Community Experience Distilled

# What you need to know about Docker

## The absolute essentials you need to get Docker up and running

Scott Gallagher

[PACKT] PUBLISHING

# What You Need to Know about Docker

The absolute essentials you need to get Docker up and running

**Scott Gallagher**

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

# What You Need to Know about Docker



First Published: May 2016

Production reference: 1190516

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

www.packtpub.com

# About the Author

**Scott Gallagher** has been fascinated with technology since he was in elementary school, when he used to play Oregon Trail. His love continued through middle school, working on more Apple IIe computers. In high school, he learned how to build computers and program in BASIC!. His college years were all about server technologies such as Novell, Microsoft, and Red Hat. After college, he continued to work on Novell, all while keeping an interest in technologies. He then moved on to managing Microsoft environments and eventually into what he is the most passionate about: Linux environments. Now, his focus is on Docker and cloud environments.

# About the Reviewer

**Harald Albers** works as a Java developer and security engineer in Hamburg, Germany. In addition to developing distributed web applications, he also sets up and maintains the build infrastructure and the staging and production environments for these applications.

Most of his work is only possible because of Docker's simple and elegant solutions for the challenges of provisioning, deployment, and orchestration. He started using Docker and contributing to the Docker project in mid 2014. He is a member of 2015/2016 Docker Governance Advisory Board.

# Table of Contents


**Meet Docker** 1

**A history of Docker** 1

What is containerization? 2

Docker differences 2

**Docker benefits** 3

Overall benefits 3

**Working with Containers** 5

**Using Docker images** 6

Searching Docker images 7

Manipulating Docker images 9

**Stopping containers** 11

**Other Docker Feature Sets** 13

**Storing images on Docker registries** 13

**Docker Machine** 14

**Docker Compose** 17

**Docker Swarm** 22

**Docker UCP** 23

**Creating Your Own Containers** 24

**Creating containers using Dockerfile** 24

Short Dockerfile review 24

Dockerfile in depth 25

LABEL 25

ADD or COPY 25

ENTRYPOINT 26

USER 26

WORKDIR 26

ONBUILD 26

Dockerfile best practices 27

**Docker build** **28**
The docker build command 28
The .dockerignore file 29
Modifying and committing an existing image 31
Building your own containers 32
Building using tar 32
Building using scratch 33
**Command Cheat Sheet** **34**
**Running containers** **34**
**Building containers** **35**
**Docker Hub commands** **36**
**Docker Swarm commands** **36**
**Docker Machine commands** **37**
**Docker Compose commands** **37**
**Summary** **38**
**What to do next?** **39**
**Broaden your horizons with Packt** **39**

# What you need to know about Docker

This eGuide is designed to act as a brief, practical introduction to Docker. It is full of practical examples which will get you up and running quickly with the core tasks of Docker.

We assume that you know a bit about what Docker is, what it does, and why you want to use it, so this eGuide won't give you a history lesson in the background of Docker. What this eGuide will give you, however, is a greater understanding of the key basics of Docker so that you have a good idea of how to advance after you've read the guide. We can then point you in the right direction of what to learn next after giving you the basic knowledge to do so.

*What You Need to Know about Docker* will do the following:

- Cover the fundamentals and the things you really need to know, rather than niche or specialized areas
- Assume that you come from a fairly technical background and so understand what the technology is and what it broadly does
- Focus on what things are and how they work
- Include practical examples to get you up, running, and productive quickly

# Overview

Docker is the hottest topic in technology these days and everybody is scrambling to learn about it; but where do you start? This small guide will help you get a better understanding of Docker and some of the common components surrounding Docker and give you insight on how to get caught up to speed.

Docker is being used by almost everybody these days, from developers and high education institutions to large corporations, and everybody is trying to get a handle on how to best utilize it. This guide will help you get a firm understanding of Docker, Docker Machine, Docker Compose, and Docker Swarm. It will also guide you on how to use containers, use the trusted images, create your own, manipulate images, and remove the ones you aren't using anymore. Learn what benefits you will gain by using Docker and how it compares to the typical virtual machine environments you are currently accustomed to.

The various registries to store your Docker images are also covered so you can make the right educated decision when the time comes. There is also a short command cheat sheet you can reference when you are learning commands or need to reference them quickly. You won't have to dig through documentation to find a particular command. These commands are focused on running and building containers, Docker registry commands, Docker Swarm, Compost, and Machine commands as well.

# Meet Docker

In the first chapter of this book, we will give you some background information about Docker and how it became such a huge success in such a short amount of time. We will also cover how it can benefit you as a developer and how Docker containers are different to the environments that you may currently use. How does Docker relate to containers anyway? We will cover this as well in this chapter. Lastly, we'll cover the benefits of Docker to you—the reader, and how it will help accelerate your development.

## The emergence of Docker

How did Docker come about and how did it become the latest buzzword in such a short amount of time? We all know that technology moves fast, but Docker has been moving and gathering interest at breakneck speed.

Docker began as an internal project for the dotCloud organization. It was developed in-house and then later open sourced in 2013. dotCloud was a **platform as a service** (**PaaS**) that allowed users to run applications without having to worry about the underlying infrastructure. They were spinning up servers or virtual machines more and more quickly, and they needed a way to spin up these environments faster. In order to further increase startup times, they began using containers, and Docker was born out of this need.

Its growth has been massive! Shortly after launching, Docker was being evaluated by over 10,000 developers. It had over 2.75 million users after their 1.0 launch out of beta in June of 2014, and this number has now grown to well over 100 million downloads. Docker has companies, such as RedHat and Amazon, adding support so that you can "link" into their environments to use Docker to manage your existing infrastructure there.

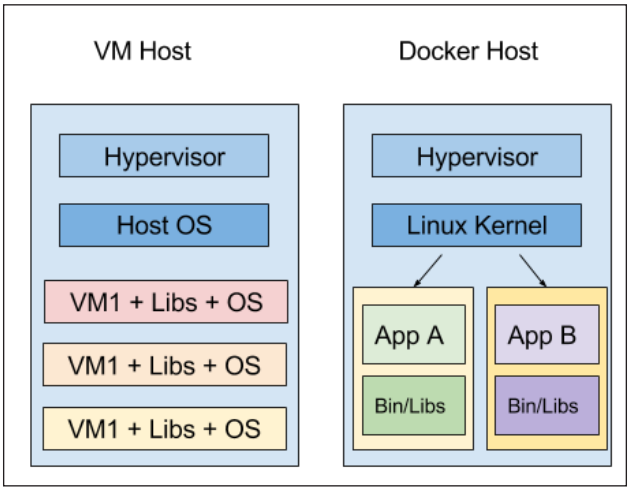
# What is containerization?

Docker utilizes Linux containers. So, what are Linux containers? Linux containers, commonly referred to as LXC, originated in August of 2008, and they rely on the Linux kernel **cgroups** functionality that originated in Linux kernel version 2.6.24. Linux containers themselves are an operating system virtualization method that you can utilize to run multiple isolated Linux systems on a single host. They all utilize the kernel version that is running on the host on which the containers are running. In the next section, we will take a look at the differences between a Linux container versus a typical virtual machine environment, such as Microsoft Hyper-V or VMware ESXi, which should help clarify what you may typically use and let you compare it to what a Linux container setup may look like.

