still inside the container, and it cannot be accessed directly from the Docker host. Here, in this section, we elucidate the procedure of accessing the log files from the Docker host in a step-by-step manner:

1. Let's begin with launching an Apache2 HTTP service container by mounting the /var/log/myhttpd directory of the Docker host to the /var/log/apache2 directory of the container, using the -v option of the docker run subcommand. In this example, we are leveraging the apache2 image, which we had built in the previous chapter, by invoking the following command:

```
$ sudo docker run -d -p 80:80 \
      -v /var/log/myhttpd:/var/log/apache2 apache2
9c2f0c0b126f21887efaa35a1432ba7092b69e0c6d523ffd50684e27eeab37ac
```

If you recall the Dockerfile in **Running Services in a Container**, the APACHE_LOG_DIR environment variable is set to the /var/log/apache2 directory, using the ENV instruction. This will make the Apache2 HTTP service to route all log messages to the /var/log/apache2 data volume.

2. Once the container is launched, we can change the directory to /var/log/myhttpd on the Docker host:

```
$ cd /var/log/myhttpd
```

3. Perhaps, a quick check of the files present in the /var/log/myhttpd directory is appropriate here:

```
$ ls -1
```

```
access.log
error.log
other_vhosts_access.log
```

Here, the access.log file contains all the access requests handled by the Apache2 HTTP server. The error.log file is a very important log file, where our HTTP server records the errors it encounters while processing any HTTP requests. The other_vhosts_access.log file is the virtual host log, which will always be empty in our case.

4. We can display the content of all the log files in the /var/log/myhttpd directory using the tail command with the –foption:

```
$ tail -f *.log
==> access.log <==

==> error.log <==
AH00558: apache2: Could not reliably determine the
server's fully qualified domain name, using 172.17.0.17.
Set the 'ServerName' directive globally to suppress this
message
[Thu Nov 20 17:45:35.619648 2014] [mpm_event:notice]
[pid 16:tid 140572055459712] AH00489: Apache/2.4.7
(Ubuntu) configured -- resuming normal operations
[Thu Nov 20 17:45:35.619877 2014] [core:notice]
[pid 16:tid 140572055459712] AH00094: Command line:
'/usr/sbin/apache2 -D FOREGROUND'

==> other_vhosts_access.log <==
```

5. The tail -f command will run continuously and display the content of the files, as soon as they get updated. Here, both access.log and other_vhosts_access.log are empty, and there are a few error messages on the error.log file. Apparently, these error logs are generated by the HTTP service running inside the container. The logs are then stocked in the Docker host directory, which is mounted during the launch of the container.

```
==> access.log <==
111.111.172.18 - - [20/Nov/2014:17:53:38 +0000] "GET /
HTTP/1.1" 200 3594 "-" "Mozilla/5.0 (Windows NT 6.1;
WOW64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.65
Safari/537.36"
```

```
111.111.172.18 - - [20/Nov/2014:17:53:39 +0000] "GET
/icons/ubuntu-logo.png HTTP/1.1" 200 3688
"http://111.71.123.110/" "Mozilla/5.0 (Windows NT 6.1;
WOW64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.65
Safari/537.36"
111.111.172.18 - - [20/Nov/2014:17:54:21 +0000] "GET
/favicon.ico HTTP/1.1" 404 504 "-" "Mozilla/5.0 (Windows
NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/39.0.2171.65 Safari/537.36"
```

Sharing data between containers

In the previous section, you learned how seamlessly the Docker Engine enables data sharing between the Docker host and the container. Although it is a very effective solution for most of the use cases, there are use cases wherein you will have to share data between one or more containers. The Docker's prescription to address this use case is to mount the data volume of one container to other containers using the `--volume-from` option of the `docker run` subcommand.

Data-only containers

Before Docker introduced the top-level volume management feature, the data-only container was the recommended approach to achieve data persistency. It is worth understanding data-only containers because you will find many implementations that are based on data-only containers. The prime responsibility of a data-only container is to preserve the data. Creating a data-only container is very similar to the method illustrated in the **Data volume** section. In addition, the containers are named explicitly for other containers to mount the data volume using the container's name. Besides, the container's data volumes are accessible from other containers even when the data-only containers are in the stopped state. The data-only containers can be created in two ways, as follows:

- During the launch of the container by configuring the data volume and the container's name

- Data volume can also be inscribed with Dockerfile during image-building, and later, the container can be named during the container's launch

In the following example, we are launching a data-only container by configuring the container launch with the -v and --name options of the docker run subcommand, as shown here:

```
$ sudo docker run --name datavol \
            -v /DataMount \
            busybox:latest /bin/true
```

Here, the container is launched from the busybox image, which is widely used for its smaller footprint. Here, we choose to execute the /bin/true command because we don't intend to do any operations on the container. Therefore, we named the container datavol using the --name option and created a new /DataMount data volume using the -v option of the docker run subcommand. The /bin/true command exits immediately with the 0 exit status, which in turn will stop the container and continue to be in the stopped state.

Mounting data volume from other containers

The Docker Engine provides a nifty interface to mount (share) the data volume from one container to another. Docker makes this interface available through the --volumes-from option of the docker run subcommand. The --volumes-fromoption takes a container name or container ID as its input and automatically mounts all the data volumes available on the specified container. Docker allows you to mount multiple containers with data volume using the --volumes-from option multiple times.

Here is a practical example that demonstrates how to mount data volume from another container and showcases the data volume mount step by step:

1. We begin with launching an interactive Ubuntu container by mounting the data volume from the data-only container (datavol), which we launched in the previous section:

```
$ sudo docker run -it \
            --volumes-from datavol \
            ubuntu:latest /bin/bash
```

2. Now from the container's prompt, let's verify the data volume mounts using the mount command:

```
root@e09979cacec8:/# mount | grep DataMount
/dev/xvda2 on /DataMount type ext3
(rw,noatime,nobarrier,errors=remount-ro,data=ordered)
```

Here, we successfully mounted the data volume from the datavol data-only container.

3. Next, we need to inspect the data volume of this container from another Terminal using the docker inspect subcommand:

```
$ sudo docker inspect --format='{{json .Mounts}}'
    e09979cacec8
    [{"Name":
    "7907245e5962ac07b31c6661a4dd9b283722d3e7d0b0fb40a90
    43b2f28365021","Source":
    "/var/lib/docker/volumes
    /7907245e5962ac07b31c6661a4dd9b283722d3e7d0b0fb40a9043b
    2f28365021/_data","Destination":"
    /DataMount","Driver":"local","Mode":"",
    "RW":true,"Propagation":""}]
```

Evidently, the data volume from the datavol data-only container is mounted as if they were mounted directly on to this container.

We can mount a data volume from another container and also showcase the mount points. We can make the mounted data volume to work by sharing data between containers using the data volume, as demonstrated here:

1. Let's reuse the container that we launched in the previous example and create a /DataMount/testfile file in the /DataMount data volume by writing some text to the file, as shown here:

```
root@e09979cacec8:/# echo \
        "Data Sharing between Container" > \
          /DataMount/testfile
```

2. Just spin off a container to display the text that we wrote in the previous step, using the cat command:

```
$ sudo docker run --rm \
            --volumes-from datavol \
            busybox:latest cat /DataMount/testfile
```

3. The following is the typical output of the preceding command:

```
Data Sharing between Container
```

Evidently, the preceding Data Sharing between Container output of our newly containerized cat command is the text that we have written in /DataMount/testfile of the datavol container in step 1.