# Docker differences

First, we must know what exactly Docker is and what it does. Docker is a container management system that helps manage containers in an easier and universal fashion. This lets you create containers in virtual environments (on Mac and Windows) on your laptop and run commands or operations against them. The actions you perform on the containers that you run in these environments locally on your own machine will be the same commands or operations that you run against them when they are running in your production environment. This helps with not having to do things differently when you go from a development environment, such as the one 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 an insight into the biggest key benefit of Docker. This is that there is no need for a full operating system every time we need to bring up a new container, which cuts down on the overall size and resource footprint of containers. Docker relies on using the host OS's Linux kernel (as almost all the versions of Linux use the standard kernel models) for the OS it was built on, such as RedHat, CentOS, Ubuntu, and so on. For this reason, you can have almost any Linux OS as your host operating system and be able to layer other OSes on top of the host. For example, in the earlier illustration, the host OS could be Ubuntu, and we could have RedHat 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 RedHat or Debian on the host. Thus, another benefit of Docker is the size of images when they are born. They do not contain the largest piece: the kernel or the operating system. This makes them incredibly small, compact, and easy to ship.

# Docker benefits

Docker provides a lot of benefits, though it may take some time to get used to switching over from using a typical virtual machine environment for development to reap the rewards of using Docker.

## Overall benefits

The easiest way to understand the benefits of Docker and all its pieces is to use bullet points, so let's jump right into them:

- **Portability**: If you have experienced having to move servers or environments from one type of infrastructure to another, then you know what a pain that can be. With Docker, you can easily ship your environments to all different kinds of infrastructure without having to worry about building up new virtual machines and tearing down the old ones.
- **Quick deployment/teardown**: With a single command, you can spin up new containers or tear down existing ones. Typically, if you try to clone a virtual machine or spin up a new one, you are looking at waiting for close to or over a few hours. With Docker, it will take a few minutes to achieve what you need.
- **Managing infrastructure-like code**: When it comes to upgrades, you can simply update your Dockerfile, which we will explain in the *Creating Your Own Containers* chapter, and then tear down the old one. This helps not only with updates, but it can also help with rollbacks as well.

- **Open source**: As all the code is open source, you can customize it to your heart's content. This allows not only for customization but to be able to submit pull requests, which are code additions that the Docker core team can approve. In turn, they make these pull requests available to anyone who downloads and installs Docker.
- **Consistency**: No more of the "well it works on my machine!" excuse. As everyone uses the same images to work, consistency is always guaranteed. You know that if they start up a container using the Dockerfile, the container will act the same in your environment as it will on others.

There are also many other benefits that Docker provides not only in a developer environment but also in an system administration environment, where you can use Docker to control things, such as clustered machine environments, or refer back to the rolling updates or rollbacks as well.

# Working with Containers

We will start with some common commands. Then, we'll take a peek at commands that are used for Docker images. We will then take a dive into commands that are used for containers.

The first command we will look at is one of the most useful commands in Docker and in any command-line utility you may use. This is the `help` command. This is run simply by executing the command, as follows:

```
$ docker --help
```

The preceding command will give you a full list of all the Docker commands at your disposal and a brief description of what each command does. For further help with a particular command, you can run the following command:

```
$ docker COMMAND --help
```

You will then receive additional information about using the command, such as options, arguments, and descriptions for the arguments.

You can also use the Docker `version` command to gather information about what version of Docker you are running:

```
$ docker version


Client:
 Version:      1.10.3
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   20f81dd
 Built:        Thu Mar 10 21:49:11 2016
 OS/Arch:      darwin/amd64
```

```
Server:
 Version:      1.10.3
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   20f81dd
 Built:        Thu Mar 10 21:49:11 2016
 OS/Arch:      linux/amd64
```

This is helpful when you want to see what version of the Docker daemon you may be running to see whether you need or want to upgrade.

# Using Docker images

Next, let's take a dive into images. Let's learn how to view which images you currently have that you can run, and let's also search for images on the Docker Hub. Finally, let's pull these images down to your environment so that you can run them. Let's first take a look at the `docker images` command. On running the command, we will get an output similar to the following output:

```
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
ubuntu              14.04               ab57dbafeeea        11 days ago         194.5 MB
ubuntu              trusty              6d4946999d4f        11 days ago         188.3 MB
ubuntu              latest              6d4946999d4f        11 days ago         188.3 MB
```

Your output will differ based upon whether you have any images already in your Docker environment or what images you do have. There are a few important pieces to understand from the output that you see. Let's go over the columns and what is contained in each of them. The first column that you see is the repository column. This column contains the name of the repository, as it exists on the Docker Hub. If you were to have a repository that was from some other user's account, it may show up, as follows:

```
REPOSITORY              TAG             IMAGE ID         CREATED          VIRTUAL SIZE
scottpgallagher/mysql   latest          57df9c7989a1     9 weeks ago      321.7 MB
```

The next column is the tag column. This will show you what tag the image has. As you can see in the preceding example, with the Ubuntu repository, there are tag names for the different images. These images contain different versions of the Ubuntu operating system. So if you wanted to specify a particular version of a repository in your Dockerfile, you could do this. This is useful because you are not always reliant on having to use the latest version of an operating system, and you can use the one that your application supports the best. This can also help backwards compatibility testing for your application.

The next column is labeled image ID, and it is based off a unique 64 hexadecimal digit string of characters. The image ID simplifies this down to the first twelve digits for easier viewing. Imagine if you had to view all 64 bits on one line! You will later learn when to use this unique image ID for later tasks.

The last two columns are pretty straightforward, the first being the creation date for the image, followed by the virtual size of the image. The size is very important because you want to keep or use images that are very small in size if you plan to move them around a lot. The smaller the image the faster the load times; and who doesn't like things faster?!

# Searching Docker images

Okay, so let's take a look at how we can search for images that are on the Docker Hub (a place to store your Docker images) using the Docker commands. The command that we will be looking at is `docker search`. With the `docker search` command, you can search based on the different criteria that you are looking for. For example, we can search for all images with the term, `Ubuntu`, in their name and see what is available. The command would go something like the following:

```
$ docker search ubuntu
```

Here is what we would get back in our results:

```
NAME                          DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
ubuntu                        Ubuntu is a Debian-based Linux operating s...   1835      [OK]
ubuntu-upstart                Upstart is an event-based replacement for ...   26        [OK]
tutum/ubuntu                  Ubuntu image with SSH access. For the root...   25        [OK]
torusware/speedus-ubuntu      Always updated official Ubuntu docker imag...   25        [OK]
```

```
ubuntu-debootstrap              debootstrap --variant=minbase --components...   10        [OK]
rastasheep/ubuntu-sshd          Dockerized SSH service, built on top of of...   4         [OK]
maxexcloo/ubuntu                Docker base image built on Ubuntu with Sup...   2         [OK]
nuagebec/ubuntu                 Simple always updated Ubuntu docker images...   2         [OK]
nimmis/ubuntu                   This is a docker images different LTS vers...   1         [OK]
alsanium/ubuntu                 Ubuntu Core image for Docker                    1         [OK]
```

Based off these results, we can now decipher some information. We can see the name of the repository, a reduced description, how many people have starred it as being something they think is a good repository, whether it's an official repository (which means that it's been approved by the Docker team), as well as whether it's an automated build. An automated build is a Docker image that builds automatically when a Git repository that it is linked to is updated. The code gets updated, a web hook gets called, and a new Docker image is built in the Docker Hub. If we find an image that we want to use, we can simply pull it using its repository name with the `docker pull` command, as follows:

```
$ docker pull tutum/ubuntu
```

The image will be downloaded and show up in our list when we now run the `docker images` command that we ran earlier.

Now that we know how to search for Docker images and pull them down to our machine, what if we want to get rid of them? That's where the `docker rmi` command comes into play. With the `docker rmi` command, you can remove unwanted images from your machine. So, let's take a look at the images that we currently have on our machine with the `docker images` command. We will get the following output:

```
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
ubuntu              14.10               ab57dbafeeea        11 days ago         194.5 MB
ubuntu              trusty              6d4946999d4f        11 days ago         188.3 MB
ubuntu              latest              6d4946999d4f        11 days ago         188.3 MB
```

We can perform this removal with the `docker rmi` command, as follows:

```
$ docker rmi ubuntu:trusty
```

Now if you issue the `docker images` command, you will see that `ubuntu:trusty` no longer shows up in your images list and has been removed. Now, you can remove machines based on their image ID as well. However, be careful when doing this because in this scenario, not only will you remove `ubuntu:trusty` but you will also remove `ubuntu:latest` as they have the same image ID. You may need to add the `–f` option if the image is referenced in one or more repositories. The `–f` option performs a force removal of the image.

# Manipulating Docker images

We just went over images and how to obtain them and manage them. Next, we are going to take a look at what it takes to fire them up and manipulate them. This is the part where images become containers! Let's first go over the basics of the `docker run` command and how to run containers. We will cover some basic `docker run` items in this section, and we will cover more advanced `docker run` items in later sections, so let's just look at how to get images up and running and turned into containers. The most basic way to run a container is as follows:

```
$ docker run -i -t <image_name>:<tag> /bin/bash
$ docker run –i –t nginx:latest /bin/bash
```

This will override the default command that is run when a container is envoked.

Upon closer inspection of the preceding command, we start off with the `docker run` command, followed by two options, `-i` and `-t`. The first `-i` option, gives us an interactive shell into the running container. The second `-t` option will allocate a pseudo tty, which when using interactive processes, must be used together with the `-I` switch. You can also use switches together; for example, `-it` is commonly used for these two switches. This will help you test out the container to see how it operates before running it as a daemon. Once you are comfortable with your container, you can test how it operates in daemon mode:

```
$ docker run -d <image_name>:<tag>
```

If the container is set up correctly and has an entry point setup, you should be able to see the running container by issuing the `docker ps` command, seeing something similar to the following:

```
$ docker ps

CONTAINER ID    IMAGE           COMMAND        CREATED          STATUS           PORTS       NAMES
cc1fefcfa098    ubuntu:14.10    "/bin/bash"    3 seconds ago    Up 3 seconds                 boring_mccarthy
```

There is also the `docker ps -a` command, which will show you all containers, even the ones that aren't running.

Running the preceding command, we get a lot of other important information beyond that the container is running. We can see the container ID, the image name on which the container is based, the command that is running to keep the image alive, when the container started up, its current status, a listing of any exposed network ports, as well as the name given to the container. Now, these names are random unless otherwise specified by the `--name=` switch. You can also expose ports on your containers using the `-p` switch, just like this:

```
$ docker run -d -p  <host_port>:<container_port> <image>:<tag>
$ docker run -d -p 8080:80 ubuntu:14.10
```

This will run the Ubuntu 14.10 container in the daemonized mode, exposing port `8080` on the Docker host to port `80` on the running container:

```
CONTAINER ID    IMAGE           COMMAND        CREATED          STATUS          PORTS                   NAMES
55cfdcb6beb6    ubuntu:14.10    "/bin/bash"    2 seconds ago    Up 2 seconds    0.0.0.0:8080->80/tcp    babbage_washington
```

Now, there will come a time when containers don't want to behave, and for this, you can see what issues you have using the `docker logs` command. This command is very straightforward. You specify the container for which you want to see the logs, which is just a redirect from stdout. For this command, you use the container ID or the name of the container from the `docker ps` output:

```
$ docker logs 55cfdcb6beb6
```

Or, you use the following:

```
$ docker logs babbage
```

You can also get this ID when you first initiate the `docker run -d` command, as follows:

```
$ docker run -d ubuntu:14.10 /bin/bash
da92261485db98c7463fffadb43e3f684ea9f47949f287f92408fd0f3e4f2bad
```

# Stopping containers

Now, let's take a look at how we can stop these containers. There can be various reasons that we want to do this. There are a few commands that we can use to do this. They are `docker kill` and `docker stop`. Let's cover them briefly as they are fairly straightforward, but let's look at the difference between `docker kill` and `docker stop`. The `docker kill` command will kill the container immediately. For a graceful shutdown of the container, you use the `docker stop` command. When you are testing, you will usually use `docker kill`, and when you are in your production environments, you will want to use `docker stop` to ensure that you don't corrupt any data. The commands are used exactly like the `docker logs` command, where you can use the container ID, the random name given to the container, or the one that you specify with the `--name=` option.

Now, let's take a dive into how we can execute some commands, view information about our running containers, and manipulate them in a small sense. We will discuss container manipulation in later chapters as well. The first thing that we want to take a look at that will make things a little easier with the upcoming commands is the `docker rename` command. With the `docker rename` command, we can change the name that has been randomly generated for the container. When we used the `docker run` command, a random name was assigned to our container. Most of the time, these names are fine. However, if you are looking for an easy way to manage containers, sometimes a name can be easier to remember. For this, you can use the `docker rename` command, as follows:

```
$ docker rename <current_container_name> <new_container_name>
```

Now that we have a recognizable and easy-to-remember name, let's take a peek inside our containers with the `docker stats` and `docker top` commands. Taking them in order, this is what we get:

```
$ docker stats <container_name>

CONTAINER     CPU %         MEM USAGE/LIMIT     MEM %           NET I/O
web1          0.00%         1.016 MB/2.099 GB   0.05%           0 B/0 B
```

The other `docker top` command gives us a list of all running processes inside the container. Again, we can use the name of the container to pull the information:

```
$ docker top <container_name>
```

We will receive an output similar to the following, based on what processes are running inside the container:

```
UID                 PID                 PPID                C
STIME               TTY                 TIME                CMD
root                8057                1380                0
13:02               pts/0               00:00:00            /bin/bash
```

We can see who is running the process (in this case, the root user), the command being run (which is `/bin/bash` in this instance), as well as other information that might be useful.

Lastly, let's cover how we can remove containers. In the same way that we looked at removing images earlier with the `docker rmi` command, we can use the `docker rm` command to remove unwanted containers. This is useful if you want to reuse a name you assigned to a container:

```
$ docker rm <container_name>
```

# Other Docker Feature Sets

In this chapter, we will take a look at the following feature sets beyond the Docker CLI:

- Docker registries
- Docker Machine
- Docker Compose
- Docker Swarm
- Docker UCP

## Storing images on Docker registries

The Docker Hub comes in a variety of flavors—three to be exact. They are used to store the images that you can then serve out to users, whether this is done internally only or if the images are also available publicly:

- **Docker Hub**: This is the hub that almost all users use, or at the very least, they start out using this. This is a free service that is hosted by Docker; however, there is a price involved when you start to utilize more than one private image repository. You can access the Docker Hub from `https://hub.docker.com`.
- **Docker Trusted Registry**: This is a solution that is hosted or can be used on premise, and the backend infrastructure is maintained by Docker. This provides you with the management piece as well as commercial support. More information can be found at `https://www.docker.com/products/docker-trusted-registry`.