Cool, isn't it? You can share data seamlessly between containers by sharing the data volumes. Here, in this example, we used data-only containers as the base container for data sharing. However, Docker allows us to share any type of data volumes and to mount data volumes one after another, as depicted here:

```
$ sudo docker run --name vol1 --volumes-from datavol \
       busybox:latest /bin/true
$ sudo docker run --name vol2 --volumes-from vol1 \
       busybox:latest /bin/true
```

Here, in the vol1 container, we mounted the data volume from the datavol container. Then, in the vol2 container, we mounted the data volume from the vol1 container, which is eventually from the datavol container.

The practicality of data sharing between containers

Earlier in this chapter, you learned the mechanism of accessing the log files of the Apache2 HTTP service from the Docker host. Although it was fairly convenient to share data by mounting the Docker host directory to a container, later we came to know that data can be shared between containers by just using data volumes. So here, we are bringing in a twist to the method of the Apache2 HTTP service log handling by sharing data between containers. To share log files between containers, we will spin off the following containers as enlisted in the following steps:

1. First, a data-only container that will expose the data volume to other containers.
2. Then, an Apache2 HTTP service container leveraging the data volume of the data-only container.
3. A container to view the log files generated by our Apache2 HTTP service.

Note:

If you are running any HTTP service on the 80 port number of your Docker host machine, pick any other unused port number for the following example. If not, first stop the HTTP service, then proceed with the example in order to avoid any port conflict.

Now, we'll meticulously walk you through the steps to craft the respective images and launch the containers to view the log files:

1. Here, we begin with crafting a Dockerfile with the /var/log/apache2 data volume using the VOLUME instruction. The /var/log/apache2 data volume is a direct mapping to APACHE_LOG_DIR, the environment variable set in the Dockerfile in **Running Services in a Container**, using the ENV instruction:

```
###########################################################
# Dockerfile to build a LOG Volume for Apache2 Service
###########################################################
# Base image is BusyBox
FROM busybox:latest
# Author: Dr. Peter
MAINTAINER Dr. Peter <peterindia@gmail.com>
# Create a data volume at /var/log/apache2, which is
# same as the log directory PATH set for the apache image
VOLUME /var/log/apache2
# Execute command true
CMD ["/bin/true"]
```

Since this Dockerfile is crafted to launch data-only containers, the default execution command is set to /bin/true.

2. We will continue to build a Docker image with the apache2log name from the preceding Dockerfile using docker build, as presented here:

```
$ sudo docker build -t apache2log .
Sending build context to Docker daemon 2.56 kB
Sending build context to Docker daemon
Step 0 : FROM busybox:latest
... TRUNCATED OUTPUT ...
```

3. Launch a data-only container from the apache2log image using the docker run subcommand and name the resulting container log_vol, using the --name option:

```
$ sudo docker run --name log_vol apache2log
```

Acting on the preceding command, the container will create a data volume in /var/log/apache2 and move it to a stop state

4. Meanwhile, you can run the docker ps subcommand with the -a option to verify the container's state:

```
$ sudo docker ps -a
CONTAINER ID      IMAGE            COMMAND
CREATED           STATUS           PORTS
NAMES
40332e5fa0ae      apache2log:latest    "/bin/true"
2 minutes ago     Exited (0) 2 minutes ago
log_vol
```

As per the output, the container exits with the 0 exit value.

5. Launch the Apache2 HTTP service using the docker run subcommand. Here, we are reusing the apache2 image we crafted in **Running Services in a Container**. Besides, in this container, we will mount the /var/log/apache2 data volume from log_vol, the data-only container that we launched in step 3, using the --volumes-from option:

```
$ sudo docker run -d -p 80:80 \
```

```
        --volumes-from log_vol \
        apache2
7dfbf87e341c320a12c1baae14bff2840e64afcd082dda3094e7cb0a0023cf42
```

With the successful launch of the Apache2 HTTP service with the /var/log/apache2 data volume mounted from log_vol, we can access the log files using transient containers.

6. Here, we are listing the files stored by the Apache2 HTTP service, using a transient container. This transient container is spun off by mounting the /var/log/apache2 data volume from log_vol, and the files in /var/log/apache2 are listed using the ls command. Further, the --rm option of the docker run subcommand is used to remove the container once it is done executing the ls command:

```
$ sudo docker run --rm \
            --volumes-from log_vol \
            busybox:latest ls -l /var/log/apache2
total 4
-rw-r--r--   1   root   root   0 Dec 5 15:27
access.log
-rw-r--r--   1   root   root   461 Dec 5 15:27
error.log
-rw-r--r--   1   root   root   0 Dec 5 15:27
other_vhosts_access.log
```

7. Finally, the error log produced by the Apache2 HTTP service is accessed using the tail command, as highlighted in the following command:

```
$ sudo docker run --rm \
            --volumes-from log_vol \
            ubuntu:16.04 \
            tail /var/log/apache2/error.log
AH00558: apache2: Could not reliably determine the
server's fully qualified domain name, using 172.17.0.24.
Set the 'ServerName' directive globally to suppress this
message
[Fri Dec 05 17:28:12.358034 2014] [mpm_event:notice]
[pid 18:tid 140689145714560] AH00489: Apache/2.4.7
(Ubuntu) configured -- resuming normal operations
[Fri Dec 05 17:28:12.358306 2014] [core:notice]
```

```
[pid 18:tid 140689145714560] AH00094: Command line:
'/usr/sbin/apache2 -D FOREGROUND'
```

Docker inbuilt service discovery

The Docker platform inherently supports the service discovery for the containers that are attached to any user-defined network using an embedded **Domain Name Service** (**DNS**). This functionality has been added to Docker since the version `1.10`. The embedded DNS feature enables the Docker containers to discover each other using their names or aliases within the user-defined network. In other words, the name resolution request from the container is first sent to the embedded DNS. The user-defined network then uses a special `127.0.0.11` IP address for the embedded DNS, which is also listed in `/etc/resolv.conf`.

The following example will help to gain a better understanding of Docker's built-in service discovery capability:

1. Let's begin by creating a user-defined bridge network, `mybridge`, using the following command:

```
$ sudo docker network create mybridge
```

2. Inspect the newly created network to understand the subnet range and gateway IP:

```
$ sudo docker network inspect mybridge
[
  {
    "Name": "mybridge",
    "Id":
"36e5e088543895f6d335eb92299ee8e118cd0610e0d023f7c42e6e603b935e17",
    "Created":
    "2017-02-12T14:56:48.553408611Z",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
```

```
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

Here, the subnet assigned to the mybridge network is 172.18.0.0/16 and the gateway is 172.18.0.1.

3. Now, let's create a container by attaching it to the mybridge network, as shown here:

```
$ sudo docker container run -itd --net mybridge --name testdns ubuntu
```

4. Continue to list the IP address assigned to the container, as illustrated here:

```
$ sudo docker container inspect --format \
'{{.NetworkSettings.Networks.mybridge.IPAddress}}' \
testdns
172.18.0.2
```

Evidently, the testdns container is assigned a 172.18.0.2 IP address. The 172.18.0.2 IP address is from the subnet of the mybridge network (that is, 172.18.0.0/16).

5. Having got the IP address of the container, let's look into the content of the /etc/resolv.conf file of the container using the docker container exec subcommand, as shown here:

```
$ sudo docker container exec testdns \
cat /etc/resolv.conf
nameserver 127.0.0.11
```

```
options ndots:0
```

6. As a final step, let's ping the testdns container using the busybox image. We picked the busybox image here because the ubuntu image is shipped without the ping command:

```
$ sudo docker container run --rm --net mybridge \
                busybox ping -c 2 testdns
PING testdns (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64
time=0.085 ms
64 bytes from 172.18.0.2: seq=1 ttl=64
time=0.133 ms

--- testdns ping statistics ---
2 packets transmitted, 2 packets received,
0% packet loss
round-trip min/avg/max = 0.085/0.109/0.133 ms
```

Linking containers

Before the introduction of the concept of the user-defined network, container linking was predominantly used for inter-container discovery and communication. That is, cooperating containers can be linked together to offer complex and business-aware services. The linked containers have a kind of source-recipient relationship, wherein the source container gets linked to the recipient container, and the recipient securely receives a variety of information from the source container. However, the source container will know nothing about the recipients to which it is linked. Another noteworthy feature of linking containers in a secured setup is that the linked containers can communicate using secure tunnels without exposing the ports used for the setup to the external world. Though you will find lots of deployments that use container-linking techniques, they are cumbersome and time-consuming to configure.

Also, they are error-prone. So the new method of embedded DNS is highly preferred over the traditional container-linking techniques.

The Docker Engine provides the --link option in the docker run subcommand to link a source container to a recipient container.

The format of the --link option is as follows:

```
--link <container>:<alias>
```

Here, <container> is the name of the source container and <alias> is the name seen by the recipient container. The name of the container must be unique in a Docker host, whereas alias is very specific and local to the recipient container, and hence, the alias need not be unique in the Docker host. This gives a lot of flexibility to implement and incorporate functionalities with a fixed source alias name inside the recipient container.

When two containers are linked together, the Docker Engine automatically exports a few environment variables to the recipient container. These environment variables have a well-defined naming convention, where the variables are always prefixed with the capitalized form of the alias name. For instance, if src is the alias name given to the source container, then the exported environment variables will begin with SRC_. Docker exports three categories of environment variables, as enumerated here:

- NAME: This is the first category of environment variables. These variables take the form of <ALIAS>_NAME, and they carry the recipient container's hierarchical name as their value. For instance, if the source container's alias is src and the recipient container's name is rec, then the environment variable and its value will be SRC_NAME=/rec/src.
- ENV: This is the second category of environment variables used to export the environment variables configured in the source container by the -e option of the docker run subcommand or the ENV instruction of the Dockerfile. This type of an environment variable takes the form of <ALIAS>_ENV_<VAR_NAME>. For instance, if the source container's alias is src and the variable name is SAMPLE, then the environment variable will be SRC_ENV_SAMPLE.

- PORT: This is the final and third category of environment variables that is used to export the connectivity details of the source container to the recipient. Docker creates a bunch of variables for each port exposed by the source container through the -p option of the docker run subcommand or the EXPOSE instruction of the Dockerfile.

These variables take the <ALIAS>_PORT_<port>_<protocol> form. This form is used to share the source's IP address, port, and protocol as a URL. For example, if the source container's alias is src, the exposed port is 8080, the protocol is tcp, and the IP address is 172.17.0.2, then the environment variable and its value will be SRC_PORT_8080_TCP=tcp://172.17.0.2:8080. This URL further splits into the following three environment variables:

- <ALIAS>_PORT_<port>_<protocol>_ADDR: This form carries the IP address part of the URL (for example, SRC_PORT_8080_TCP_ADDR= 172.17.0.2)
- <ALIAS>_PORT_<port>_<protocol>_PORT: This form carries the port part of the URL (for example, SRC_PORT_8080_TCP_PORT=8080)
- <ALIAS>_PORT_<port>_<protocol>_PROTO: This form carries the protocol part of the URL (for example, SRC_PORT_8080_TCP_PROTO=tcp)

In addition to the preceding environment variables, the Docker Engine exports one more variable in this category, that is, of the <ALIAS>_PORT form, and its value will be the URL of the lowest number of all the exposed ports of the source container. For instance, if the source container's alias is src, the exposed port numbers are 7070, 8080, and 80, the protocol is tcp, and the IP address is 172.17.0.2, then the environment variable and its value will be SRC_PORT=tcp://172.17.0.2:80.

Docker exports these autogenerated environment variables in a well-structured format so that they can be easily discovered programmatically. Thus, it becomes very easy for the recipient container to discover the information about the source container. In addition, Docker automatically updates the source IP address and its alias as an entry in the /etc/hosts file of the recipient.

In this chapter, we will dive deep into the mentioned features provided by the Docker Engine for container linkage through a bevy of pragmatic examples.

To start with, let's choose a simple container linking example. Here, we will show you how to establish a linkage between two containers, and transfer some basic information from the source container to the recipient container, as illustrated in the following steps:

1. We begin with launching an interactive container that can be used as a source container for linking, using the following command:

```
$ sudo docker run --rm --name example -it \
     busybox:latest
```

The container is named example using the --name option. In addition, the --rm option is used to clean up the container as soon as you exit from the container.

2. Display the /etc/hosts entry of the source container using the cat command:

```
/ # cat /etc/hosts
     172.17.0.3      a02895551686
     127.0.0.1       localhost
     ::1      localhost ip6-localhost ip6-loopback
     fe00::0  ip6-localnet
     ff00::0  ip6-mcastprefix
     ff02::1  ip6-allnodes
     ff02::2  ip6-allrouters
```

Here, the first entry in the /etc/hosts file is the source container's IP address (172.17.0.3) and its hostname (a02895551686).

3. We will continue to display the environment variables of the source container using the env command:

```
     / # env
     HOSTNAME=a02895551686
     SHLVL=1
     HOME=/root
     TERM=xterm
     PATH=
     /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
     PWD=/
```

4. We have now launched the source container. From another Terminal of the same Docker host, let's launch the interactive recipient container by linking it to our source container using the --link option of the docker run subcommand, as shown here:

```
$ sudo docker run --rm --link example:ex \
    -it busybox:latest
```

Here, the source container named example is linked to the recipient container with ex as its alias.

5. Let's display the content of the /etc/hosts file of the recipient container using the cat command:

```
/ # cat /etc/hosts
172.17.0.4      a17e5578b98e
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.3      ex
```

Of course, as always, the first entry in the /etc/hosts file is the IP address of the container and its hostname. However, the noteworthy entry in the /etc/hosts file is the last entry, where the IP address (172.17.0.3) of the source container and its alias (ex) are added automatically.

6. We will continue to display the recipient container's environment variable using the env command:

```
/ # env
HOSTNAME=a17e5578b98e
SHLVL=1
HOME=/root
EX_NAME=/berserk_mcclintock/ex
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
```

Apparently, a new EX_NAME environment variable is added automatically to /berserk_mcclintock/ex, as its value. Here EX is the capitalized form of the alias ex and berserk_mcclintock is the autogenerated name of the recipient container.

7. As a final step, ping the source container using the widely used ping command for two counts and use the alias name as the ping address:

```
/ # ping -c 2 ex
PING ex (172.17.0.3): 56 data bytes
64 bytes from 172.17.0.3: seq=0 ttl=64
time=0.108 ms
64 bytes from 172.17.0.3: seq=1 ttl=64
time=0.079 ms

--- ex ping statistics ---
2 packets transmitted, 2 packets received,
0% packet loss
round-trip min/avg/max = 0.079/0.093/0.108 ms
```

Evidently, the alias ex of the source container is resolved to the 172.17.0.3 IP address, and the recipient container is able to successfully reach the source. In the case of secured container communication, pinging between containers is not allowed.

In the preceding example, we can link two containers together, and also, observe how elegantly networking is enabled between the containers by updating the IP address of the source container in the /etc/hosts file of the recipient container.

The next example is to demonstrate how container linking exports the environment variables of the source container, which are configured using the -e option of the docker run subcommand or the ENV instruction of Dockerfile, to the recipient container. For this purpose, we are going to craft a file named Dockerfile with the ENV instruction, build an image, launch a source container using this image, and then launch a recipient container by linking it to the source container:

1. We begin with composing a Dockerfile with the ENV instruction, as shown here:

```
FROM busybox:latest
ENV BOOK="Docker"  \
   CHAPTER="Linking Containers"
```

Here, we are setting up two environment variables, BOOK and CHAPTER.

2. Proceed to build a Docker image envex using the docker build subcommand from the preceding Dockerfile:

```
$ sudo docker build -t envex .
```

3. Now, let's launch an interactive source container with the example name using the envex image we just built:

```
$ sudo docker run -it --rm \
            --name example envex
```

4. From the source container prompt, display all the environment variables by invoking the env command:

```
/ # env
HOSTNAME=b53bc036725c
SHLVL=1
HOME=/root
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
BOOK=Docker
CHAPTER=Linking Containers
PWD=/
```

In all the preceding environment variables, both the BOOK and the CHAPTER variables are configured with the ENVinstruction of the Dockerfile.

5. As a final step, to illustrate the ENV category of environment variables, launch the recipient container with the envcommand, as shown here:

```
$ sudo docker run --rm --link example:ex \
            busybox:latest env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=a5e0c07fd643
TERM=xterm
EX_NAME=/stoic_hawking/ex
EX_ENV_BOOK=Learning Docker
```

```
EX_ENV_CHAPTER=Orchestrating Containers
HOME=/root
```

Strikingly, in the preceding output, the variables that are prefixed with EX_ are the outcome of container linking. The environment variables of our interest are EX_ENV_BOOK and EX_ENV_CHAPTER, which were originally set through the Dockerfile as BOOK and CHAPTER but modified to EX_ENV_BOOK and EX_ENV_CHAPTER, as an effect of container linking. Though the environment variable names get translated, the values stored in these environment variables are preserved as is. We already discussed the EX_NAME variable name in the previous example.

In the preceding example, we experienced how elegantly and effortlessly Docker exports the ENV category variables from the source container to the recipient container. These environment variables are completely decoupled from the source and the recipient, thus a change in the value of these environment variables in one container does not impact the other. To be even more precise, the values the recipient container receives are the values set during the launch of the source container. Any changes made to the value of these environment variables in the source container after its launch have no effect on the recipient container. It does not matter when the recipient container is launched because the values are being read from the JSON file.

In our final illustration of linking containers, we are going to show you how to take advantage of the Docker feature to share the connectivity details between two containers. In order to share the connectivity details between containers, Docker uses the PORT category of environment variables. The following are the steps used to craft two containers and share the connectivity details between them:

1. Craft a Dockerfile to expose port 80 and 8080 using the EXPOSE instruction, as shown here:

```
FROM busybox:latest
EXPOSE 8080 80
```

2. Proceed to build a portex Docker image using the docker build subcommand from the Dockerfile, we created just now, by running the following command:

```
$ sudo docker build -t portex .
```

3. Now, let's launch an interactive source container with the example name using the earlier built portex image:

```
$ sudo docker run -it --rm --name example portex
```

4. Now that we have launched the source container, let's continue to create a recipient container on another Terminal by linking it to the source container, and invoke the env command to display all the environment variables, as shown here:

```
$ sudo docker run --rm --link example:ex \
            busybox:latest env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=c378bb55e69c
TERM=xterm
EX_PORT=tcp://172.17.0.4:80
EX_PORT_80_TCP=tcp://172.17.0.4:80
EX_PORT_80_TCP_ADDR=172.17.0.4
EX_PORT_80_TCP_PORT=80
EX_PORT_80_TCP_PROTO=tcp
EX_PORT_8080_TCP=tcp://172.17.0.4:8080
EX_PORT_8080_TCP_ADDR=172.17.0.4
EX_PORT_8080_TCP_PORT=8080
EX_PORT_8080_TCP_PROTO=tcp
EX_NAME=/prickly_rosalind/ex
HOME=/root
```

From the preceding output of the env command, it is quite evident that the Docker Engine exported a bunch of four PORT category environment variables for each port that was exposed using the EXPOSE instruction in the Dockerfile. In addition, Docker also exported another PORT category variable EX_PORT.

Orchestration of containers

Orchestrating containers using docker-compose

In this section, we will discuss the widely used container orchestration tool docker-compose. The docker-compose tool is a very simple, yet power tool and has been conceived and concretized to facilitate the running of a group of Docker containers. In other words, docker-compose is an orchestration framework that lets you define and control a multi-container service. It enables you to create a fast and isolated development environment as well as orchestrating multiple Docker containers in production. The docker-compose tool internally leverages the Docker Engine for pulling images, building the images, starting the containers in the correct sequence, and making the right connectivity/linking among the containers/services based on the definition given in the docker-compose.yml file.

Installing docker-compose

Use the wget tool like this:

```
sudo sh -c 'wget -qO- \
  https://github.com/docker/compose/releases/tag/1.11.2/ \
  docker-compose-`uname -s`-`uname -m` > \
  /usr/local/bin/docker-compose; \
  chmod +x /usr/local/bin/docker-compose'
```

Use the curl tool like this:

```
curl -L https://github.com/docker/compose/releases/download/1.11.2/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
chmod +x /usr/local/bin/docker-compose
```

The docker-compose tool is also available as a Python package, which you can install using the pip installer, as shown here:

```
$ sudo pip install -U docker-compose
```

Having successfully installed docker-compose, you can now check the docker-compose version:

The docker-compose file

The docker-compose tool orchestrates containers using **YAML**, which is a **Yet Another Markup Language** called the docker-compose file. YAML is a human-friendly data serialization format. Docker began its journey as a container enablement tool, and it is growing by leaps and bounds as an ecosystem to automate and accelerate most of the tasks such as container provisioning, networking, storage, management, orchestration, security, governance, and persistence. Consequently, the docker-compose file format and its version are revised multiple times to keep up with the Docker platform. At the time of writing this edition, the latest version of the docker-compose file is version 3. The following table lists the docker-compose file and the Docker Engine version compatibility matrix:

Docker Compose file format	Docker Engine	Remarks
3, 3.1	1.13.0+	Provides support for docker stack deploy and docker secrets
2.1	1.12.0+	Introduced a few new parameters
2	1.10.0+	Introduced support for named volumes and networks
1	1.9.0+	Will be deprecated in the future

		compose releases

The docker-compose tool by default uses a file named as docker-compose.yml or docker-compose.yaml to orchestrate containers. This default file can be modified using the -f option of the docker-compose tool. The following is the format of the docker-compose file:

```
version: "<version>"
services:
<service>:
<key>: <value>
<key>:
      - <value>
      - <value>
networks:
<network>:
<key>: <value>

volumes:
<volume>:
<key>: <value>
```

Here, the options used are as follows:

- <version>: This is the version of the docker-compose file. Refer to the preceding version table.
- <service>: This is the name of the service. You can have more than one service definition in a single docker-composefile. The service name should be followed by one or more keys. However, all the services must either have an image or a build key, followed by any number of optional keys. Except for the image and build keys, the rest of the keys can be directly mapped to the options in the docker run subcommand. The value can be either a single value or multiple values. All the <service> definitions must be grouped under the top-level services key.
- <network>: This is the name of the networks that are used by the services. All the <network> definitions must be grouped under the top-level networks key.
- <volume>: This is the name of the volume that is used by the services. All the <volume> definitions must be grouped under the top-level volume key.

Here, we are listing a few keys supported in the docker-compose file version 3. Refer to https://docs.docker.com/compose/compose-file for all the keys supported by docker-compose.

- image: This is the tag or image ID.
- build: This is the path to a directory containing a Dockerfile.
- command: This key overrides the default command.
- deploy: This key has many subkeys and is used to specify deployment configuration. This is used only in the docker swarm mode.
- depends_on: This is used to specify the dependencies between services. It can be further extended to chain services based on their conditions.
- cap_add: This adds a capability to the container.
- cap_drop: This drops a capability of the container.
- dns: This sets custom DNS servers.
- dns_search: This sets custom DNS search servers.
- entrypoint: This key overrides the default entrypoint.
- env_file: This key lets you add environment variables through files.
- environment: This adds environment variables and uses either an array or a dictionary.
- expose: This key exposes ports without publishing them to the host machine.
- extends: This extends another service defined in the same or a different configuration file.
- extra_hosts: This enables you to add additional hosts to /etc/hosts inside the container.
- healthcheck: This allows us to configure the service health check.
- labels: This key lets you add metadata to your container.
- links: This key links to containers in another service. Usage of links is strongly discouraged.
- logging: This is used to configure the logging for the service.
- network: This is used to join the service to the network defined in the top-level networks key.
- pid: This enables the PID space sharing between the host and the containers.
- ports: This key exposes ports and specifies both the HOST_port:CONTAINER_port ports.

- volumes: This key mounts path or named volumes. The named volumes need to be defined in the top-level volumes key.

The docker-compose command

The docker-compose tool provides sophisticated orchestration functionality with a handful of commands. In this section, we will list out the docker-compose options and commands:

```
docker-compose [<options>] <command> [<args>...]
```

The docker-compose tool supports the following options:

- -f, --file <file>: This specifies an alternate file for docker-compose (default is the docker-compose.yml file)
- -p, --project-name <name>: This specifies an alternate project name (default is the directory name)
- --verbose: This shows more output
- -v, --version: This prints the version and exits
- -H, --host <host>: This is to specify the daemon socket to connect to
- -tls, --tlscacert, --tlskey, and --skip-hostname-check: The docker-compose tool also supports these flags for **Transport Layer Security** (**TLS**)

The docker-compose tool supports the following commands:

- build: This command builds or rebuilds services.
- bundle: This is used to create a Docker bundle from the compose file, this is still an experimental feature on Docker 1.13.
- config: This is a command to validate and display the compose file.
- create: This creates the services defined in the compose file.
- down: This command is used to stop and remove containers and networks.
- events: This can be used to view the real-time container life cycle events.
- exec: This enables you to run a command in a running container. It is used predominantly for debugging purposes.
- kill: This command kills running containers.

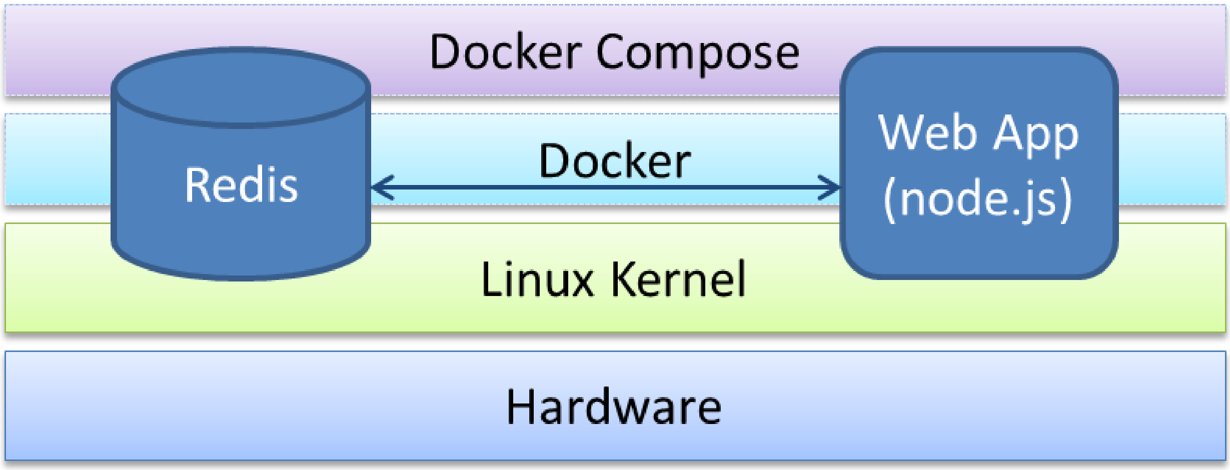
- logs: This displays the output from the containers.
- pause: This command is used to pause services.
- port: This prints the public port for a port binding.
- ps: This lists the containers.
- pull: This command pulls the images from the repository.
- push: This command pushes the images to the repository.
- restart: This is used to restart the services defined in the compose file.
- rm: This removes the stopped containers.
- run: This runs a one-off command.
- scale: This sets a number of containers for a service.
- start: This command starts services defined in the compose file.
- stop: This stops services.
- unpause: This command is used to unpause services.
- up: This creates and starts containers.
- version: This prints the version of Docker Compose.

Common usage

In this section, we are going to experience the power of the orchestration feature provided by the Docker Compose framework with the help of an example. For this purpose, we are going to build a two-tiered web application that will receive your inputs through a URL and respond with the associated response text. This application is built using the following two services, as enumerated here:

- **Redis**: This is a key-value database used to store a key and its associated value
- **Node.js**: This is a JavaScript runtime environment used to implement the web server functionality as well the application logic

Each of these services is packed inside two different containers that are stitched together using the docker-compose tool. The following is the architectural representation of the services:



Here, in this example, we begin with implementing the example.js module, a Node.js file to realize the web server, and the key lookup functionality. Further, we will craft the Dockerfile on the same directory as example.js to package the Node.js runtime environment, and then, define the service orchestration using a docker-compose.yml file in the same directory as example.js.

The following is the example.js file, which is a Node.js implementation of the simple request/response web application. For demonstration, in this sample code, we restrict the request and response for just two docker-compose commands (build and kill). For the code to be self-explanatory, we added comments in the code:

```
// A Simple Request/Response web application

// Load all required libraries
var http = require('http');
var url = require('url');
var redis = require('redis');

// Connect to redis server running
// createClient API is called with
//  -- 6379, a well-known port to which the
//           redis server listens to
//  -- redis, is the name of the service (container)
//           that runs redis server
var client = redis.createClient(6379, 'redis');
```

```
// Set the key value pair in the redis server

// Here all the keys proceeds with "/", because
// URL parser always have "/" as its first character
client.set("/", "Welcome to Docker-Compose helpernEnter the docker-compose command in the URL for helpn", redis.print);
client.set("/build", "Build or rebuild services", redis.print);
client.set("/kill", "Kill containers", redis.print);

var server = http.createServer(function (request, response) {
  var href = url.parse(request.url, true).href;
  response.writeHead(200, {"Content-Type": "text/plain"});

  // Pull the response (value) string using the URL
  client.get(href, function (err, reply) {
    if ( reply == null ) response.write("Command: " +
    href.slice(1) + " not supportedn");
    else response.write(reply + "n");
    response.end();
  });
});

console.log("Listening on port 80");
server.listen(80);
```

The following text is the content of Dockerfile that packs the Node.js image, the redis driver for Node.js, and the example.js file, as defined earlier:

```
###############################################
# Dockerfile to build a sample web application
###############################################

# Base image is node.js
FROM node:latest

# Author: Dr. Peter
MAINTAINER Dr. Peter <peterindia@gmail.com>

# Install redis driver for node.js
RUN npm install redis

# Copy the source code to the Docker image
ADD example.js /myapp/example.js
```

The following text is from the docker-compose.yml file that defines the services that the Docker Compose tool orchestrates:

```
version: "3.1"
services:
  web:
    build: .
    command: node /myapp/example.js
    depends_on:
       - redis
    ports:
    - 8080:80
  redis:
    image: redis:latest
```

We defined two services in this docker-compose.yml file, wherein these services serve the following purposes:

- The service named web is built using the Dockerfile in the current directory. Also, it is instructed that you launch the container by running the node (the Node.js runtime) with /myapp/example.js (web application implementation), as its argument. Since this Node.js application uses the redis database, the web service is forced to start after the redisservice using the depends_on instruction. Besides, the 80 container port is mapped to the 8080 Docker host's port.
- The service named redis is instructed to launch a container with the redis:latest image. If the image is not present in the Docker host, the Docker Engine will pull it from the central repository or the private repository.

Now, let's continue with our example by building the Docker images using the docker-compose build command, launch the containers using the docker-compose up command, and connect with a browser to verify the request/response functionality, as explained step by step here:

1. The docker-compose commands must be executed from the directory in which the docker-compose.yml file is stored. Besides, docker-compose considers each docker-compose.yml file as a project, and it assumes the project name from the docker-compose.yml file's directory. Of course, this can be overridden using the -p option. So, as a first step, let's change the directory, wherein the docker-compose.yml file is stored:

```
$ cd ~/example
```

2. Build the services using the docker-compose build command:

```
$ sudo docker-compose build
```

3. Pull the images from the repository using the docker-compose pull command:

```
$ sudo docker-compose pull
```

4. Proceed to bring up the services as indicated in the docker-compose.yml file using the docker-compose up command:

```
$ sudo docker-compose up
Creating network "example_default" with the default
driver
Creating example_redis_1
Creating example_web_1
Attaching to example_redis_1, example_web_1
redis_1 | 1:C 03 Feb 18:09:40.743 # Warning: no
config file specified, using the default config.
In order to specify a config file use redis-server
/path/to/redis.conf
. . . TRUNCATED OUTPUT . . .
redis_1 | 1:M 03 Feb 18:03:47.438 * The server
is now ready to accept connections on port 6379
web_1    | Listening on port 80
web_1    | Reply: OK
web_1    | Reply: OK
web_1    | Reply: OK
```

Since the directory name is example, the docker-compose tool has assumed that the project name is example. If you pay attention to the first line of the output, you will notice the example_default network being created. The Docker Compose tool creates this bridge network by default and this network is used by the service for IP address resolution. Thus the services can reach the other services by just using the service names defined in the compose file.

5. Having successfully orchestrated the services using the docker-compose tool, let's invoke the docker-compose pscommand from a different Terminal to list the containers associated with the example docker-compose project:

```
$ sudo docker-compose ps
   Name                  Command
State                Ports
--------------------------------------------------
-------------------------
   example_redis_1        /entrypoint.sh redis-server
Up                    6379/tcp
   example_web_1 node      /myapp/example.js
Up                    0.0.0.0:8080->80/tcp
```

Evidently, the two example_redis_1 and example_web_1 containers are up and running. The container name is prefixed with example_, which is the docker-compose project name.

6. Explore the functionality of our own request/response web application on a different Terminal of the Docker host, as illustrated here:

```
$ curl http://localhost:8080
Welcome to Docker-Compose helper
Enter the docker-compose command in the URL for help
$ curl http://localhost:8080/build
Build or rebuild services
$ curl http://localhost:8080/something
Command: something not supported
```

Debugging Containers

Debugging has been an artistic component in the field of software engineering. All kinds of software building blocks individually, as well as collectively, need to go through a stream of deeper and decisive investigations by software development and testing professionals to ensure the security and safety of the resulting software applications. As Docker containers are said to be key runtime environments for next generation mission-critical software workloads, it is pertinent and paramount for containers, crafters, and composers to embark on a systematic and sagacious verification and validation of containers.

This chapter has been dedicatedly written to enable technical guys who have all the accurate and relevant information to meticulously debug both the applications running inside containers and the containers themselves. In this chapter, we will also look at the theoretical aspects of process isolation for processes running as containers. A Docker container runs at a user-level process on host machines and typically has the same isolation level as provided by the operating system. With the latest Docker releases, many debugging tools are available which can be efficiently used to debug your applications. We will also cover the primary Docker debugging tools, such as `docker exec`, `stats`, `ps`, `top`, `events`, and `logs`. The current version of Docker is written in Go and it takes advantage of several features of the Linux kernel to deliver its functionality.

After installing the Docker Engine on your host machine, the Docker daemon can be started with the `-D` debug option:

This `-D` debug flag can be enabled to the Docker configuration file (`/etc/default/docker`) also in the debug mode:

```
DOCKER_OPTS="-D"
```

After saving and closing the configuration file, restart the Docker daemon.

Debugging a containerized application

Computer programs (software) sometimes fail to behave as expected. This is due to faulty code or due to the environmental changes between the development, testing, and deployment systems. Docker container technology eliminates the environmental issues between development, testing, and deployment as much as possible by containerizing all the application dependencies. Nonetheless, there could be still anomalies due to faulty code or variations in the kernel behavior, which needs debugging. Debugging is one of the most complex processes in the software engineering world and it becomes much more complex in the container paradigm because of the isolation techniques. In this section, we are going to learn a few tips and tricks to debug a containerized application using the tools native to Docker, as well as the tools provided by external sources.

Initially, many people in the Docker community individually developed their own debugging tools, but later Docker started supporting native tools, such as exec, top, logs, and events. In this section, we will dive deep into the following Docker tools:

- exec
- ps
- top
- stats
- events
- logs
- attach

The docker exec command

The docker exec command provides the much-needed help to users, who are deploying their own web servers or have other applications running in the background. Now, it is not necessary to log in to run the SSH daemon in the container.

1. First, create a Docker container:

```
$ sudo docker run --name trainingapp \
training/webapp:latest
Unable to find image
'training/webapp:latest' locally
latest: Pulling from training/webapp
9dd97ef58ce9: Pull complete
a4c1b0cb7af7: Pull complete
Digest:
sha256:06e9c1983bd6d5db5fba376ccd63bfa529e8d02f23d5079b8f74a616308fb11d
```

```
Status: Downloaded newer image for
training/webapp:latest
```

2. Next, run the docker ps -a command to get the container ID:

```
$ sudo docker ps -a
    a245253db38b      training/webapp:latest
    "python app.py"
```

3. Then, run the docker exec command to log in to the container:

```
$ sudo docker exec -it a245253db38b bash
root@a245253db38b:/opt/webapp#
```

4. Note that the docker exec command can only access the running containers, so if the container stops functioning, then you need to restart the stopped container in order to proceed. The docker exec command spawns a new process in the target container using the Docker API and CLI. So if you run the ps -aef command inside the target container, it looks like this:

```
# ps -aef
UID       PID  PPID  C STIME  TTY     TIME
CMD
root      1     0    0 Nov 26  ?    00:00:53
python app.py
root      45    0    0  18:11  ?    00:00:00
bash
root      53    45   0 18:11   ?    00:00:00
ps -aef
```

Here, python app.y is the application that is already running in the target container, and the docker exec command has added the bash process inside the container. If you run kill -9 pid(45), you will be automatically logged out of the container.

The docker ps command

The docker ps command, which is available inside the container, is used to see the status of the process. This is similar to the standard ps command in the Linux environment and is **not** a docker ps command that we run on the Docker host machine.

This command runs inside the Docker container:

```
root@5562f2f29417:/# ps -s
UID   PID   PENDING   BLOCKED   IGNORED   CAUGHT STAT TTY        TIME
COMMAND
0     1  00000000  00010000  00380004  4b817efb Ss
?        0:00 /bin/bash
0    33  00000000  00000000  00000000  73d3fef9 R+   ?        0:00 ps -s
root@5562f2f29417:/# ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY        TIME CMD
4 S     0     1     0  0  80   0 -  4541 wait   ?      00:00:00  bash
root@5562f2f29417:/# ps -t
PID TTY      STAT   TIME COMMAND
  1 ?        Ss     0:00 /bin/bash
 35 ?        R+     0:00 ps -t
root@5562f2f29417:/# ps -m
PID TTY         TIME CMD
  1 ?       00:00:00 bash
  - -       00:00:00 -
 36 ?       00:00:00 ps
  - -       00:00:00 -
root@5562f2f29417:/# ps -a
```

Use ps --help <simple|list|output|threads|misc|all> or ps --help <s|l|o|t|m|a> for additional help text.

The docker top command

You can run the top command from the Docker host machine using the following command:

```
docker top [OPTIONS] CONTAINER [ps OPTIONS]
```

This gives a list of the running processes of a container without logging in to the container, as follows:

```
$ sudo docker top  a245253db38b
UID          PID         PPID         C
STIME        TTY         TIME         CMD
root         5232        3585         0
Mar22        ?           00:00:53     python  app.py
$ sudo docker top  a245253db38b  -aef
UID          PID         PPID         C
STIME        TTY         TIME         CMD
root         5232        3585         0
Mar22        ?           00:00:53     python  app.py
```

The Docker top command provides information about the CPU, memory, and swap usage if you run it inside a Docker container:

```
root@a245253db38b:/opt/webapp# top
top - 19:35:03 up 25 days, 15:50,  0 users,  load average: 0.00,  0.01, 0.05
Tasks:   3 total,   1 running,   2 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0%us,  0.0%sy,  0.0%ni, 99.9%id,  0.0%wa,  0.0%hi,   0.0%si,  0.0%st
Mem:   1016292k total,   789812k used,   226480k free,    83280k  buffers
Swap:       0k total,       0k used,       0k free,   521972k  cached
PID USER     PR  NI  VIRT  RES  SHR S %CPU %MEM
TIME+  COMMAND
  1 root     20   0 44780  10m 1280 S  0.0  1.1   0:53.69 python
  62 root     20  0 18040 1944 1492 S  0.0  0.2   0:00.01 bash
  77 root     20  0 17208 1164  948 R  0.0  0.1   0:00.00 top
```

In case you get the error - TERM environment variable not set error while running the top command inside the container, perform the following steps to resolve it:

Run the echo $TERM command. You will get the result as dumb. Then, run the following command:

```
$ export TERM=dumb
```

The docker stats command

The docker stats command provides you with the capability to view the memory, CPU, and the network usage of a container from a Docker host machine, as illustrated here:

```
$ sudo docker stats a245253db38b
CONTAINER        CPU %       MEM USAGE/LIMIT      MEM %   NET I/O
a245253db38b      0.02%       16.37 MiB/992.5 MiB   1.65%
3.818   KiB/2.43 KiB
```

You can run the stats command to also view the usage for multiple containers:

```
$ sudo docker stats a245253db38b f71b26cee2f1
```

Docker provides access to container statistics **read only** parameters. This streamlines the CPU, memory, network IO, and block IO of containers. This helps you choose the resource limits and also in profiling. The Docker stats utility provides you with these resource usage details only for running containers.

The Docker events command

Docker containers will report the following real-time events: create, destroy, die, export, kill, omm, pause, restart, start, stop, and unpause. The following are a few examples that illustrate how to use these commands:

```
$ sudo docker pause  a245253db38b
a245253db38b

$ sudo docker ps -a
a245253db38b   training/webapp:latest    "python app.py"
4 days ago     Up 4 days (Paused)      0.0.0.0:5000->5000/tcp sad_sammet

$ sudo docker unpause  a245253db38b
a245253db38b
```

```
$ sudo docker ps -a
a245253db38b      training/webapp:latest    "python app.py"
4 days ago    Up 4 days      0.0.0.0:5000->5000/tcpsad_sammet
```

The Docker image will also report the untag and delete events.

The usage of multiple filters will be handled as an AND operation; for example,

`--filter container= a245253db38b --filter event=start` will display events for the container `a245253db38b` and the event type is `start`.

The docker logs command

This command fetches the log of a container without logging in to the container. It batch-retrieves logs present at the time of execution. These logs are the output of stdout and stderr. The general usage is shown in `docker logs [OPTIONS] CONTAINER`.

The `-follow` option will continue to provide the output till the end, `-t` will provide the timestamp, and `--tail= <number of lines>` will show the number of lines of the log messages of your container:

```
$ sudo docker logs a245253db38b
* Running on http://0.0.0.0:5000/
172.17.42.1 - - [22/Mar/2015 06:04:23] "GET / HTTP/1.1" 200 -
172.17.42.1 - - [24/Mar/2015 13:43:32] "GET / HTTP/1.1" 200 -

$ sudo docker logs -t a245253db38b
2015-03-22T05:03:16.866547111Z  * Running on http://0.0.0.0:5000/
2015-03-22T06:04:23.349691099Z  172.17.42.1 - - [22/Mar/2015 06:04:23] "GET /
HTTP/1.1" 200 -
2015-03-24T13:43:32.754295010Z  172.17.42.1 - - [24/Mar/2015 13:43:32] "GET /
HTTP/1.1" 200 -
```

The docker attach command

The docker attach command attaches the running container and it is very helpful when you want to see what is written in stdout in real time:

```
$ sudo docker run -d --name=newtest alpine /bin/sh -c "while true; do sleep 2; df -h; done"
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
3690ec4760f9: Pull complete
Digest: sha256:1354db23ff5478120c980eca1611a51c9f2b88b61f24283ee8200bf9a54f2e5c
1825927d488bef7328a26556cfd72a54adeb3dd7deafb35e317de31e60c25d67
$ sudo docker attach newtest
Filesystem           Size    Used Available Use% Mounted on
none                 7.7G    3.2G    4.1G  44% /
tmpfs              496.2M       0  496.2M   0% /dev
tmpfs              496.2M       0  496.2M   0% /sys/fs/cgroup
/dev/xvda1           7.7G    3.2G    4.1G  44% /etc/resolv.conf
/dev/xvda1           7.7G    3.2G    4.1G  44% /etc/hostname
/dev/xvda1           7.7G    3.2G    4.1G  44% /etc/hosts
shm                 64.0M       0   64.0M   0% /dev/shm
tmpfs              496.2M       0  496.2M   0% /proc/sched_debug
Filesystem           Size    Used Available Use% Mounted on
none                 7.7G    3.2G    4.1G  44% /
tmpfs              496.2M       0  496.2M   0% /dev
```

By default, this command attaches stdin and proxies signals to the remote process. Options are available to control both of these behaviors. To detach from the process, use the default **Ctrl** + **C** sequence.

Debugging a Dockerfile

Sometimes creating a Dockerfile may not start with everything working. A Dockerfile does not always build images and sometimes it does, but starting a container would crash on startup.

Every instruction we set in the Dockerfile is going to be built as a separate, temporary image for the other instruction to build itself on top of the previous instruction. The following example explains this:

1. Create a Dockerfile using your favorite editor:

```
FROM busybox
RUN ls -lh
CMD echo Hello world
```

2. Now, build the image by executing the following command:

```
$ docker build .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM busybox
latest: Pulling from library/busybox
56bec22e3559: Pull complete
Digest:
sha256:29f5d56d12684887bdfa50dcd29fc31eea4aaf4ad3bec43daf19026a7ce69912
Status: Downloaded newer image for busybox:latest
---> e02e811dd08f
Step 2 : RUN ls -lh
---> Running in 7b47d3c46cfa
total 36
drwxr-xr-x    2 root     root       12.0K Oct  7 18:18 bin
dr-xr-xr-x  130 root     root           0 Nov 27 01:36 proc
drwxr-xr-x    2 root     root        4.0K Oct  7 18:18 root
dr-xr-xr-x   13 root     root           0 Nov 27 01:36 sys
drwxrwxrwt    2 root     root        4.0K Oct  7 18:18 tmp
---> ca5bea5887d6
Removing intermediate container 7b47d3c46cfa
Step 3 : CMD echo Hello world
---> Running in 490ecc3d10a9
---> 490d1c3eb782
Removing intermediate container 490ecc3d10a9
Successfully built 490d1c3eb782
```

Notice the ---> Running in 7b47d3c46cfa line. 7b47d3c46cfa is a valid image and can be used to retry the failed instruction and see what's happening

To debug this image, we need to create a container and then log in to analyze the error. Debugging is a process of analyzing what's going on and it's different for every situation, but usually, the way we start debugging is by trying to manually make the instruction that fails work manually and understand the error. When I get the instruction to work, I usually exit the container, update my Dockerfile, and repeat the process until I have something working.

The Docker use cases

Containerization is emerging as the way forward for the software industry as it brings forth a newer and richer way of building and bundling any kind of software, shipping and running them everywhere. That is the fast-evolving aspect of containerization that promises and provides software portability, which has been a constant nuisance for IT developers and administrators for many decades now. The Docker idea is flourishing here because of a number of enabling factors and facets. This section is specially prepared for specifying the key use cases of the Docker idea.

Integrating containers into workflows

Workflows are a widely accepted and used abstraction for unambiguously representing the right details of any complicated and large-scale business and scientific applications and executing them on distributed computing systems such as clusters, clouds, and grids. However, workflow management systems have been largely evasive in conveying the relevant information of the underlying environment on which the tasks inscribed in the workflow are to run. This means that the workflow tasks can run perfectly on the environment for which they were designed. The real challenge is to run the tasks across multiple IT environments without tweaking and twisting the source codes of the required tasks. Increasingly, the IT environments are heterogeneous with the leverage of disparate operating systems, middleware, programming languages and frameworks, databases, and so on. Typically, workflow systems focus on data interchange between tasks and are environment-specific. A workflow, which is working fine in one environment, starts to crumble when it is being migrated and deployed on different IT environments. All kinds of known and unknown dependencies and incompatibilities spring up to denigrate the workflows delaying the whole job of IT setup, application installation and configuration, deployment, and delivery. Containers are the best bet for resolving this imbroglio once and for all.

In the article, **Integrating Containers into Workflows: A Case Study Using Makeflow, Work Queue, and Docker,** **Chao Zheng**and **Douglas Thain** have done a good job of analyzing several methods in order to experimentally prove the unique contributions of containers in empowering workflow/process management systems. They have explored the performance of a large bioinformatics workload on a Docker-enabled cluster and observed the best configuration to be locally managed on containers that are shared between multiple tasks.

Docker for HPC and TC applications

According to Douglas M. Jacobsen and Richard Shane Canon, currently, containers are being overwhelmingly used for the web, enterprise, mobile, and cloud applications. However, there are questions being asked and doubts being raised on whether containers can be a viable runtime for hosting technical and scientific computing applications. Especially, there are many **High-Performance Computing** (**HPC**) applications yearning for a perfect deployment and execution environment. The authors of this research paper have realized that Docker containers can be a perfect answer for HPC workloads.

In many cases, users desire to have the ability to easily execute their scientific applications and workflows in the same environment used for development or adopted by their community. Some researchers have tried out the cloud option, but the challenges are many. The users need to solve how they handle workload management, filesystems, and basic provisioning. Containers promise to offer the flexibility of cloud-type systems coupled with the performance of bare-metal systems. Furthermore, containers have the potential to be more easily integrated into traditional HPC environments, which means that users can obtain the benefits of flexibility without the added burden of managing other layers of the system (that is, batch systems, filesystems, and so on).

Minh Thanh Chung and the team have analyzed the performance of VMs and containers for high-performance applications and benchmarked the results that clearly show containers are the next-generation runtime for HPC applications. In short, Docker offers many attractive benefits in an HPC environment. To test these, IBM Platform LSF and Docker have been integrated outside the core of Platform LSF and the integration leverages the rich Platform LSF plugin framework.

We all know that the aspect of compartmentalization is for resource partitioning and provisioning. This means that physical machines are subdivided into multiple logical machines (VMs and containers). Now on the reverse side, such kinds of logical systems carved out of multiple physical machines can be linked together to build a virtual supercomputer to solve certain complicated problems. **Hsi-En Yu** and **Weicheng Huang** have described how they built a virtual HPC cluster in the research paper, **Building a Virtual HPC Cluster with Auto Scaling by the Docker.** They have integrated the autoscaling feature of service discovery with the lightweight virtualization paradigm (Docker) and embarked on the realization of a virtual cluster on top of physical cluster hardware.

Containers for telecom applications

Csaba Rotter and the team have explored and published a survey article with the title, **Using Linux Containers in Telecom Applications.** Telecom applications exhibit strong performance and high availability requirements; therefore, running them in containers requires additional investigations**.** A telecom application is a single or multiple node application responsible for a well-defined task. Telecom applications use standardized interfaces to connect to other network elements and implement standardized functions. On top of the standardized functions, a telecom application can have vendor-specific functionality. There is a set of QoS and **Quality of Experience** (**QoE**) attributes such as high availability, capacity, and performance/throughput. The paper has clearly laid out the reasons for the unique contributions of containers in having next-generation telecom applications.

Efficient Prototyping of Fault Tolerant Map-Reduce Applications with Docker-Hadoop by **Javier Rey and the team** advocated that distributed computing is the way forward for compute and data-intensive workloads. There are two major trends. Data becomes big and there are realizations that big data leads to big insights through the leverage of pioneering algorithms, scripts, and parallel languages such as Scala, integrated platforms, new-generation databases, and dynamic IT infrastructures. MapReduce is a parallel programming paradigm currently used to perform computations on massive amounts of data. Docker-Hadoop1 is a virtualization testbed conceived to allow the rapid deployment of a Hadoop cluster. With Docker-Hadoop, it is possible to control the characteristics of the node and run scalability and performance tests that otherwise would require a large computing environment. Docker-Hadoop facilitates simulation and reproduction of different failure scenarios for the validation of an application.

Regarding interactive social media applications, Alin Calinciuc and the team have come out with a research publication titled as **OpenStack and Docker: Building a high-performance IaaS platform for interactive social media applications**. It is a well-known truth that interactive social media applications face the challenge of efficiently provisioning new resources in order to meet the demands of the growing number of application users. The authors have given the necessary description on how Docker can run as a hypervisor, and how the authors can manage to enable the fast provisioning of computing resources inside of an OpenStack IaaS using the nova-docker plugin that they had developed