- **Docker Registry**: This gives you the ability to run your own Docker registry on your own hardware or in a cloud environment to store images and make them public or private. It also allows for a simple solution that doesn't offer user management needs out of the box. More information about Docker Registry can be found at `https://docs.docker.com/registry/`.

# Docker Machine

Docker Machine is the tool that you can utilize to set up and manage your Docker hosts. You can use Docker Machine to provision Docker hosts on Mac or Windows machines and provision and/or manage remote Docker hosts. To install Docker Machine, visit `https://docs.docker.com/machine/install-machine/`.

The installation directions are dependent on your operating system. After you have installed it, you can run through the commands that Docker Machine can perform, as follows:

```
$ docker-machine

Usage: docker-machine [OPTIONS] COMMAND [arg...]

Create and manage machines running Docker.

Version: 0.6.0, build e27fb87

Author:
  Docker Machine Contributors - <https://github.com/docker/machine>

Options:
  --debug, -D            Enable debug mode
  -s, --storage-path "/Users/spg14/.docker/machine"  Configures storage
path [$MACHINE_STORAGE_PATH]
  --tls-ca-cert           CA to verify remotes against [$MACHINE_TLS_CA_
CERT]
  --tls-ca-key             Private key to generate certificates
[$MACHINE_TLS_CA_KEY]
  --tls-client-cert           Client cert to use for TLS [$MACHINE_TLS_
CLIENT_CERT]
  --tls-client-key           Private key used in client TLS auth
[$MACHINE_TLS_CLIENT_KEY]
```

```
  --github-api-token           Token to use for requests to the Github
API [$MACHINE_GITHUB_API_TOKEN]
  --native-ssh            Use the native (Go-based) SSH implementation.
[$MACHINE_NATIVE_SSH]
  --bugsnag-api-token           BugSnag API token for crash reporting
[$MACHINE_BUGSNAG_API_TOKEN]
  --help, -h            show help
  --version, -v            print the version

Commands:
  active    Print which machine is active
  config    Print the connection config for machine
  create    Create a machine
  env       Display the commands to set up the environment for the Docker
client
  inspect   Inspect information about a machine
  ip        Get the IP address of a machine
  kill      Kill a machine
  ls        List machines
  provision Re-provision existing machines
  regenerate-certs  Regenerate TLS Certificates for a machine
  restart   Restart a machine
  rm        Remove a machine
  ssh       Log into or run a command on a machine with SSH.
  scp       Copy files between machines
  start     Start a machine
  status    Get the status of a machine
  stop      Stop a machine
  upgrade   Upgrade a machine to the latest version of Docker
  url       Get the URL of a machine
  version   Show the Docker Machine version or a machine docker version
  help      Shows a list of commands or help for one command

Run 'docker-machine COMMAND --help' for more information on a command.
```

The main subcommands that you will initially want to focus on are as follows:

```
$ docker-machine create
```

The one command that you will probably use the most is this one. This is the command that will allow you to create the Docker hosts that your containers will run on. Let's take a look at an example:

```
$ docker-machine create -d virtualbox node1
```

This will create a new Docker host on a locally installed Virtualbox that will be named `node1`. If you plan to use items, such as a cloud provider or something other than Virtualbox, you will want to look at what drivers can be used with Docker Machine. You can find that list of supported drivers at `https://docs.docker.com/machine/drivers/`.

Have a look at the following command:

```
$ docker-machine ls
```

The `ls` subcommand will give you a list of the Docker hosts that you currently have running and some basic information about them. Let's take a look at some sample output:

```
NAME            ACTIVE   DRIVER    STATE     URL                         SWARM    DOCKER     ERRORS
chefclient      -        virtualbox        Running   tcp://192.168.99.100:2376                      v1.10.3
default         -        virtualbox        Running   tcp://192.168.99.101:2376                      v1.10.3
```

We can see from this output that we get information, such as the node name, whether it's the active host of not (that is, if you issue Docker commands what host will the commands run against), the driver that is being used, the state the host is in, and the URL that is used. If this were part of a Docker Swarm cluster, we would see information about what swarm cluster it was joined to. Information such as the Docker version the host is running is now part of the `ls` subcommand information as well. Lastly, if there were any errors, we would see them here as well. This information is very useful when you want to know what host your commands will be running against or the IP address of a particular Docker host.

Consider the following command:

```
$ docker-machine restart
```

One of the other commands you might use frequently at first is the `restart` subcommand. This is very straightforward and will restart the Docker host that you specify. Using our preceding output as an example, let's assume that we want to restart the `chefclient` host because it's been acting up, as follows:

```
$ docker-machine restart chefclient
Restarting "chefclient"...
Waiting for SSH to be available...
Detecting the provisioner...
Restarted machines may have new IP addresses. You may need to re-run the
`docker-machine env` command.
```

More information about Docker Machine can be found at `https://www.docker.com/products/docker-machine`.

# Docker Compose

Docker Compose is another tool in the Docker ecosystem that can be used to create multiple containers with a single command. This allows you to spin up application stacks that may include some web servers, a database server, and/or file servers as well. Docker Compose utilizes a `docker-compose.yml` file to start up and configure all the containers that you have specified. Similar to the last section on Docker Machine, let's cover Docker Compose in the same way. To start out, you can install Docker Compose by following the instructions at `https://docs.docker.com/compose/install/`.

You will want to follow the instructions at the links if you do not use Linux, as the installers are different based on the operating system that you use.

After you install it, you can run it and get the help output with the following command:

```
$ docker-compose

Define and run multi-container applications with Docker.

Usage:
  docker-compose [-f=<arg>...] [options] [COMMAND] [ARGS...]
  docker-compose -h|--help
```

```
Options:
  -f, --file FILE           Specify an alternate compose file (default:
docker-compose.yml)
  -p, --project-name NAME   Specify an alternate project name (default:
directory name)
  --verbose                 Show more output
  -v, --version             Print version and exit

Commands:
  build              Build or rebuild services
  config             Validate and view the compose file
  create             Create services
  down               Stop and remove containers, networks, images, and
volumes
  events             Receive real time events from containers
  help               Get help on a command
  kill               Kill containers
  logs               View output from containers
  pause              Pause services
  port               Print the public port for a port binding
  ps                 List containers
  pull               Pulls service images
  restart            Restart services
  rm                 Remove stopped containers
  run                Run a one-off command
  scale              Set number of containers for a service
  start              Start services
  stop               Stop services
  unpause            Unpause services
  up                 Create and start containers
  version            Show the Docker-Compose version information
```

An example `docker-compose.yml` file is as follows:

```
master:
      image:
        scottpgallagher/galeramaster
      hostname:
        master
    node1:
      image:
        scottpgallagher/galeranode
      hostname:
        node1
      links:
        - master
    node2:
      image:
        scottpgallagher/galeranode
      hostname:
        node2
      links:
        - master
```

The main subcommands that you will initially want to focus on are as follows:

```
$ docker-compose ps
```

Remember that you should run `docker-compose` commands in the directory where your `docker-compose.yml` file is located.

The `docker-compose ps` subcommand can be used to display information on the containers running within a particular Docker Compose folder. This command will help us get this information:

```
Name Command State
Ports
-------------------------------------------------------------------------
-------------------------------
galeracompose_master_1 /entrypoint.sh Up
```

```
0.0.0.0:3306->3306/tcp,
4444/tcp, 4567/tcp,
4568/tcp, 53/tcp,
```

We can gain a lot of information from this output. We can get the name of the containers that are running. These names are assigned based upon `folder_name` + `service` + `_<instance number of the service>`.

For example, the naming patter for `galeracompose_master_1` is as follows:

- The `galeracompose` part is our folder name
- The `master` part is the service name that is being used in the `docker-compose.yml` file
- The `1` part is the index of the first service instance

We also see the command that is running inside the container as well as the state of each container. In our earlier example, we see that one container is up and two are in an exit status, which means that they are off. From the one that is up, we can see all the ports that are being utilized in the container, including the protocol. Then, we can see the ports that are exposed to the outside and also the container port they are connected to:

```
$ docker-compose restart
```

The `restart` command does exactly what it says it does. As with the `pull` subcommand, it can be used in two ways. You can run it as follows:

```
$ docker-compose restart
```

It will restart all the containers that are being used in the `docker-compose.yml` file. You can also specify which service to restart, as follows:

```
$ docker-compose restart <service>
```

```
$ docker-compose restart node1
```

The `restart` command will only restart the containers that are currently running. If a container is in an exit state, then it won't start this container up to a running state:

```
$ docker-compose up
```

The `up` subcommand is used to start all the containers that are specified in a `docker-compose.yml` file. It can also be used to start up a single service from a compose file. By default, when you issue the `up` subcommand, it will keep everything in the foreground and you will be able to see the output of the container as it runs. However, you can use the `-d` switch to push all this information into a daemon and just get information on the container names on the screen.

We will take a look at `docker-compose up -d` and `docker-compose up`:

```
$ docker-compose up -d

Starting wordpresstest_db_1...
Starting wordpresstest_web_1...
$ docker-compose up
Starting wordpresstest_db_1...
Starting wordpresstest_web_1...
Attaching to wordpresstest_db_1, wordpresstest_web_1
db_1 | 150905 14:39:02 [Warning] Using unique option prefix key_buffer
instead of key_buffer_size is deprecated and will be removed in a future
release. Please use the full name instead.
db_1 | 150905 14:39:02 [Warning] Using unique option prefix key_buffer
instead of key_buffer_size is deprecated and will be removed in a future
release. Please use the full name instead.
db_1 | 150905 14:39:03 [Warning] Using unique option prefix key_buffer
instead of key_buffer_size is deprecated and will be removed in a future
release. Please use the full name instead.
db_1 | 150905 14:39:03 [Warning] Using unique option prefix myisamrecover
instead of myisam-recover-options is deprecated and will be
removed in a future release. Please use the full name instead.
........
db_1 | 150905 14:41:36 [Note] Plugin 'FEDERATED' is disabled.
db_1 | 150905 14:41:36 InnoDB: The InnoDB memory heap is disabled
Docker Compose
[ 16 ]
db_1 | 150905 14:41:36 InnoDB: Mutexes and rw_locks use GCC atomic
builtins
```

```
db_1 | 150905 14:41:36 InnoDB: Compressed tables use zlib 1.2.3.4
db_1 | 150905 14:41:36 InnoDB: Initializing buffer pool, size = 128.0M
db_1 | 150905 14:41:36 InnoDB: Completed initialization of buffer pool
db_1 | 150905 14:41:36 InnoDB: highest supported file format is
Barracuda.
db_1 | 150905 14:41:36 InnoDB: Waiting for the background threads to
start
db_1 | 150905 14:41:37 InnoDB: 5.5.38 started; log sequence number
1595675
db_1 | 150905 14:41:37 [Note] Server hostname (bind-address): '0.0.0.0';
port: 3306
db_1 | 150905 14:41:37 [Note] - '0.0.0.0' resolves to '0.0.0.0';
db_1 | 150905 14:41:37 [Note] Server socket created on IP: '0.0.0.0'.
db_1 | 150905 14:41:37 [Note] Event Scheduler: Loaded 0 events
db_1 | 150905 14:41:37 [Note] /usr/sbin/mysqld: ready for connections.
db_1 | Version: '5.5.38-0ubuntu0.12.04.1-log' socket: '/var/run/mysqld/
mysqld.sock' port: 3306 (Ubuntu)
```

You can see a huge difference. Remember that if you don't use the `-d` switch and hit *Ctrl + C* in the terminal window, it will start shutting down the running containers. While this is good for testing purposes, if you are going into a production environment, we recommend that you use the `-d` switch.

More information about Docker Compose can be found at `https://www.docker.com/products/docker-compose`.

# Docker Swarm

Docker Swarm allows you to create and manage clustered Docker servers. Swarm can be used to disperse containers across multiple hosts. It also has the ability to scale containers as well.

The installation for Docker Swarm actually launches a container that is used as the Swarm Manager master to communicate to all the nodes in a Swarm cluster. For information on how to get Docker Swarm up and running, visit `https://docs.docker.com/swarm/get-swarm/`.

Docker Swarm also has some subcommands that can be used to manage the cluster of nodes as well as the other aspects that relate to Docker Swarm.

More general and detailed information about Docker Swarm can be found at `https://docs.docker.com/swarm/`.

# Docker UCP

Docker **UCP** (**Universal Control Plane**) is a solution for Docker that enables you to control various aspects of your Docker environment through a web interface. This can be extremely helpful if you want to steer clear of the command line. You can use Docker UCP to deploy to various cloud solutions, tie into your existing authentication infrastructure, and in turn control user access.

More information about Docker UCP can be found at `https://docs.docker.com/ucp/`.

# Creating Your Own Containers

In the first chapter, we looked at how we could use containers that were already built by others. In this chapter, you will learn how to create your own containers from scratch.

## Creating containers using Dockerfile

In this chapter, we will cover the Dockerfile from a more in-depth perspective than the previous chapter and look at the best practices to use. By the end of this section, you will be structuring your Dockerfile using the most practical and efficient method. You will also be able to read and troubleshoot both yours and other people's Dockerfiles.

## Short Dockerfile review

The following is an example Dockerfile that we can then use to create a Docker image and later run as a container:

```
FROM ubuntu:latest
MAINTAINER Scott P. Gallagher <email@somewhere.com>

RUN apt-get update && apt-get install -y apache2

ADD 000-default.conf /etc/apache2/sites-available/
RUN chown root:root /etc/apache2/sites-available/000-default.conf

EXPOSE 80
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

These are the basic items that are inside the Dockerfile. The `FROM` and `MAINTAINER` fields have information about what image has to be used and the information about who is the maintainer of this image. The `RUN` instruction can be used to fetch and install packages, along with various other commands. The `ADD` instruction allows you to add files or folders into the Docker image. The `EXPOSE` instruction allows us to expose ports from the image to the outside world. Lastly, the `CMD` instruction executes the given commands and keeps the container alive. Now that we've done a really short review, let's jump in and study more in-depth items with our Dockerfile.

# Dockerfile in depth

Let's take a look at the following commands in depth:

- `LABEL`
- `COPY` or `ADD`
- `ENTRYPOINT`
    - `ENTRYPOINT` with `CMD`
- `USER`
- `WORKDIR`
    - `ONBUILD`

# LABEL

The `LABEL` command can be used to add additional information to the image. This information can be anything from a version number to a description. You will want to combine labels into a single line whenever possible. We also recommend that you limit how many labels you use. Every time you use a label, it adds a layer to the image, thus increasing the size of the image. Using too many labels can also cause the image to become inefficient. You can view container labels with the `docker inspect` command:

```
$ docker inspect <IMAGE_ID>
```

# ADD or COPY

Now, in the previous chapter and in the preceding Dockerfile example, we used the `ADD` instruction to add a file to a folder location. There is also another instruction that you can use in your Dockerfile and this is the `COPY` instruction. You can use the `ADD` instruction and specify a URL straight to a file, and it will be downloaded when the container is built. The `ADD` instruction will also unpack or untar a compressed file when added. The `COPY` instruction is the same as the `ADD` instruction but without the URL handling or the unpacking or untarring of files.

## ENTRYPOINT

In the Dockerfile example, we used the `CMD` instruction to make the container executable and to ensure that it stays alive and running. You can also use the `ENTRYPOINT` instruction instead. The benefit of using `ENTRYPOINT` over `CMD` is that you can use them in conjunction with each other.

For example, let's assume that you want to have a default command that you want to execute inside a container but then also set additional switches that may change over time. These switches are based on the command that you execute inside the `CMD` command, such as the following:

```
CMD [ "sh", "-c", "echo", "$HOME" ]


FROM ubuntu:latest
ENTRYPOINT ["ps", "-au"]
CMD ["-x"]
```

## USER

The `USER` instruction lets you specify what username to use when a command is run. The `USER` instruction influences the following `RUN` instruction, the `CMD` instruction, or the `ENTRYPOINT` instruction in the Dockerfile.

## WORKDIR

The `WORKDIR` instruction sets the working directory for the same set of instructions that the `USER` instruction can use (`RUN`, `CMD`, and `ENTRYPOINT`). This will also allow you to use the `CMD` and `ADD` instructions. These commands, `RUN` and `CMD`, are instructions that follow, such as executing the NGINX service to run.

## ONBUILD

The `ONBUILD` instruction lets you stash a set of commands that will be used when the image is used again as a base image in another Dockerfile. For example, if you want to give an image to developers and they all have different code that they want to test, you can use the `ONBUILD` instruction to lay the groundwork ahead of needing the actual code. Then, the developers simply add their code in the directory that you tell them, and when they do, a new Docker build will add their code to the build-time image. The `ONBUILD` instructions will be executed as the first statement after the `FROM` directive. `ONBUILD` can be used in conjunction with the `ADD` instruction and `RUN` instruction:

```
ONBUILD ADD
ONBUILD RUN
```

# Dockerfile best practices

Now that we have covered in depth Dockerfile instructions, let's take a look at the best practices to write these Dockerfiles:

- You should try to get in the habit of using a `.dockerignore` file. We will cover the `.dockerignore` file in the next section, but the `.dockerignore` file will seem very familiar if you are used to using a `.gitignore` file. It will essentially ignore the items that you specify in the file during the build process.
- Minimize the number of packages you need per image. One of the biggest goals that you want to achieve when building your images is to keep them as small as possible. By not installing packages that aren't necessary, it will greatly help you achieve this goal.
- Limit the number of layers in your Dockerfile.

  Every time you utilize the `RUN` command in the Dockerfile, it creates a new layer; with every layer comes added space. You will want to chain your commands together in the `RUN` command. The following is an example of how to do this:

  ```
  RUN yum update; yum install -y nginx
  ```

- Execute only one application process per container. Every time you need a new application, it is best practice to use a new container to run this application in. While you can couple commands into a single container, it's best to separate them out.
- Sorting commands can be done in the following ways:
  - You can sort them based upon the actual command itself:

    ```
    RUN apt-get update && apt-get install -y
    ```

  - You can sort them alphabetically so that it's easier to change later:

    ```
    RUN apt-get update && apt-get install -y \
           apache2 \
           git \
           memcached \
           mysql
    ```

# Docker build

In this section, we will cover the `docker build` command. It's time for us to build the base that all our future images will start out being built on. We will be looking at different ways to accomplish this goal. Consider this as a template that you may have created earlier with virtual machines. This will help you save time by having the hard work already completed. Then, just the application that needs to run has to be added to the new images that you will create.

# The docker build command

Now that we have learned how to create and properly write a Dockerfile, it's now time to learn how to take it from just a file to an actual image. Now, there are a lot of switches that you can use when using the `docker build` command, so let's use the always handy `--help` switch on the `docker build` command to view all we can do, as follows:

```
$ docker build --help

Usage:  docker build [OPTIONS] PATH | URL | -

Build an image from a Dockerfile

  --build-arg=[]                    Set build-time variables
  --cpu-shares                      CPU shares (relative weight)
  --cgroup-parent                   Optional parent cgroup for the
container
  --cpu-period                      Limit the CPU CFS (Completely Fair
Scheduler) period
  --cpu-quota                       Limit the CPU CFS (Completely Fair
Scheduler) quota
  --cpuset-cpus                     CPUs in which to allow execution (0-3,
0,1)
  --cpuset-mems                     MEMs in which to allow execution (0-3,
0,1)
  --disable-content-trust=true      Skip image verification
  -f, --file                        Name of the Dockerfile (Default is
'PATH/Dockerfile')
  --force-rm                        Always remove intermediate containers
  --help                            Print usage
```

```
  --isolation                      Container isolation level
  -m, --memory                     Memory limit
  --memory-swap                    Swap limit equal to memory plus swap:
'-1' to enable unlimited swap
  --no-cache                       Do not use cache when building the
image
  --pull                           Always attempt to pull a newer version
of the image
  -q, --quiet                      Suppress the build output and print
image ID on success
  --rm=true                        Remove intermediate containers after a
successful build
  --shm-size                       Size of /dev/shm, default value is 64MB
  -t, --tag=[]                     Name and optionally a tag in the
'name:tag' format
  --ulimit=[]                      Ulimit options
```

Now, it looks like a lot to digest but the most important ones will be the `-f` and the `-t` switches. You can use the other switches to limit how much CPU and memory the build process uses. In some cases, you may not want the `build` command to take as much CPU or memory as it can use. The process may run a little slower. However, if you are running this on your local machine or a production server and it's a long build process, you may want to set a limit. Typically, you don't use the `-f` switch as you run the `docker build` command from the same folder that the Dockerfile is in. By keeping the Dockerfiles in separate folders, it helps sort the files and keeps the naming convention of the files the same:

```
$ docker build -t <Docker Hub username> / <repository name> <directory
```

You can use the `.` (period) character to specify the current directory:

```
$ docker build -t scottpgallagher/apache-web .
```

# The .dockerignore file

The `.dockerignore` file, as we discussed earlier, is used to exclude files or folders that we don't want to be included in the Docker build from being sent to the Docker daemon before the build. We also discussed placing the Dockerfile in a separate folder, and the same applies for the `.dockerignore` file. This should go in the folder where the Dockerfile was placed. Keeping all the items that you want to use in an image in the same folder helps you keep the items, if any, in the `.dockerignore` file to a minimum.

# Building containers using Dockerfile

The first way that we are going to look at to build your base Docker images is by creating a Dockerfile, populating the Dockerfile with some instructions, and then executing a `docker build` command against it to get ourselves a base container. So, let's first start off by looking at a typical Dockerfile:

```
FROM ubuntu:latest
MAINTAINER Scott P. Gallagher <email@somewhere.com>

RUN apt-get update && apt-get install -y apache2

EXPOSE 80
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

In this Dockerfile, this is very straightforward. We are going to use the latest Ubuntu image, and then we are going to run the `apt-get update` command as well as the `apt-get install` command of Apache web server. We will set the container to expose port `80` when it is run and then start Apache in the foreground of the container.

So, there are two ways we can go about building this image. The first way would be by specifying the `-f` switch when we use the `docker build` command. We will also utilize the `-t` switch to give the new image a unique name:

```
$ docker build -f <path_to_Dockerfile> -t <REPOSITORY>:<TAG>
```

Now, the `<REPOSITORY>` is typically the repository name that is prefixed by your Docker Hub username:

```
$ docker build -f <path_to_Dockerfile> -t scottpgallagher/ubuntu_apache
```

Typically, the `-f` switch isn't used and can be a little tricky when you have other files that need to be included with the new image. An easier way to do the build is to place the Dockerfile in a separate folder by itself, along with any other files that you will be placing in the image with the `ADD` or `COPY` instructions:

```
$ docker build -t scottpgallagher:ubuntu_apache .
```

The most important thing to remember is the `.` character—the period at the very end. This is used to tell the `docker build` command to build in the current folder.

If you are using your own registry to push your images, then you can use any naming convention that you like, but try to keep it simple and use something easy to identify when looking at the name.

# Modifying and committing an existing image

The easiest way to build a base image is to start off using one of the official builds from the Docker Hub. Docker also keeps the Dockerfiles for these official builds on their GitHub repositories. So, there are at least two choices that you have to use existing images that others have already created. Using the Dockerfile, you can see exactly what is included in the build and add what you need. You can then version control this Dockerfile for it if you want to change it at a later time.

The other way is using an already existing image that requires a little bit more work, though this is essentially the same method. We would first need to get the base image that we want, as follows:

```
$ docker pull ubuntu:latest
```

Then, we would run the container in the foreground so that we can add packages to it:

```
$ docker run -it ubuntu:latest /bin/bash
```

After the container is running, you can add the packages as necessary using the `apt-get` command in this case or whatever the package manager commands are for your Linux flavor. After you have installed the packages that you require, you then need to save the container. To do so, you first need to get the container ID. We do this in the following manner:

```
$ docker ps
```

Once you have the container ID, you can now save (or commit) the container. So, to save this container, we would do something similar to the following:

```
$ docker commit <container_ID> <REPOSITORY>:<TAG>
```

Now if you are planning on using the Docker Hub (which we will be discussing here shortly in the next section), you will want to structure your image names, as follows:

```
$ docker commit <container_ID> <Docker_Hub_Username>:<Unique_Name>
$ docker commit <container_ID> scottpgallagher:ubuntu_apache2
```

This will preserve any `CMD` or `ENTRYPOINT` instructions that are inherited from the base image. You can change those with the `--change` option:

```
$ docker commit --change 'CMD ["/bin/bash"]' <container_ID> <image_name>"
```

# Building your own containers

There are two ways to go about building your own containers:

- Using tar to import a filesystem
- Using a scratch image

## Building using tar

So, you have a machine already running as a virtual machine or on a bare metal box, and you want to convert this to a Docker image. How do you go about doing this? The first thing that you will need to do is install something similar to `debootstrap`. This will create your base Debian system. Later, we will add the files to the following image:

```
$ sudo apt-get install -y debootstrap
```

Next, you need to get the release name of the distribution of Linux that you are running. To do this, we can look at the contents of the `/etc/lsb-release` file:

```
$ cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=14.04
DISTRIB_CODENAME=trusty
DISTRIB_DESCRIPTION="Ubuntu 14.04.2 LTS"
```

We can tell from the preceding output that we are running the **trusty** release of Ubuntu. Now, we can execute the next command (to build the new container) using the newly installed `debootstrap` command, which will take some time to run:

```
$ sudo debootstrap trusty <directory_name> > /dev/null
```

Now, we can execute the next command after the preceding one has completed:

```
$ sudo tar -C <directory_name> -c . | sudo docker import - <image_name>
```

The preceding command will switch to the directory that you specify after the `-C` option, and then it will create a new archive from this directory based off the `-c` switch and specifying the `.` (for current directory). This will then import the archive into a new Docker image with the `docker import` command.

You can see this image by issuing the `docker images` command, as follows:

```
$ docker images
REPOSITORY      TAG          IMAGE ID      CREATED       VIRTUAL SIZE
ubuntu_trusty   latest     376bfebd75c    17 minutes ago     228.3 MB
```

You can then use this image for base images and share on the Docker Hub or on your own Docker registry. We will be covering how to push these images to various locations in the next section. However, we first need to look at the other method of creating images, and that is building from scratch.

If you use something other than Ubuntu (or Debian), Docker has also created scripts that you can utilize to create images from, which you can check out at `https://github.com/docker/docker/tree/master/contrib`.

You will want to look at the `mkimage-` files, based on what distribution you use.

## Building using scratch

You also have the option to build from scratch. Now, when you usually hear the term scratch, it literally means you start from nothing. That's what we have here. You get absolutely nothing and have to build upon it. Now, this can be a benefit because it will keep the image size very small, but it can also not be a benefit if you are fairly new to the Docker game because this may be a little complicated.

Docker has done the hard work for us already and created an empty tar file that is on the Docker Hub named `scratch` that you can use in the `FROM` section of your Dockerfile. You can base your entire Docker build off of this and then add parts as needed. So, your Dockerfile might look something like the following:

```
FROM scratch
ADD <script_to_add> /<path_to_add_to_on_container>
CMD ["/<path_to_add_to_on_container>"]
```

# Command Cheat Sheet

This chapter was written specifically for reference purposes. This is a place to give you a list of useful Docker commands that you should be familiar with before proceeding on to more advanced Docker items or commands. This sheet is great to print off and hang somewhere handy in the event that you get stuck on a command.

Always reference the `help` switch on any Docker command to get additional help as well.

## Running containers

The following command can be run to start a container, but by utilizing the `-it` switch, you are asking for the container to provide you with interactive shell as well as allocate a pseudo tty or a terminal:

```
$ docker run -it nginx:latest /bin/bash
```

This command is great to use when you want to try to troubleshoot a container or test out a new container before utilizing the next command, which utilizes the daemon mode:

```
$ docker run -d nginx:latest
```

This command will allow you to run the container in the background. This is great when you don't want to interact with the container. You can do this through other Docker commands. However, this will start the container up, and it will continue to run in the background until stopped or interrupted.

While you are running containers, they will start to take up space on your Docker host. For this reason, you can add the `--rm` switch to your `run` command:

```
$ docker run -d --rm nginx:latest
```

This will remove the container from the Docker host it is running on when it exits. Think of this like automatic cleanup.

If you wanted to execute a command against a running container, you would use the Docker `exec` subcommand:

```
$ docker exec <container_name> <command>
$ docker exec nginx "yum –y update nginx"
```

Remember to always look at the `help` subcommands as well. You can do this by attaching the `--help` switch onto any command:

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

# Building containers

Now, let's move on to building containers. There is one basic command that you always need to remember when building. This is the command that we will look at next:

```
$ docker build -t scottpgallagher/nginx .
```

In the preceding command you can see we are performing a `build` subcommand and utilizing the `-t` switch. This allows us to tag the instance that we are building. Based on the Docker Hub, you typically name your containers based on the following:

- Your username
- Name for the container/image

In our example, our username is `scottpgallagher` and our image name is `nginx`.

The last part, which might be hard to see, but is *extremely* important is the `.` (period) that specifies the `docker build` command to build based on the current directory, that is, the directory in which the Dockerfile file is located.

Lastly, don't forget to check out the other switches that you can utilize with the `docker build` subcommand using the `--help` switch.:

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

# Docker Hub commands

There are a few Docker commands that relate to the Docker Hub that are very useful to know.

The first is the following command, which allows you to log in to the Docker Hub:

```
$ docker login
```

This command will then prompt you for the following:

- Username
- Password
- E-mail address

What if you want an image from the Docker Hub? How do you get it onto your Docker host? You use the following command:

```
$ docker pull nginx
```

This will pull the latest NGINX image from the Docker Hub to your Docker host.

After you have created images or updated another image, you need to push the image using the following command:

```
$ docker push scottpgallagher/nginx
```

The format again is `<Docker Hub username>/<Image Name>`.

# Docker Swarm commands

To get started with Docker Swarm, you first need to issue the following command, which will generate a random token that you will use in later commands to join nodes to:

```
$ docker run --rm swarm create
```

Based on the output from the previous command, you can then invoke a Docker Machine command to create a new Docker container and join it to the existing Swarm cluster:

```
$ docker-machine create \
-d virtualbox \
--swarm \
--swarm-discovery token://85b335f95e9a37b679e2ea9e6ad8d6361 \
swarm-node1
```

From the preceding command, note the `--swarm` switch that tells Docker that this is a container to use with Swarm, followed by the `--swarm-discovery` switch into which we can put the token that was generated from our previous command.

Lastly, if we want to look at all the Docker hosts that are currently joined to a Swarm cluster, we can use the following command. Again, we use the `--rm` switch which will remove the container once it has completed its work. In this case, it gives us a list of the hosts joined using a particular token:

```
$ docker run --rm swarm list token://85b335f95e9a37b679e2ea9e6ad8d6361
```

# Docker Machine commands

A few Docker Machine commands that are useful and that you will use a lot are given here.

To create a new Docker host, use the following command:

```
$ docker-machine create -d virtualbox node1
```

The structure of this command is `-d <driver_name> <Docker host name>`.

What if you want to see all the Docker hosts you have?

```
$ docker-machine ls
```

The `ls` switch will provide you with information about the Docker hosts, such as their IP addresses, name of host, which one is active (that is which one the Docker commands will run against), and some other useful information.

Lastly, if you want to stop a Docker host, you can use the `stop` subcommand:

```
$ docker-machine stop node1
```

# Docker Compose commands

Remember that Docker Compose is used to bring up a multicontainer environment as opposed to the `docker run` command, which will start just one container at a time. Docker Compose uses the Dockerfile, but it also relies on the `docker-compose.yml` file that has all the information about the multicontainer environment and how to set it all up.

To get started, you need to know how to start the multicontainer environment:

```
$ docker-compose up
```

Remember that this command needs to be run inside the folder where both the Dockerfile and `docker-compose.yml` file are located.

Next, what if you want to scale a particular service or container that is running inside your environment? You can use the `scale` subcommand to do this:

```
$ docker-compose scale web=3
```

You will specify the container name (in our case, this is `web`) that you are using inside your `docker-compose.yml` file. In our example, this will create two additional web containers and also run all the commands that are inside the `docker-compose.yml` file for this container. If there is linking that needs to be done, it will do this; or if a volume needs to be mounted from another container, it will do this as well.

# Summary

In this short time, we have covered quite a lot of information. We first looked at a short history of how Docker came about, how Docker differs from traditional virtual machine environments, and some of the quick benefits of using Docker.

Then, we took a look at how to work with containers and images. Some useful commands to start and stop containers were also covered. We took a look at the other Docker feature sets, such as Docker registries and how many options there are to store your images. Other items, such as Docker Machine to control your Docker hosts, Docker Compose to make multitenant environments, and also Docker Swarm to create clustered Docker server environments were also covered.

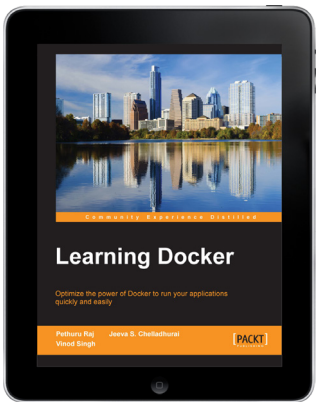
We followed this up by looking at how to create your own containers using both the Dockerfile format and the existing environments and leveraging tools, such as tar and `debootstrap`.

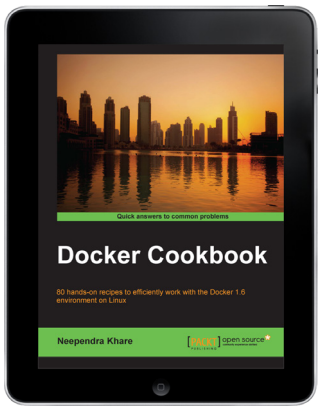
Finally, we performed a quick overview of the useful Docker commands throughout the various Docker products. This will be a great section to look back on and be able to quickly reference as needed.

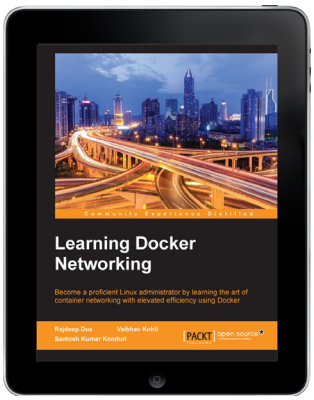
# What to do next?

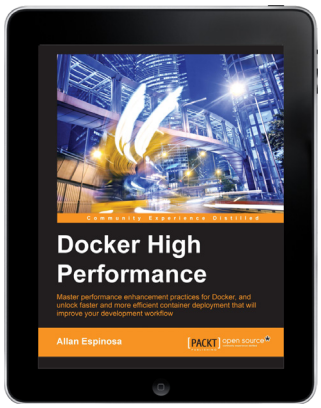
## Broaden your horizons with Packt

If you're interested in Docker, then you've come to the right place. We've got a diverse range of products that should appeal to budding as well as proficient specialists in the field of Docker.



To learn more about Docker and find out what you want to learn next, visit the Docker technology page at `https://www.packtpub.com/tech/docker`.

If you have any feedback on this eBook, or are struggling with something we haven't covered, let us know at `customercare@packtpub.com`.

Get a 50% discount on your next eBook or video from `www.packtpub.com` using the code: