

DOCKER

A Project-Based
Approach
to Learning

JASON CANNON



DOCKER

A Project-Based Approach to Learning

Jason Cannon

LinuxTrainingAcademy.com

Copyright © 2021 Jason Cannon

All rights reserved

No part of this book may be reproduced, or stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without express written permission of the publisher.

CONTENTS

[Title Page](#)

[Copyright](#)

[Online Resources](#)

[Introduction and Overview](#)

[Installing Docker](#)

[Introduction to Docker](#)

[Docker Basics and Common Commands](#)

[Managing Docker Container Images](#)

[Exercise: Managing Images](#)

[Running and Managing Docker Containers](#)

[Exercise: Running Containers](#)

[Making a Container Publicly Available](#)

[Exercise: Making a Container Publicly Available](#)

[Connecting to Running Containers and Managing Container Output](#)

[Exercise: Entering and Connecting to Containers](#)

[Docker Logging](#)

[Docker Registries](#)

[Building Images with Dockerfiles](#)

[Exercise: Build and Push an Image](#)

[Docker Volumes](#)

[Exercise: Managing Docker Volumes](#)

[Docker Networking and Dockerizing Applications](#)

[Exercise: Docker Networking](#)

[Docker Swarm](#)

[Exercise: Deploying a Private Docker Registry](#)

[The End and the Beginning](#)

[Other Books by the Author](#)

[Courses by the Author](#)

ONLINE RESOURCES

To download all the exercises contained in this book, as well as additional resources, visit <https://www.LinuxTrainingAcademy.com/docker>. Any updates or corrections to this book will be shared there as well.

INTRODUCTION AND OVERVIEW

Welcome to this book on Docker, where you'll learn the ins-and-outs of the world's most popular container platform.

My name is Jason Cannon, and I'm the author of several best-selling Linux books, including *Linux for Beginners* and *Command Line Kung Fu*. I've also created dozens of courses that have helped people just like you level-up their careers by teaching them valuable, in-demand skills. I've been using Docker since its release, and I've put all of my best Docker tips, tricks, and techniques into this book.

In order for you to get Docker up and running as quickly as possible, the beginning of the book will show you how to install it on your chosen operating system. It will outline how to start, stop, and resume containers, how to display important information about those containers, and how to use the built-in Docker help system.

From there, you'll learn all about Docker images: what images are, where to get them, how to manage them, and even how to create your very own custom images. You'll also learn how to create a private Docker image registry, so you can have complete control over your images in your environment. If you're worried about keeping sensitive data out of the wrong hands, you can deploy a private registry.

Next, you'll learn how to manage the data needed by and created by your Docker containers. You'll learn about the different ways to persist data between container runs, including how to make the most of Docker volumes. Plus, you'll find out how to view and manage the output and logs generated by your containers.

I will demonstrate how Docker supervises network traffic, including how it

routes traffic from the outside world to and from a given container, and how it isolates containers for greater security. Most importantly, you'll learn why, when, and how to create and manage Docker networks for your specific needs and applications.

Having taught thousands of students over the years, I've found that people learn best by doing. To help you with this, there are practice exercises throughout the book, each with detailed, step-by-step instructions.

Not only will you learn the concepts, you will be able to put those concepts to good use by practicing these new skills immediately so that you fully understand and retain what you're learning.

Each project and exercise will allow you to build upon the skills you've acquired along the way. By the end of this book, you'll be able to deploy complex applications using multi-container configurations, user-defined networks, and Docker volumes with ease.

This book is ideal for anyone who has a desire to learn how to configure, deploy, and manage Docker systems, Docker containers, Docker registries, Docker orchestrators, or Dockerized applications.

Whether you're a developer, programmer, or a software engineer looking for best-practices to deploy your applications quickly and easily using Docker; or a System Administrator, Systems Engineer, or Operations Engineer who needs to support Docker deployments, this book is for you.

Ultimately, this is the perfect guide for those who are looking to advance their careers by learning a key DevOps Skill: Docker.

So, if you're tired of being scared of Docker, or simply aren't making the progress you'd hope for by dabbling with Docker containers, it's time to take your Docker skills to the next level by reading this book.

INSTALLING DOCKER

In this chapter, you will learn which edition of Docker to install and why. It can be a relatively complicated choice for newcomers to Docker, as there are so many options. Here, we will cover the differences between the available versions, so you can decide which best suits your requirements. Additionally, you'll learn how to install Docker on various operating systems, including Windows, Mac, and Linux.

Docker Community Edition vs. Docker Enterprise Edition

The two main versions of the Docker Engine promoted by Docker today are ‘Docker CE’, which is the Community Edition, and ‘Docker EE’, which is the Enterprise Edition. This book is based on the Community Edition, as it is suitable for individuals, small teams of developers, and even large organizations that want to run Docker without paying a subscription fee. In short, the Community Edition is free for anyone to download and run.

The Enterprise Edition provides additional premium features that are not included in the Community Edition, such as being able to verify that images created by vendors are trustworthy and verified. These images are certified by ISVs (Independent Software Vendors), and work seamlessly with Docker Enterprise Edition. The Docker Enterprise Edition is a commercial product, available for a fee.

Another Docker offering is ‘Docker Enterprise’, which is not related to the Docker Engine, but rather a container platform with a full suite of extras. These extras include the ability to scan container images for common vulnerabilities, as well as access to an orchestrator such as Docker Swarm or Kubernetes.

Docker Community Edition Channels

There are three types of update channels for Docker CE. The most commonly

used is the stable update channel, which contains software that has been thoroughly tested. The second channel is called ‘test’, and contains software that hasn't yet completed testing. Finally, the nightly channel contains the most recent work in progress. We will focus on the stable channel, as we want to ensure our Docker environment has been thoroughly tested and works. If you install from a nightly build, for example, you may run into unexpected bugs and other issues. Again, I highly recommend using the stable channel unless you are testing a new feature that Docker hasn't yet promoted to the stable version.

One thing to note is that releases of Docker CE are supported through patches that are available for approximately seven months after each version's release. Docker EE extends that support to approximately twenty-four months after a new version is released.

Installing Docker on Windows

In this section, you're going to learn how to install Docker on Windows. If you want to install Docker on another operating system, such as Mac or Linux, skip ahead to the section that corresponds to your operating system.

Before you install Docker on Windows, let's discuss some requirements. At the time of publication, Docker requires at least Windows 10 Pro, Windows 10 Enterprise, or Windows 10 Education Edition. Currently, Docker is not supported on Windows 10 Home Edition.

Also, Docker requires that virtualization be enabled in your physical computer's BIOS. Sometimes it's labeled as VT-x for Intel processors or AMD-V for AMD processors. This varies depending on the manufacturer, so consult the documentation for your specific computer. Luckily, this is usually already enabled, but if you run into any issues, double check and make sure that virtualization is enabled in your BIOS. Note that this is different from having Hyper-V enabled. These virtualization settings are at the hardware level, not at the operating system level.

Also, be aware that Docker and other virtualization software such as VirtualBox or VMWare can't run on the same host at the same time.

Download Docker Desktop for Windows

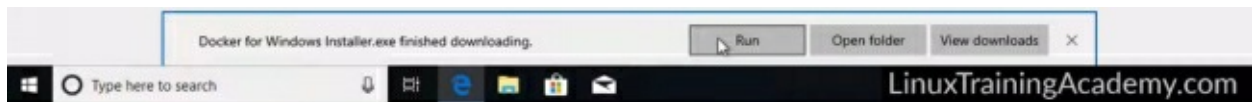
First download Docker. At the time of publication, the direct download link is:

<https://desktop.docker.com/win/stable/amd64/Docker%20Desktop%20Installer.exe>

This may change over time, so if the link does not work, go to <https://www.docker.com> and find the download section of the website to get Docker for your operating system.

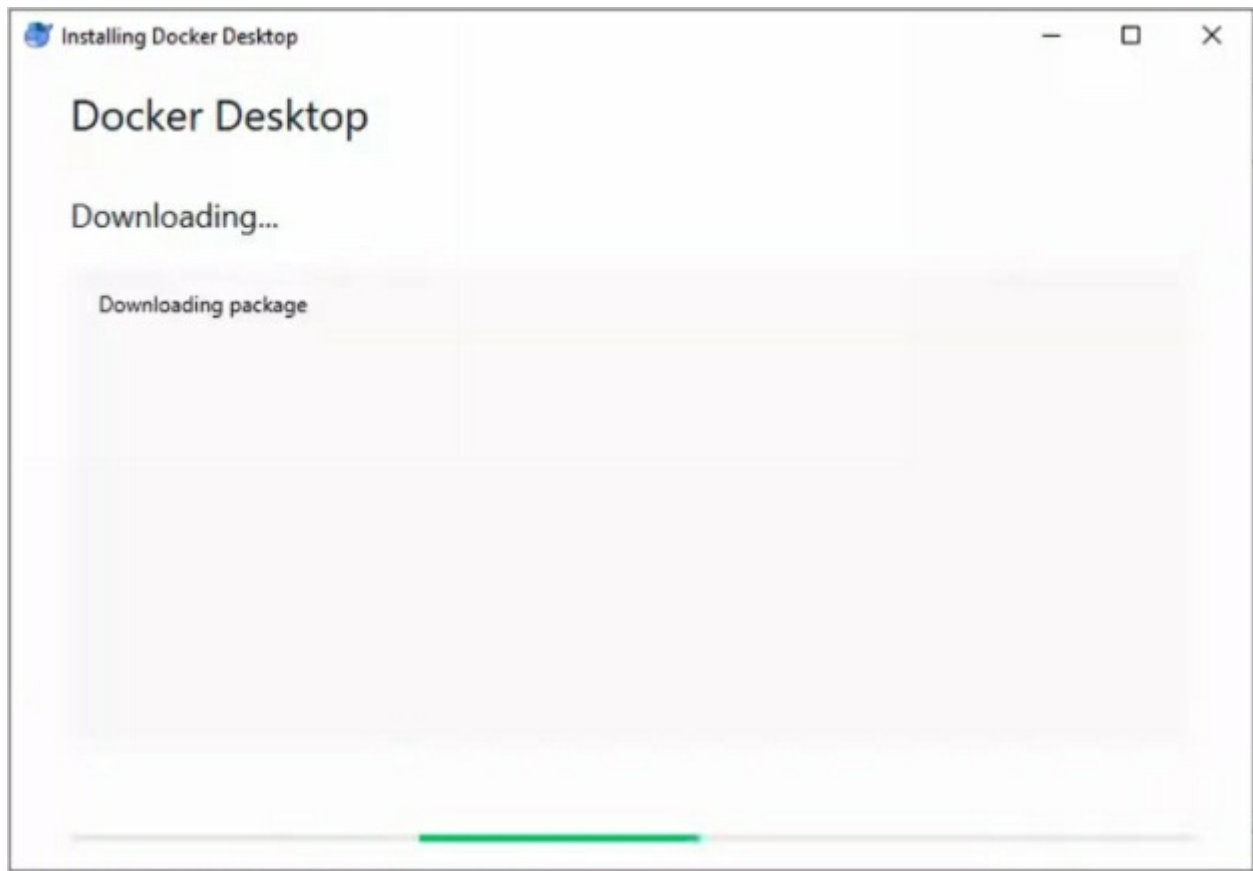
Install Docker Desktop

Once you have downloaded the installation file, run it.



Docker requires administrative privileges to perform the installation, so be sure that you're logged in as a user who has administrative privileges on your Windows system.

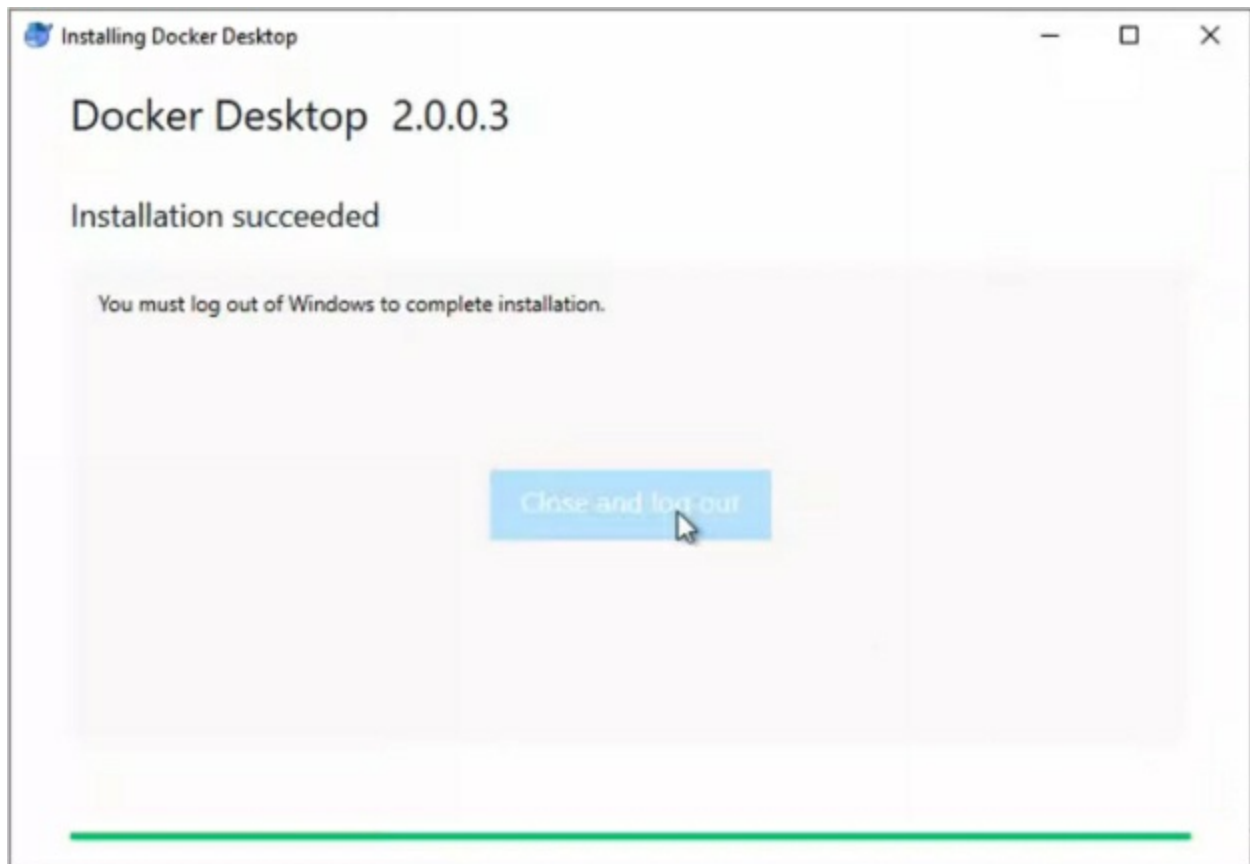
Accept the defaults. Click “OK” to continue.



Next, click "OK."



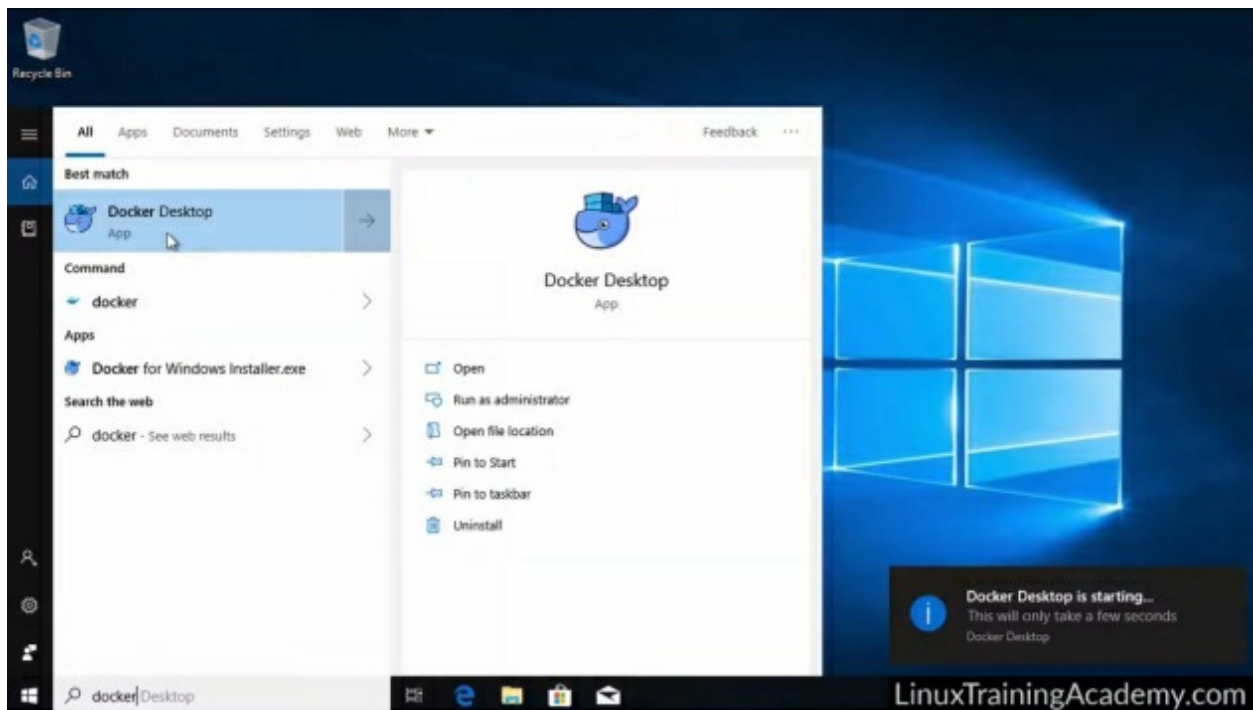
Close and log out when you're prompted to. You'll be logged out of your Windows system, and will have to log back in to continue the installation process.



Once you have logged back into your system, double-click the Docker Desktop icon on your desktop:



Alternatively, you can start Docker Desktop by using the search bar to find it.



You will see a message that Docker is starting. A small Docker icon will also

appear in your task bar.



If you do not have Hyper-V already installed, Docker will do this for you. When prompted to install Hyper-V, click "OK".



At this point, your system may need to reboot to complete the installation.

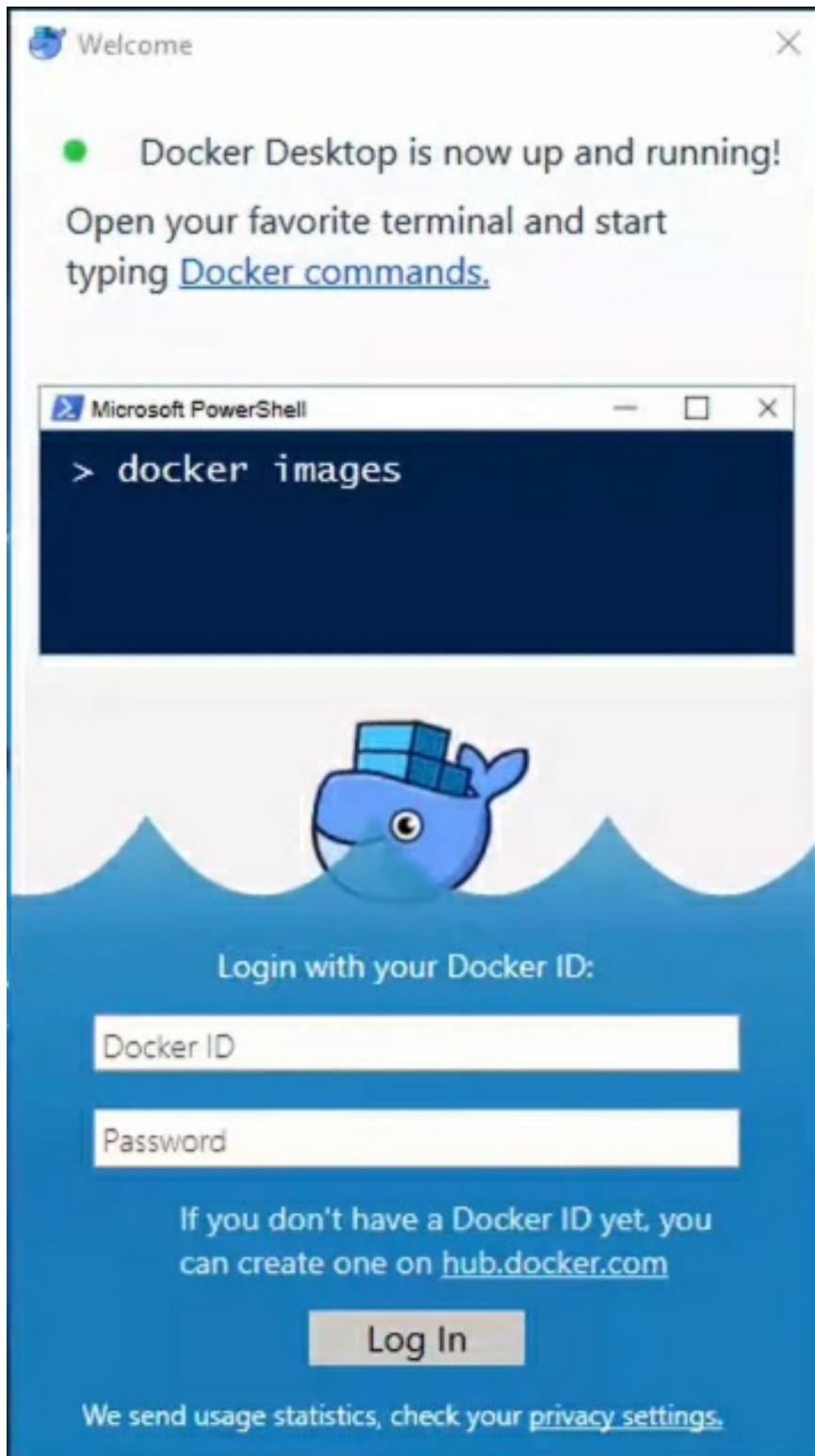
During this initial phase, you will have to manually start Docker by double-clicking on the Docker Desktop icon.



To check the status of Docker, look in the menu bar. For example, the image below says that Docker desktop is starting.



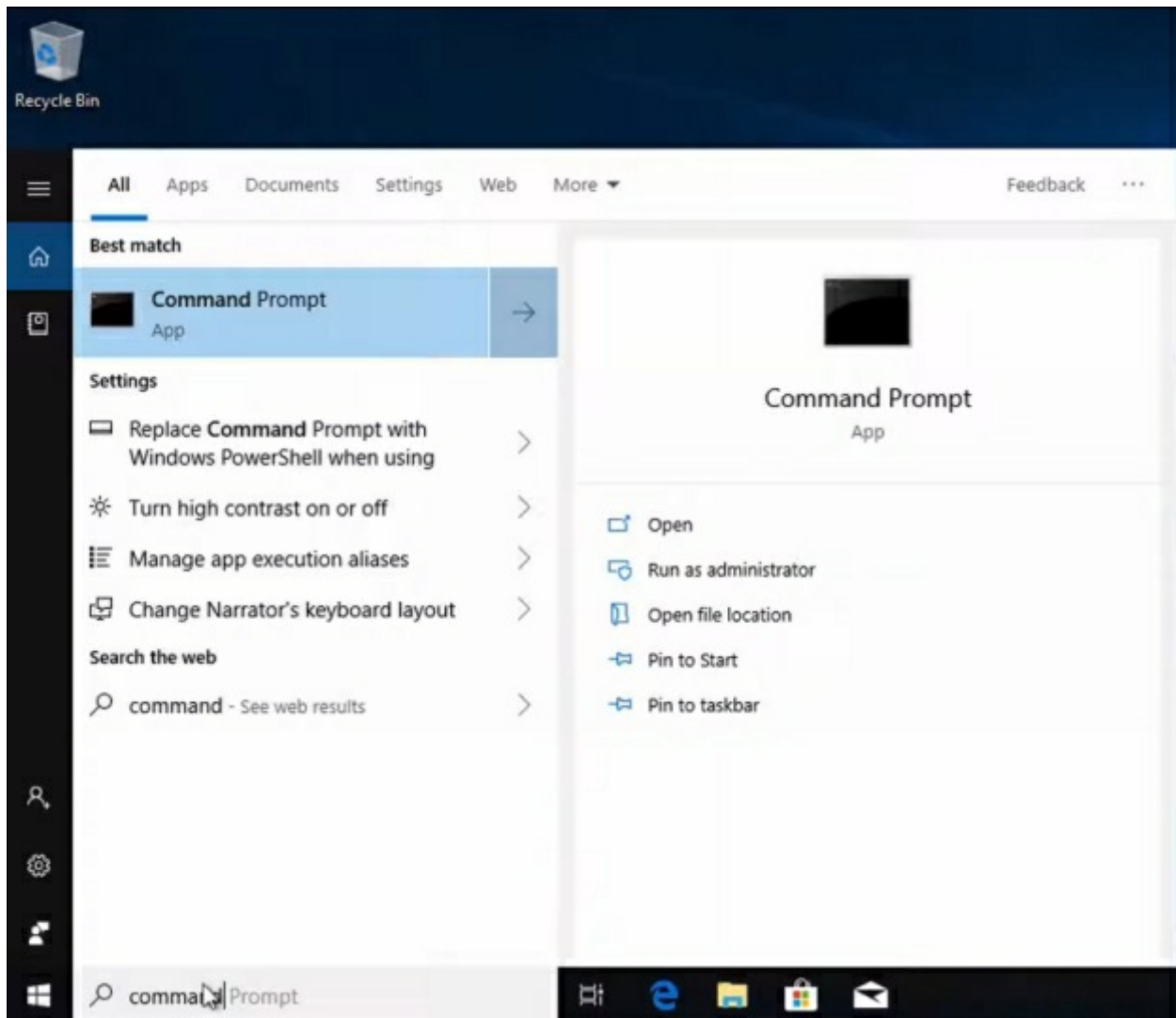
Give Docker Desktop a few minutes to start. Once Docker has been successful at completely installing and running, a pop-up message like this will appear:



Click "X" to dismiss the message.

Use the command line to verify that Docker is working properly. First, open a

command prompt as administrator. One way to do this is to click in the search bar and type in "command".



Once you have "Command Prompt" highlighted, click on "Run as administrator." By the way, you can also use PowerShell and run that as administrator.

Once you are at the command line, run the following command:

```
docker version
```

If Docker is installed and working, you will see the following information displayed on your screen:

```
Administrator: Command Prompt

Microsoft Windows [Version 10.0.17763.557]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\jason>docker version
Client: Docker Engine - Community
 Version:           18.09.2
 API version:       1.39
 Go version:        go1.10.8
 Git commit:        6247962
 Built:             Sun Feb 10 04:12:31 2019
 OS/Arch:           windows/amd64
 Experimental:      false

Server: Docker Engine - Community
 Engine:
  Version:           18.09.2
  API version:       1.39 (minimum version 1.12)
  Go version:        go1.10.6
  Git commit:        6247962
  Built:             Sun Feb 10 04:13:06 2019
  OS/Arch:           linux/amd64
  Experimental:      false

C:\Users\jason>
```

Congratulations! You've installed Docker on Windows! Going forward, remember that you need to run the Command Prompt or PowerShell as Administrator in order to interact with Docker successfully.

Installing Docker on Mac

In this section, you will learn how to install Docker on a Mac. If you want to install Docker on another operating system, skip ahead to the section that applies to your specific operating system.

First, download Docker Desktop for Mac. At the time of publication, the direct download link for Macs with Intel processors is:

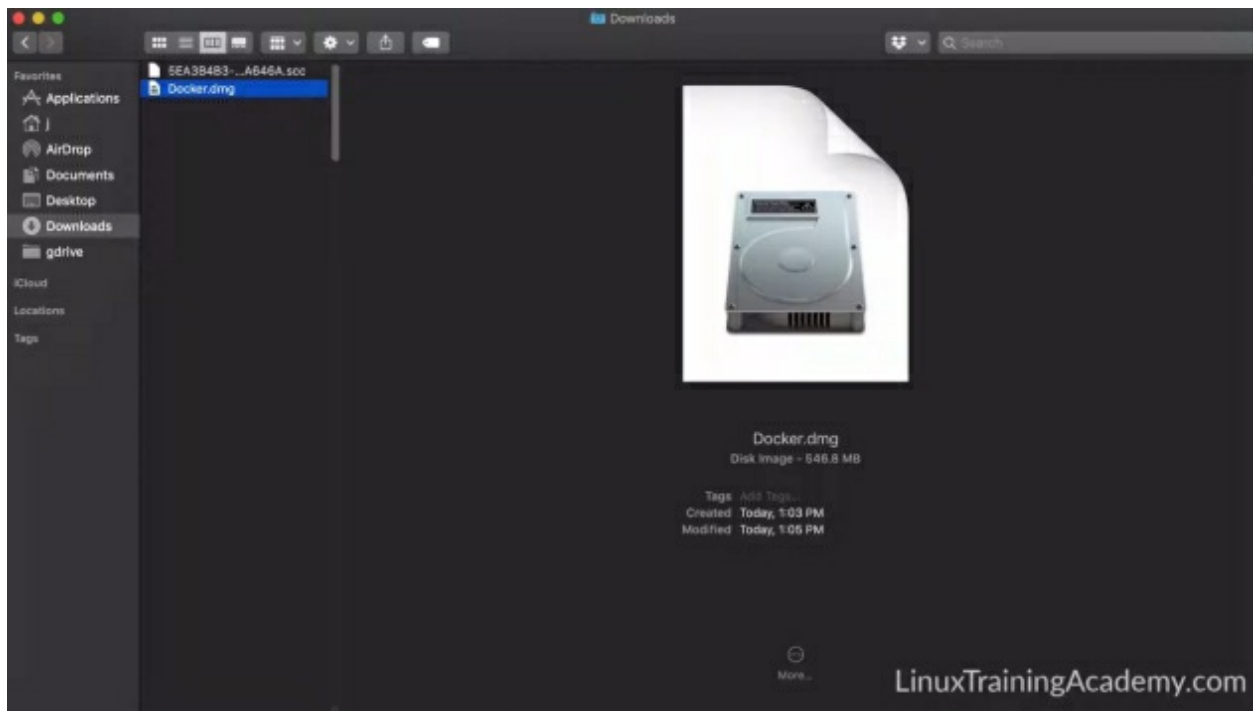
<https://desktop.docker.com/mac/stable/amd64/Docker.dmg>

For Macs using ARM processors, the link is:

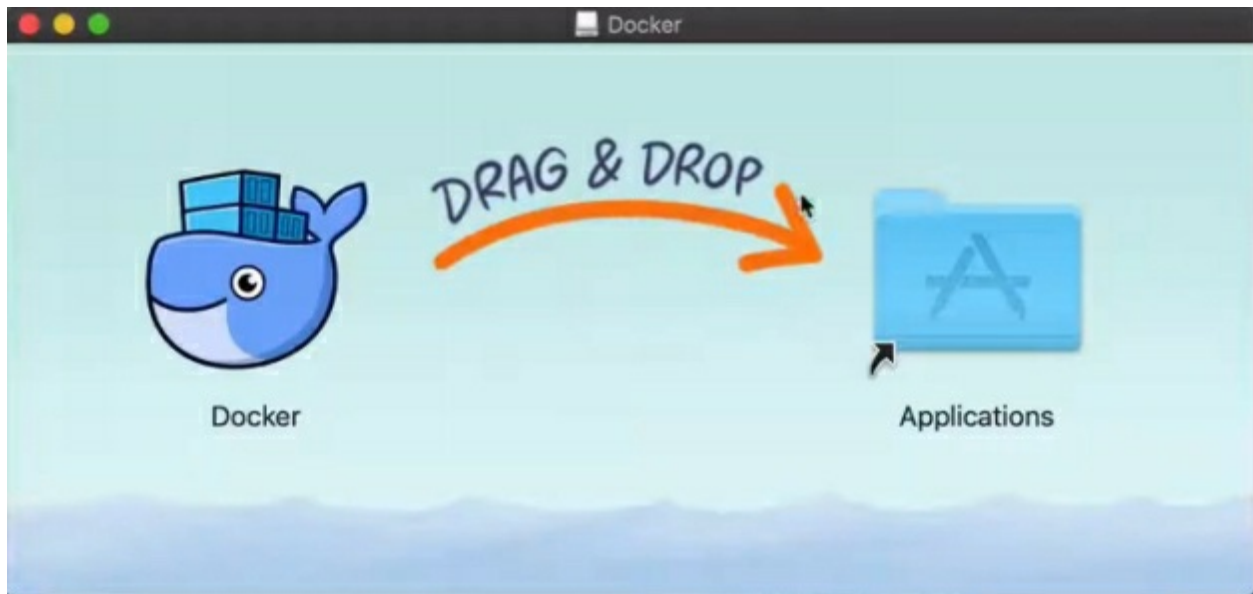
<https://desktop.docker.com/mac/stable/arm64/Docker.dmg>

This may change over time, so if the link does not work, go to <https://www.docker.com> and find the download section to get Docker for your operating system.

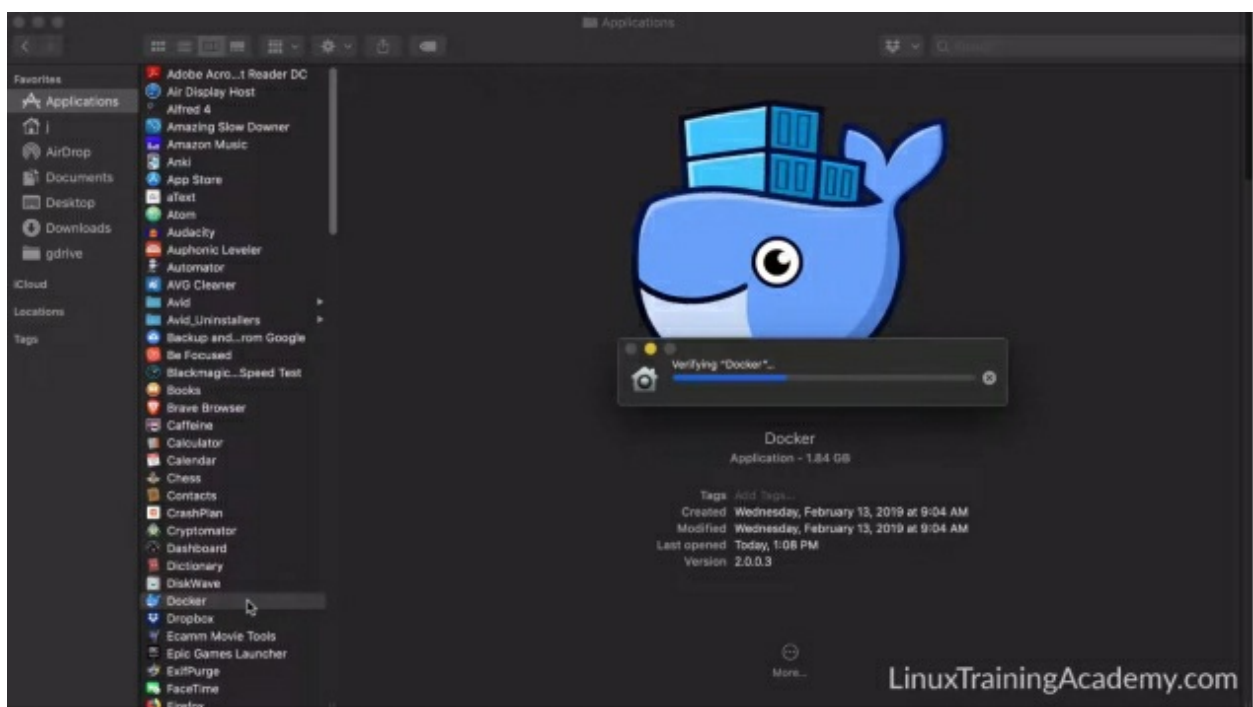
Once Docker has been downloaded, find the file on your disk and double-click to open it. It will most likely be in your "Downloads" folder.



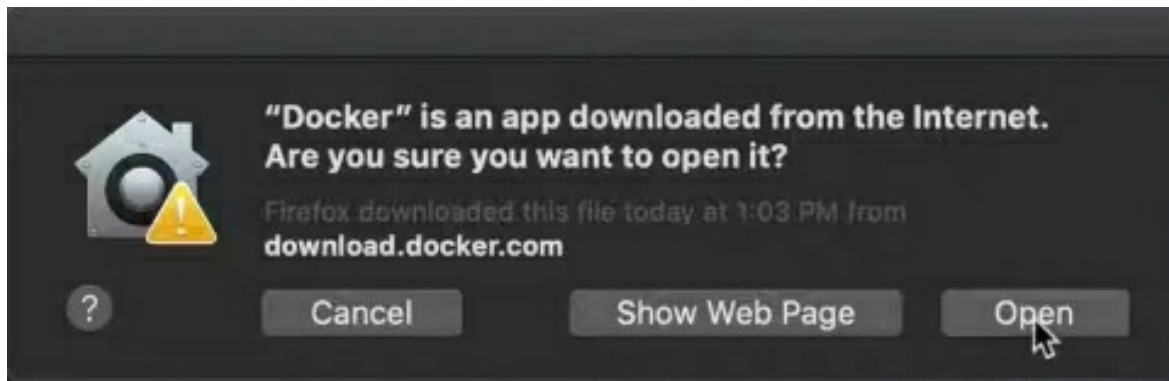
Drag the Docker icon to the application shortcut. Close that window.



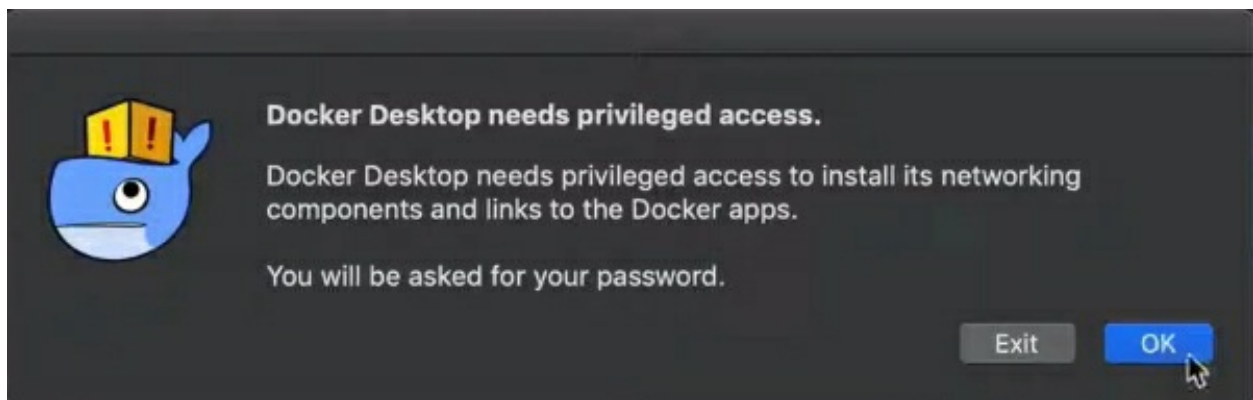
Next, navigate to the Applications folder to find Docker. Once you have selected Docker, double-click to start it.



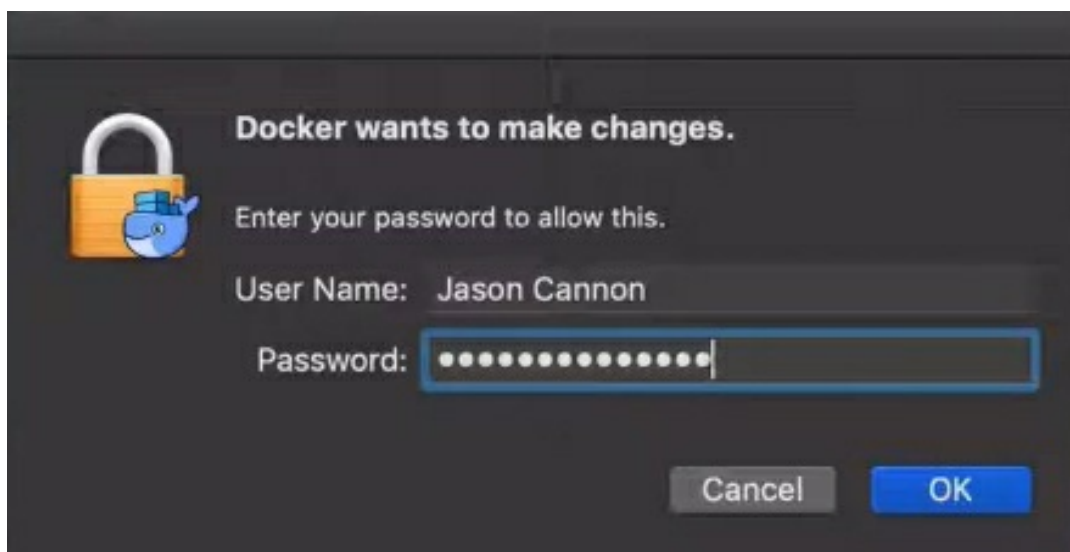
If you get a prompt that asks "Are you sure you want to open this file", click "Open". (Yes, you do want to open this file.)



If you are prompted about Docker Desktop needing privileged access, click "OK."



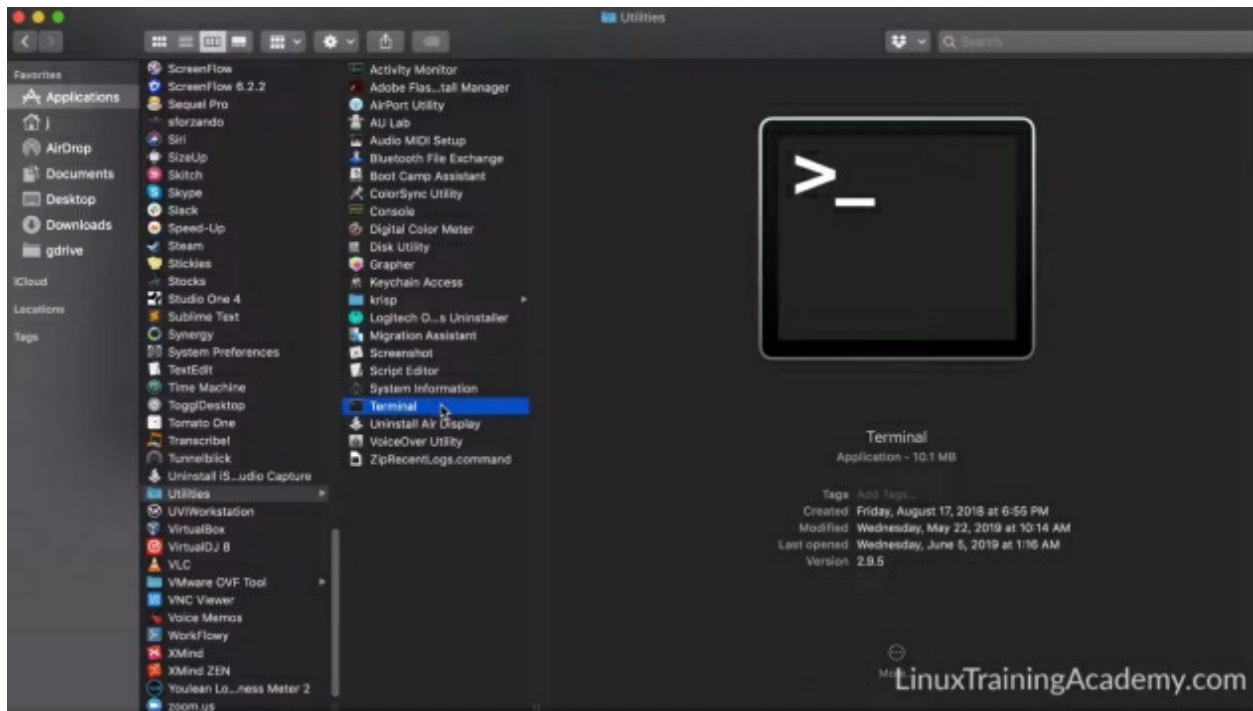
Enter your password when prompted.



Once Docker has started, you will see a message that says, "Docker Desktop is now up and running." Click the "X" to close this window.



To verify that Docker is working, use the command line. Open up the Terminal application, which is located in /Applications/Utilities. Double-click "Terminal" to start it.



Once you are at the command line, run the following command:

```
docker version
```

If Docker is installed and working properly, you will see the following information displayed on your screen:

```
j — -bash — 84x24
[jason@mac ~]$ docker version
Client: Docker Engine - Community
 Version:           18.09.2
 API version:       1.39
 Go version:        go1.10.8
 Git commit:        6247962
 Built:             Sun Feb 10 04:12:39 2019
 OS/Arch:           darwin/amd64
 Experimental:      false

Server: Docker Engine - Community
 Engine:
  Version:           18.09.2
  API version:       1.39 (minimum version 1.12)
  Go version:        go1.10.6
  Git commit:        6247962
  Built:             Sun Feb 10 04:13:06 2019
  OS/Arch:           linux/amd64
  Experimental:      false
[jason@mac ~]$
```

Installing Docker on Ubuntu and Debian

In this section, you will learn how to install Docker on Ubuntu, Debian, as well as other Debian-based distributions. If you want to install Docker on another Linux distribution, such as RedHat or CentOS, or another operating system altogether, please proceed to the appropriate section.

First, become the root user. If a root password is set on your system, and you know that root password, then you can become the root user by typing the following command:

```
su -
```

If the "su" method doesn't work, try using sudo.

```
sudo -i
```

To confirm that you are currently the root user, run the following command:

```
whoami
```

If anything other than "root" is displayed, you are not root. Again, it's important that you become the root user, as root privileges are required to perform the installation.

To ensure a clean installation of Docker CE, remove any older versions of Docker from your system to prevent potential issues.

By the way, if you have used Docker in the past, and you wish to retain your data, make a backup of the `/var/lib/docker` directory. This is where your Docker data is stored. When you uninstall Docker, it *should* keep the `/var/lib/docker` directory intact. However, if something goes wrong, it's better to have a backup. If you have not used Docker on this system before, or you don't need your old Docker work preserved, then proceed without performing a backup.

Now uninstall any existing or old versions of Docker that might be on your system.

```
apt remove docker docker-engine docker.io containerd runc
```

You can safely ignore any warnings or errors from this command. You will likely see warnings if no older versions of Docker were installed on your system.

Configuring Docker Repositories

Even though there are a few different ways to install Docker, Docker recommends using their software repositories.

Make sure that all the packages are up-to-date on the system by running:

```
apt update
```

The "apt update" command updates the list of available packages and their versions.

Next, update all the packages on the system by running:

```
apt upgrade -y
```

The "apt upgrade" command updates the installed software to the latest versions.

Next, install the dependencies – or requirements – for Docker. Often these packages are installed, but just in case they aren't, let's make sure to install them.

```
apt install -y apt-transport-https ca-certificates curl  
apt install -y gnupg-agent software-properties-common
```

Next, install the GPG key provided by Docker to help verify the identity of its service.

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

Now check that you've installed the correct key with this command:

```
apt-key fingerprint 0EBFCD88
```

This command searches fingerprints, and you should verify that the last eight characters match "0EBF CD88". Ideally, you would match the entire fingerprint, but matching the right-most characters is sufficient. Here is the pertinent line of output from the "apt-key" command:

```
9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88
```

By the way, Docker publishes the fingerprint. You can refer to Docker's installation documentation to get the latest fingerprint if needed.

For Ubuntu, the Docker software repositories correspond to the code name of the distribution.

You can get that code name by running this command:

```
lsb_release -cs
```

The "-cs" option to the "lsb_release" command displays just the distribution's code name and no other information.

Enter the following command to add the repository. If you are using a CPU architecture other than AMD64 or x86_64, adjust the "arch=amd64" section of the command. For example, if you are using a 64-bit ARM CPU, change "arch=amd64" to "arch=arm64".

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

Note: Only press "Enter" or "Return" after typing all the characters in the preceding command. Despite the line breaks you may see in this document, please be aware that the entire command should be typed contiguously until the last character. Only then should you type "Enter" or "Return." For long commands such as this one, the command may wrap in your terminal. Even if the command continues on more than one viewable line, it is still one single command.

If an error arises, check your typing. Make sure that everything is in lowercase and there are no additional spaces.

The "add-apt-repository" command will most likely run an "apt update" for you. However, don't rely on this. Manually update the local packages index:

```
apt update
```

If you see errors such as the one displayed below, go back a few steps and re-run the curl command to install the GPG key properly.

```
# The following signatures couldn't be verified because the public key is  
not available: NO_PUBKEY 7EA0A9C3F273FCD8
```

Now you're ready to install Docker. The following command will install the Docker Engine, the Docker command line tool, and the container runtime.

```
apt install -y docker-ce docker-ce-cli containerd.io
```

Check that the Docker client is available and working by running any simple command such as this one:

```
docker version
```

If you see information about Docker, then congratulations! You've successfully installed Docker.

Before we finish this section, I want to share with you why we installed docker this way.

Package Manager Installations of Docker

A Docker package is available via the default Ubuntu package repository. However, this version tends to be behind the latest Docker CE edition in terms of features.

There is no doubt that running the package manager install command is much easier. However, some downsides to using the default Ubuntu version are that it is potentially less stable, lags in security updates, and may not contain the latest Docker features. For quick tests on a new system, you might use the package manager's version. However, it is recommended to follow the Docker CE installation path as outlined above for almost all installations, especially production installations.

Installing Docker on Red Hat Enterprise Linux and CentOS

In this section, you will learn how to install Docker CE, or Docker Community Edition, on CentOS 7. These steps also apply to Red Hat Enterprise Linux (RHEL) as well. If you want to install Docker on another Linux distribution, such as Debian or Ubuntu, or another operating system altogether, please skip ahead to the appropriate section.

First, become the root user. If a root password is set on your system, and you know that root password, then you can become the root user by typing the following command:

```
su -
```

If the "su" method doesn't work for you, try using sudo.

```
sudo -i
```

To confirm that you are currently the root user, run the following command:

```
whoami
```

If anything other than "root" is displayed, you are not root. Again, you must become the root user, as root privileges are required to perform the installation.

To ensure a clean installation of Docker CE, remove any older versions from your system to prevent potential issues.

By the way, if you have used Docker in the past, and you wish to retain your data, please make a backup of the `/var/lib/docker` directory. This is where your Docker data is stored. When you uninstall Docker, the `/var/lib/docker` directory *should* still exist. However, if something goes wrong, it's better to have a backup. If you haven't used Docker on this system before, or you don't care if your old Docker work is preserved, then ignore this warning.

Now uninstall any existing or old versions on your system.

```
yum remove -y docker docker-client docker-client-latest docker-common  
docker-latest docker-latest-logrotate docker-logrotate docker-engine
```

Ensure that none of the above packages are installed before proceeding.

Of course, if you are working on a fresh installation of the OS, you most likely don't have to worry about removing previous versions of Docker. If at all possible, use a fresh installation of the OS that you plan to run Docker on.

Whether you're working on an existing or a newly installed system, make sure the system is up-to-date by executing the following commands:

```
yum update -y  
yum install -y yum-utils device-mapper-persistent-data lvm2
```

Now add the Docker repository so you have the required tools:

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

Next, install the required container packages:

```
yum install -y docker-ce docker-ce-cli containerd.io
```

If you are asked, check that the GPG key matches this one, barring the comment at the start of the line:

```
# 060A 61C5 1B55 8A7F 742B 77AA C52F EB6B 621E 9F35
```

To ensure that the Docker Engine starts on boot, enable it with systemd.

```
systemctl enable docker
```

Finally, start up the service:

```
systemctl start docker
```

To check that the installation was successful, try a simple command:

```
docker version
```

Now you have a working copy of Docker CE on "CentOS 7"!

INTRODUCTION TO DOCKER

In this book, we will cover a wide range of topics that will allow you to use Docker effectively. There will be practical exercises that will help you try out your newly acquired skills.

With Docker installed, you're probably ready to jump right in. However, let's first go over some important concepts that you'll need to understand in order to use Docker effectively. What follows is a bit theoretical, but it will help you fully understand Docker.

Docker History

You would be forgiven for thinking that Docker invented containers, as the Docker name is synonymous with containers today. Actually, Linux containers have been around in various forms for a number of years, yet Docker popularized them in a way that meant they rapidly became the way that many organizations create and run software applications today.

In this section, we will first look at why Docker became popular from 2010 onwards. We will then cover helpful terminology, before diving into some concepts that explain how Docker works.

What is a Container?

Creating a Docker container might be described as creating a small software package that can run a particular application and its associated processes.

The container that you create becomes portable and can be run on Docker installations on all types of computers in the same way without fear of compatibility issues. In other words, a single container can take care of all the software dependencies for a specific application, entirely on its own. This is

why Docker refers to a container as a "standardized unit of software".

You will find that Docker refers to a container as being very specific to a task, or even a single process. For this reason, it's common to run several containers at once on a single host machine.

Benefits of Using Containers

There are many benefits to running containers, which is why they are popular.

One great benefit of using Docker to run your Linux containers is that it is not disruptive to a system in the same way as some packages can be after installation. Some software, for example, will leave configuration files in many places, which can be difficult to clean up afterwards. Also, each container can have its own version of software. This way, you won't end up with conflicting versions of software or libraries on your main system.

Docker is intentionally lightweight; it will run as happily on a medium-powered laptop as it will on a high-powered server. This makes development easy to do anywhere and, more importantly, with predictable results.

For example, when developers want to promote updated software to introduce a new feature, they can be certain that if a container works in their development environment, it will also work in other environments.

Unlike Virtual Machines, which include a fully installed Operating System, containers are lightweight and can be as small as just a few megabytes in size. This means that moving containers between servers is quick and easy. Storage capacity is also less of a concern.

As previously mentioned, it's common to run multiple containers on one server. The workloads that containers run are more targeted and specific than traditional workloads. This means that the number of servers required is reduced too, cutting down the number of host machines that are needed.

For more than a handful of containers though, there are so-called "orchestrators", which can help organize and run multiple containers, making sure they are performing different functions efficiently. These might be Kubernetes or OpenShift, for example. We'll revisit orchestrators later on and

demonstrate how to use Docker Swarm to handle multiple workloads in containers.

When Docker was launched, it introduced a number of accessible features that previous types of Linux containers had not packaged in quite the same way. Along with a neater, more sophisticated networking model, Docker also provides an easy-to-use image file format from which you can create container images. The container images, which are required to create and then run containers, were also kept in a centralized place. This "image registry" is called Docker Hub, where members of the public and vendors can share pre-built images with ease.

Docker also made innovations using the aufs filesystem when it launched. Aufs stands for "advanced multi-layered unification filesystem". This provides a much more flexible filesystem for containers, thanks to being multi-layered. Now, the "overlay2" filesystem is used in Docker, as it offers some performance improvements over aufs.

Differences between Virtual Machines and Containers

It is important to remember that there are key differences between Virtual Machines and Docker containers.

You would usually treat containers with a view to running a single task or process, whereas a Virtual Machine might have different applications present on a single Virtual Machine. There might be a web server and a database server, for example, along with their backup scripts, and so on.

Thanks to a container's small footprint, it's refreshing to see that they can be started much, much quicker than a Virtual Machine. This is because the container is using the underlying host computer's Operating System, and not starting its own whenever it launches.

Key Things to Remember

Later in this book, we will look at security concerns around containers. Docker has made continuous improvements in this area, but it is important to note that it relies heavily on how it isolates containers from one another.

Multiple containers on a host machine will always share the kernel of the host

machine. This is efficient, but can be a potential problem if a container is attacked and compromised, which may affect other containers on the same host machine. Or, in the worst case, what is sometimes referred to as a container escape can occur, where a successful attack on a container might compromise the host machine that runs all the containers.

Docker's two main isolation techniques come directly from the Linux kernel development teams. They are called cgroups (control groups), which provide resource quotas. This can help limit one container from using up all the resources, such as the CPU or disk reads and writes on a host machine, to the detriment of other containers, and even the host itself. Additionally, namespaces are used at the kernel level to limit what each container can access within a host machine, as well as in other containers.

Terminology and Concepts

We have briefly touched on a few concepts in our introduction. Before running some commands in a terminal, let's reflect on these and some other common Docker definitions, beginning with the simplest concepts.

For example, we know that Docker uses containers to run a single process or a small group of processes to provide a service, such as a web server.

We now also know that we need a container image to start up a container, and that these images can be held in an online library, more commonly referred to as image registries, such as Docker Hub. There are various registries from different vendors, but in a business environment, a popular option is to run your own private registry to protect any commercially-sensitive data in your container images from unauthorized access.

In order to create a Docker image, a Dockerfile is required. This is a simple script that determines how an image is built from scratch. Expect to see more about Dockerfiles later on, but for now, here is a basic sample:

```
FROM debian:stable
LABEL author=LinuxTrainingAcademy
LABEL version=dockerbook

RUN apt update && \
```

```
apt install -y nano && \  
apt clean
```

```
CMD ["/bin/bash"]
```

Look closely at the layout of the Dockerfile. You can easily see how these image building configuration files are structured.

The layout of a Dockerfile is simple and logical. Once you use Docker to build an image from a Dockerfile, you can run as many identical containers from it as you wish.

All the details about creating and using Dockerfiles will be covered later in this book. This is just a small glimpse of what's going on behind the scenes when you begin using Docker.

Base Images and Layering

Base images are an important part of Docker and involve an image which you build your own image upon. Base images usually come with a tiny – but functional – Operating System.

Docker uses a layered filesystem that allows you to add to a base image without making your image much larger. Whenever you want to create a container image, a base image will almost always be used. Docker will know which base image to use when you specify a FROM line in the Dockerfile. If you refer back to the last image, you can see "FROM debian:stable" in the Dockerfile. The Debian stable image is the base image being used for your new image. The Debian base image is approximately 100 megabytes in size.

If you were to install a web server on top of that base image, you would only see the changes you had made, adding to the increase in size. These changes would be saved into a new layer of their own. If you are familiar with the "diff" command in Linux, which reports the difference between two files, you can think of this layering model as a similar concept or approach.

Microservices

Docker has become popular in part due to a new architectural approach to developing software and running applications. This microservices-approach

became popular at the same time as DevOps practices.

By loosely coupling parts of an application together, it is possible to make each service more modular. These modules can be referred to as "microservices". Decoupling formerly large, unwieldy services into individual units means that it is much easier to fix and test small functional parts of software if any issues arise.

New features can also be deployed without the need to redeploy the entire application again, since the architecture is less centralized. These days, a common approach is to use more sophisticated messaging to report on the critical elements of each microservice, which allows you to monitor for changes in the application.

As you have probably guessed, Docker containers are better suited to a microservice architecture than a traditional monolithic architecture because a container can run autonomously, without relying on other centralized processes.

Summary

In this chapter, you learned:

- A container is a standard unit of software that can run a particular application and its associated processes.
- It's common to run multiple containers on a single host, and that containers are more lightweight than virtual machines.
- How container images are necessary to start a container, and that those images are stored in a library called a registry.
- If you want to create your own Docker container images, you'll need to use a Dockerfile. You add to a base image in order to create your own Docker container images. Docker makes this possible by using a layered filesystem.

DOCKER BASICS AND COMMON COMMANDS

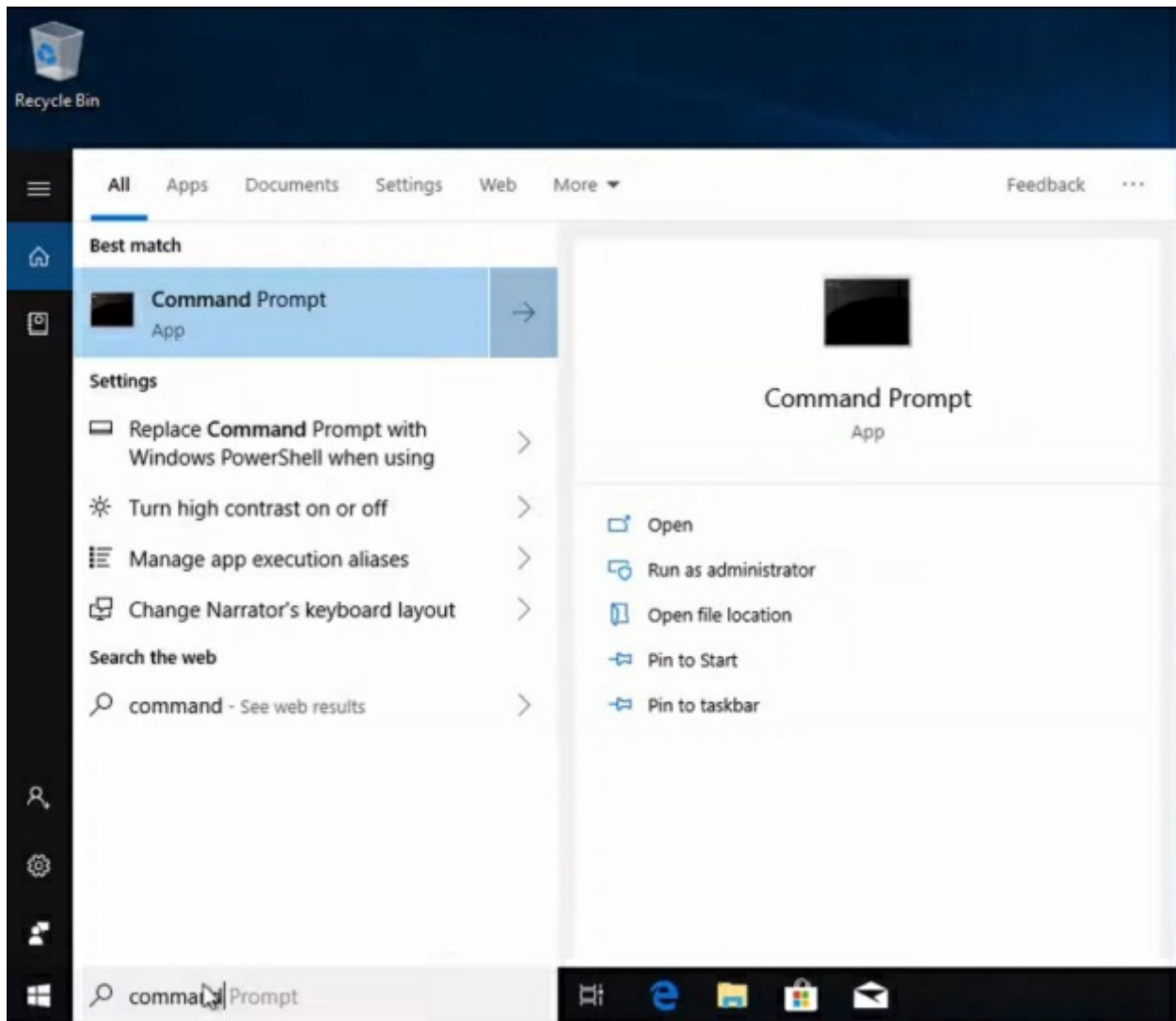
In this chapter, you'll learn how to start and stop docker containers. You will also learn how to display information about containers running on the host system, as well as container images downloaded to the host system. Finally, you'll learn how to use the built-in Docker help system.

At this point, you should have access to a system with Docker installed. If you don't, I highly recommend that you go back and install Docker now. You'll retain much more information by following along and working on the project exercises in the book.

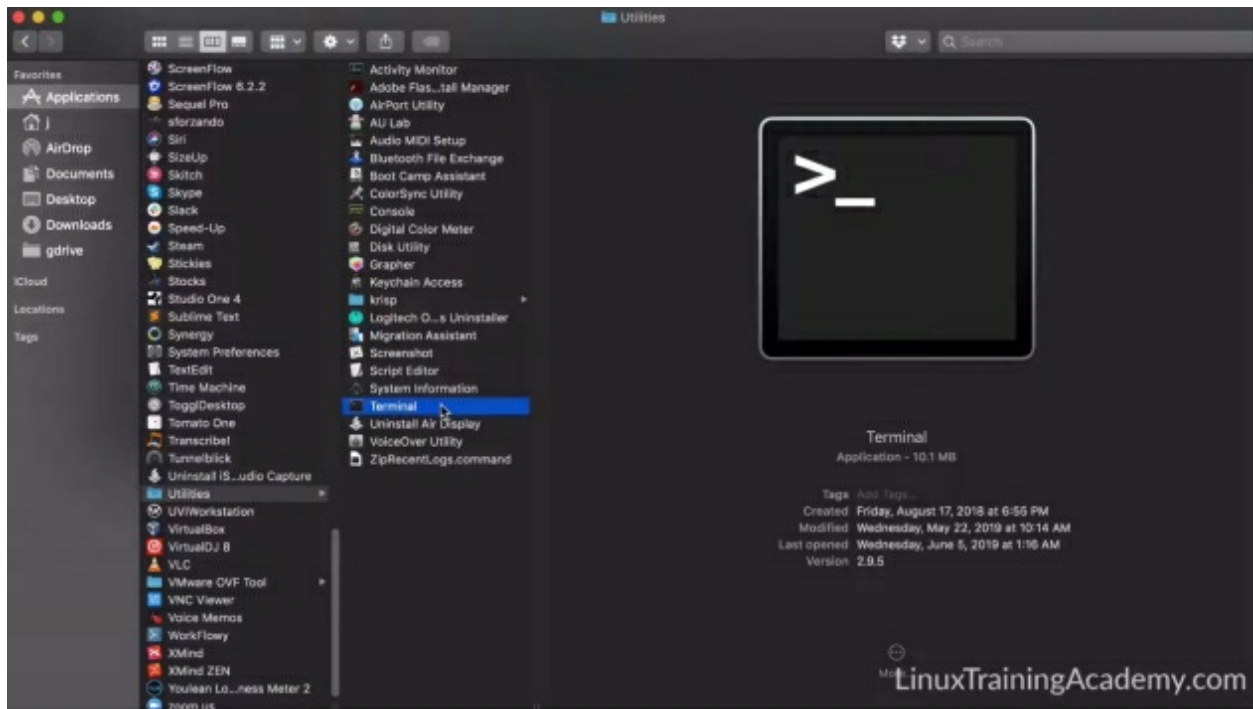
Starting a Terminal

Before we get to any of the Docker commands, you need to know that they work 100% exactly the same way, no matter if the Docker host system you are working on is running Linux, Windows or Mac. How to run Docker commands depends on your operating system.

If you have Docker installed on Windows, open up a command prompt as administrator. One way to do that is to click in the search bar and type in "command". Now you'll see "command prompt" and you'll want to run it as administrator. You can also use PowerShell and run that as administrator.



If you have Docker installed on a Mac, open up the Terminal application. The Terminal application is located in the /Applications/Utilities folder. Double click "Terminal" to start the application.



If you have Docker installed on Linux, start your favorite terminal emulator. Examples include GNOME Terminal, Konsole, and xterm.

Whether you're working from Windows Command Prompt, Windows PowerShell, Mac's Terminal application, or a terminal emulator on Linux, all the Docker commands are the same. Don't worry if my command line application looks different from yours; simply run the Docker commands, and you'll be fine.

Running a Container

As you'll find later in the book, the command used to run a container can become quite complicated. For now, we're going to keep things simple so that you can get an idea of how Docker runs containers.

We'll use the popular Linux distribution of Debian as an example of how to run a container. When you run a container, the image that you are using will be downloaded, if it hasn't been already. If the image already exists, then there is nothing to download. If the image isn't on your host system, Docker will download it for you.

You can run a Debian container with this command:

```
docker run -dit debian
```

Output:

```
root@ubuntu:~# docker run -dit debian
Unable to find image 'debian:latest' locally
latest: Pulling from library/debian
0bc3020d05f1: Pull complete
Digest: sha256:dcb20da8d9d73c9dab5059668852555c171d40cdec297da845da9c929b70e0b1
Status: Downloaded newer image for debian:latest
59d927c2f43eb3415dc2e306ffd26b60032ad7c7cfbcea84c546697d7dae9f7c
```

Notice that the first two lines of output state that Docker doesn't have the image, so it needs to pull down the image from the default image registry. That default image repository is Docker Hub.

After Docker downloads the image, it starts a container using that image. The last line of the output is the container ID. (Note that you will see a different container ID on your system when you run the command.) You can use the container ID to manage this particular container. Don't worry that it's a very long ID. As you'll soon see, you can use a small, but unique portion of that ID, or a human-friendly name.

Let's pause and discuss the options used here for the "docker run" command. The options "d", "i", and "t" are very important to know and understand. The "d" option stands for detach, and it allows a container to run in the background. When you run a container in the background, Docker will print out a container ID, as displayed in the image above.

The "i" option stands for interactive. This option keeps STDIN (standard input) open even if you're not attached to the container. This allows you to attach to a container and send standard input to it. In other words, it allows you to type commands into the container.

The "t" option is closely associated with the "i" option. The "t" option allocates a pseudo TTY or a pseudo terminal. Of course, if you want to interact with a shell, you'll need a terminal.

So, you know that your container started because Docker returned a container ID to you after you issued the "docker run" command. If you don't receive a

container ID, then something went wrong. The most common issue is a simple typing mistake. For example, here 'Debian' is purposely misspelled:

```
docker run debiann
```

Output:

```
Unable to find image 'debiann:latest' locally
docker: Error response from daemon: pull access denied for debiann,
repository does not exist or may require 'docker login': denied: requested
access to the resource is denied.
See 'docker run --help'.
```

In the error message, you'll see the phrase "the repository does not exist". This means that an image with the name you provided is not in the registry. If you do get an error, check your typing, as typos are the number one cause of an image not being found.

Show Running Containers

You can confirm that your container is running with the "docker ps" command:

```
docker ps
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
59d927c2f43e	debian	"bash"	29 seconds ago	Up 28 seconds	
nostalgic_cartwright					

Here, a number of details are displayed about the containers that are running on this host system. On the left-hand side, under the "CONTAINER ID" column, a name in the form of a hash twelve characters long has been assigned to the container. This makes it easier to tell Docker precisely which

container we want to reference. Under the "STATUS" column, we can also see how long the container has been running for.

You can find more information here, such as the image used to run the container, the command the container started with, when the container was created, its status, information about its ports, and finally, the human-readable container name.

If you want to stop a running container, look for the "CONTAINER ID" on the left-hand side and type the following command:

```
docker stop %%CONTAINER ID FROM "DOCKER PS"
COMMAND%%
```

In this particular example, the exact command would be:

```
docker stop 59d927c2f43e
```

You don't have to specify the entire container ID; just enough so Docker can tell exactly which container you want to manage. You can also stop the container by using its name. Try starting another container and stop it by using its name.

```
docker run -dit debian
docker ps
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
799d1c781bc5	debian	"bash"	15 seconds ago	Up 14 seconds	jolly_dewdney

Instead of typing out the long name Docker gave to this container, just type the first few characters and hit the TAB key. This is called tab completion, and it can be very handy.

```
docker stop jol<TAB>
```

If you check the output of "docker ps", no containers are listed, as no containers are currently running.

```
docker ps
```

Before we move on to some other Docker commands, I want to point out something that can confuse people who are new to Docker. As an example, this is how NOT to run a Docker container:

```
docker run debian
```

Notice that I didn't supply any options, and that we didn't receive a container ID. Let's look at the "docker ps" command to show running containers:

```
docker ps
```

You can see that there are no containers running. So what happened when I typed "docker run debian"? The container started and then immediately stopped. This is why the "d", "i", and "t" options were used. This way, the container would run detached in the background and not immediately exit.

Now, let's run a container the right way:

```
docker run -dit debian
```

Here, I've combined all three options after a single dash. This is nothing unique to Docker, and you're probably already familiar with this concept of using options. However, you might see other people doing it this way:

```
docker run -d -i -t debian
```

Again, this is the same command as "docker run -dit debian", but the options have been defined one at a time, instead of all together.

You might see this as well:

```
docker run -it -d debian
```

You will often notice people grouping "i" and "t" together. Note that I changed up the order on this command to demonstrate alternative ways of doing the same thing, as you would find in other documentation, for example.

Now, let's check the containers we have running.

```
docker ps
```

Output:

CONTAINER ID	IMAGE NAMES	COMMAND	CREATED	STATUS	PORTS
a485dd2fb2dd	debian	"bash"	7 seconds ago	Up 6 seconds	dazzling_ramanujan
c7810f3f7dec	debian	"bash"	55 seconds ago	Up 54 seconds	agitated_wu
8380b5341f1d	debian	"bash"	4 minutes ago	Up 4 minutes	wonderful_elgamal

Displaying Local Container Images

Another common command you'll use is displaying what images have been downloaded locally by Docker.

```
docker images
```

Output:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
debian	latest	7a4951775d15	3 weeks ago	114MB

From the output, you can see a number of columns. For the moment, the ones to pay attention to are: 1) the name which is currently simply "debian" (be aware that naming can get relatively complicated, so we'll examine that in more detail later on), and 2) the "SIZE" column showing how much disk

space the image is using on our host machine.

Inspecting Resources

Let's take a closer look at the complexities of a running container. We will use the "inspect" command and give Docker the hash of the container that we're interested in examining. The hash is always unique on a single Docker system. If no other containers start with the same first few characters of the hash, you can abbreviate the hash to refer to a container. For ease, you might use the first three characters. Let's inspect one of our running Debian containers with this command:

```
docker inspect %%FIRST THREE HASH CHARS OF CID%%
```

In this particular example, the exact command could be:

```
docker inspect a48
```

Example truncated output:

```
[
  {
    "Id": "a485dd2fb2dd62cc966a68ec08979c311968c784154152572a1305b7fec3f4db",
    "Created": "2021-07-17T09:08:16.103437624Z",
    "Path": "bash",
    "Args": [],
    "State": {
      "Status": "running",
      "Running": true,
      "Paused": false,
      # Output continues...
```

The information displayed can be very useful for troubleshooting. For example, you can see the networking details and various kernel settings for that particular container. We will look more closely at this later in the book.

Getting Assistance

One final thing to note about Docker is the way that it provides help on the command line. Similar to other applications you may have used, you can type this command to see the top most level of the help menu:

```
docker --help
```

To get access to a subcommand's help, you can simply add that subcommand to the same command. For example, to get help on the images Docker subcommand, enter the following:

```
docker image --help
```

If there are other options or further subcommands, you can get help with those as well.

```
docker image prune --help
```

Not all commands work in quite the same way, but when you are learning Docker, you are likely to find the well-written help menus extremely useful.

Summary

This chapter introduced you to some of Docker's basic commands. You learned:

- How to start and stop docker containers.
- How to display information about containers running on the host with the "docker ps" command.
- How to gather detailed information about a specific container with the "docker inspect" command.
- How to list the container images that have been downloaded to the host system.
- How to use the built-in Docker help system.

MANAGING DOCKER CONTAINER IMAGES

In this chapter, you will learn how to download an image, how to view its history, how to tag and delete an image, and how to view and clean up the storage being used by Docker for images.

Layering Model of a Container

Let's begin by looking at the layer model that Docker uses for its container images. You may remember from a previous section that the layering model is comparable to the Linux "diff" command. A container image takes up less disk space because it keeps track of the changes made to an image and then saves those changes into a layer of its own.

If, for example, all of your Docker images were based on the Debian Linux image, then newly created images would only take up storage space for what you've added to the base Debian image. The layers in an image are read-only, and each new layer is a delta – or difference – of any changes compared to the layer beneath it.

You can think of layering in the same way that the revision control system Git works, where each time a change to the codebase is made, only that commit is saved directly, but it will always reference the entire codebase. Not only does the layering model help with storage space, it also significantly speeds up rebuilding images after a small change has been made. This is because caching can be used.

In some cases, you want to limit the number of layers in an image, for the sake of simplicity. There's an easy way to do that in Dockerfiles, which you'll learn about shortly.

Downloading Images

In order to take a closer look at the layers present in an image, we can use the popular "nginx" web server as an example. First, download the image by using the "docker pull" command:

```
docker pull nginx
```

The "docker pull" command takes an image name as an argument, and it downloads that image to the local host system. You might remember from earlier that when we executed a "docker run" command, Docker checked to see if it needed to download or pull the image to the local host system first. Here, we are explicitly downloading the image without creating a container based on that image.

Once the image has been downloaded, we can check its presence on the system with this command:

```
docker images
```

Output:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nginx	latest	4cdc5dd7eaad	11 days ago	133MB

You can see how the image was constructed by using the "docker history" command.

```
docker history nginx
```

Output:

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
4cdc5dd7eaad	11 days ago	/bin/sh -c #(nop) CMD ["nginx" "-g" "daemon...	0B	
<missing>	11 days ago	/bin/sh -c #(nop) STOPSIGNAL SIGQUIT	0B	
<missing>	11 days ago	/bin/sh -c #(nop) EXPOSE 80	0B	
<missing>	11 days ago	/bin/sh -c #(nop) ENTRYPOINT ["/docker-entr...	0B	
<missing>	11 days ago	/bin/sh -c #(nop) COPY file:09a214a3e07c919a...	4.61kB	
<missing>	11 days ago	/bin/sh -c #(nop) COPY file:0fd5fca330dcd6a7...	1.04kB	
<missing>	11 days ago	/bin/sh -c #(nop) COPY file:0b866ff3fc1ef5b0...	1.96kB	
<missing>	11 days ago	/bin/sh -c #(nop) COPY file:65504f71f5855ca0...	1.2kB	
<missing>	11 days ago	/bin/sh -c set -x && addgroup --system -...	63.9MB	
<missing>	11 days ago	/bin/sh -c #(nop) ENV PKG_RELEASE=1~buster	0B	

```
<missing> 11 days ago /bin/sh -c #(nop) ENV NJS_VERSION=0.6.1 0B
<missing> 11 days ago /bin/sh -c #(nop) ENV NGINX_VERSION=1.21.1 0B
<missing> 3 weeks ago /bin/sh -c #(nop) LABEL maintainer=NGINX Do... 0B
<missing> 3 weeks ago /bin/sh -c #(nop) CMD ["bash"] 0B
<missing> 3 weeks ago /bin/sh -c #(nop) ADD file:4903a19c327468b0e... 69.2MB
```

The output shows each layer, line-by-line, along with the abbreviated commands that were executed to create each layer. Don't be concerned that some of the hashes are being reported as "missing" in the output of that command. This is normal, and is related to the build cache that Docker used to make that image locally unavailable, as we have pulled it from a remote repository.

In a later section, we'll re-visit Dockerfiles and exactly how to build your very own custom images. The goal here is to slowly introduce you to the concepts and commands, and then dive into more detail later on. You're still building your knowledge of Docker and how it works from the ground up.

Viewing Full Image Hashes

Run this command:

```
docker images --no-trunc
```

Output:

REPOSITORY	TAG	IMAGE ID
nginx	latest	sha256:4cdc5dd7eaadff5080649e8d0014f2f8d36d4ddf2e
	133MB	
nginx	1.15-	
alpine	sha256:dd025cdf837e1c6395365870a491cf16bae668218edb07d85c	
	ago 16.1MB	

The "--no-trunc" option tells Docker not to truncate the output. As you can see, Docker did not trim the length of the SHA256 hashes, as it did in your previous "docker images" command. This means that you can see the actual complete name that Docker uses internally to reference the images.

Remember, you can reference a resource's hash with just the first few characters, if the beginning of the hash is locally unique. This can save you quite a lot of typing. For obvious reasons, do this carefully to ensure that you

select the right resource.

Image Tags

You might have noticed the "TAG" column that was displayed in the last command. A tag is used to convey useful information. In most cases, the tag will tell you the version of an image.

If no tag is specified, the default tag "latest" is used. Note that "latest" is in all lowercase letters. Again, because a tag is meta-data – or information provided by a human at some point – it doesn't necessarily mean that this is the absolute newest version of that image. For example, there could be a newer version of an image that is currently being tested, but hasn't yet been approved for widespread use. Once the testing for that image is complete, it would then be tagged "latest." If you're familiar with Git, then you know that a Git tag refers to a particular commit in the Git repository's history. It's the same with Docker images; the tag points to a specific version of an image.

If you want to create a tag for a given image, use the "docker tag" command. Let's see what the built-in Docker's help system has to say about this command:

```
docker tag --help
```

Output:

```
Usage: docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]
Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
```

Here you can see that the syntax of the "docker tag" command is: "docker tag", followed by the source image, optionally followed by a colon. This is followed by a tag name, and finally a target image, also optionally followed by a colon and a tag name.

When reading help output or man pages, anything that appears in brackets is optional. As you have already learned, if no tag is specified, then the "latest" tag is assumed. That's why "[:TAG]" appears in brackets (":TAG") and is thus optional.

You can now try to create a tag of your own. Let's say that you've tested this particular nginx image, and it's working perfectly for your needs. You want to make sure you can use this exact image for future deployments. To do that, tag that image with something meaningful to you. Pretend you are using this image to host your blog; you can tag it "myblog_stable".

```
docker tag nginx:latest nginx:myblog_stable
```

No output is created by the "docker tag" command. To view the result of the command, view the images.

```
docker images
```

Output:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nginx	latest	4cdc5dd7eaad	11 days ago	133MB
nginx	myblog_stable	4cdc5dd7eaad	11 days ago	133MB

If you look closely, you will see that the "IMAGE ID" column shows the exact same hash for the nginx:latest image and the nginx:myblog_stable images. In other words, Docker hasn't used up an additional 133 megabytes of disk space to re-tag the original image, but simply created an alias to the original.

When we come to look at image registries in a later chapter, you will see a number of different tags, so it's important to be able to search for them. Without learning any new commands, even the Linux "grep" command can be used to search. If you prefer to use a web browser, you can also search for the available tags for a particular image inside Docker Hub. Images and their tags can be found at <https://hub.docker.com>. (NOTE: The precise location of Docker Hub may change over time. If the link does not work when you attempt to access it, search for "Docker image library" or similar.)

Dockerfiles

We've mentioned Dockerfiles already, and we'll look a little closer at their contents now. Dockerfiles can range from being very simple to very complex

and lengthy configuration files. However, as long as you have a grasp of the basics, you will find them logical and relatively easy to understand when the need arises to use them.

Earlier, we looked at the many layers involved in a container image using the "docker history" command. Let's remind ourselves of what they look like, but from the perspective of a build file, or Dockerfile.

We'll use the popular nginx image again, which runs a lightweight web server. Similar to the tags available for an image, you can also find Dockerfiles on <https://hub.docker.com> for most popular images. Go to Docker Hub and search for nginx. Click on the top result. From there, click on a tag such as "latest". The link on Docker Hub to the Dockerfile for an image points to a file hosted on GitHub, which is very common.

I've created a simplified version of the Dockerfile for nginx, by just removing some lines so that we can focus on the concepts instead of getting lost in the specifics.

Abbreviated Dockerfile for nginx:

```
FROM debian:buster-slim
LABEL maintainer="NGINX Docker Maintainers <docker-maint@nginx.com>"

ENV NGINX_VERSION 1.21.1
ENV NJS_VERSION 0.6.1
ENV PKG_RELEASE 1~buster

RUN apt-get update \
    && apt-get install -y nginx \
    && apt-get clean

RUN ln -sf /dev/stdout /var/log/nginx/access.log \
    && ln -sf /dev/stderr /var/log/nginx/error.log

COPY index.html /var/www/html/

EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

By looking at an image's Dockerfile, you can see the exact commands that constructed that image. You can also quickly rebuild the container image from the Dockerfile with a few changes that you want to make yourself, knowing precisely what was changed from the original. This might involve reducing the resulting storage size of the built image by avoiding certain packages being installed into the image, or by subtly changing the way a process starts, or altering a file path, for example.

If we look at the top of the Dockerfile, the "FROM" instruction tells Docker what to use as the base image. By running a "docker build" command using this Dockerfile (it's sometimes easier to think of them as build files), that line means that a layer will be created on top of the base image in your image. Note that the version of Debian has the tag "buster-slim" shown after the colon.

The "LABEL" entry is a good way of offering contact details for the author.

You will also see "ENV" instructions, which introduce environment variables, like those that Bash would use.

The "RUN" instruction tells Docker to run a shell command to help build your image.

The "COPY" command copies a file from the local system into the image at build time. This COPY statement will put a copy of the index.html file in the /var/www/html directory. By the way, the COPY command is most commonly used to copy small configuration files into an image.

As you may have guessed, the "EXPOSE" entry is telling Docker to open up – or expose – TCP port 80 when a container runs from the image. Of course, port 80 is the standard port used to serve HTTP traffic.

Finally, the "CMD" instruction runs the nginx binary with two options from inside the container. The syntax being used here is actually a JSON (JavaScript Object Notation) array.

The "daemon off" option keeps nginx running in the foreground of the container. Without this addition, the container would stop running as soon as it is started!

As you can imagine, some applications require slight changes when they are containerized, and this is a good example. Most well-written applications support the required changes, just as nginx does.

Note that in our example Dockerfile, two ampersand characters (&&) are used in the "RUN" instructions. This is a standard shell convention to chain commands together. Specifically, the command following the double ampersands will execute only if the preceding command succeeds. Concatenating – or joining – multiple commands is a handy workaround to remember. This prevents Docker from adding a new layer for each of the commands in a Dockerfile.

Having fewer layers is generally – but not always – a good idea. In most cases, having fewer layers means that it will be quicker to build an image from a Dockerfile. There are a number of ways to improve the caching that Docker uses during its build time. For example, if you are currently tweaking the content at the end of a Dockerfile, you can tease apart longer "RUN" instructions and add them nearer to the start of the Dockerfile to speed up the build process.

Notice the "apt-get clean" command. It's good practice to add this command when working on Debian-based Linux distros, such as Debian and Ubuntu. For RPM based Linux distros, such as RedHat or CentOS, use "yum clean all". These commands delete the downloaded package files. Not doing this means those downloaded packages will continue to reside in your image and increase its size.

By the way, it's fairly common to chain the clean command to other apt commands with two ampersands, to ensure all the package manager's work is performed in one layer in an image.

Let's build in an image using this Dockerfile, so you can see what this process looks like. In a later section, we'll delve deeper into creating your own images, however, it's important to see the process now, since we're talking about images.

Before you perform the build, make sure you are in the same directory as the Dockerfile. Create an index.html file by running the following command:


```
echo 'Hi from inside the container!' > index.html
```

Now that you have an image, use the following command:

```
docker build -t mynginx .
```

The "-t" option allows you to specify a tag – or name – for the image. The period or dot, tells Docker to use the Dockerfile in the current directory.

Now that you have an image, run it as a container.

```
docker run -dit mynginx
```

Output:

```
ce795af52ea56a6796fcafa0333db1774f26d26a94c7815ad6e882e012210e5a
```

Note: The output is a container ID, and you will see a different container ID when you execute the command.

View the running container.

```
docker ps
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
ce795af52ea5	mynginx	"nginx -g"	2 minutes ago	Up 2 minutes	80/tcp
ecstatic_lumiere					

To view all the steps and layers used in creating the image, run this command:

```
docker history mynginx
```

You can correlate the history of the image with the contents of the Dockerfile.

Deleting Images

We looked at listing images locally, but haven't discussed how to delete them to save space or tidy up old versions. Let's list our current images again to see the two versions of "nginx" with this command:

```
docker images
```

Output:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mynginx	latest	e66c6e8c939d	11 days ago	133MB
nginx	latest	4cdc5dd7eaad	11 days ago	133MB
nginx	myblog_stable	4cdc5dd7eaad	11 days ago	133MB
debian	buster-slim	7c5871f12659	3 weeks ago	69.2MB

Here, there is a Debian image tagged with "buster-slim" in the output. The reason that image exists is that the Dockerfile we used to create the mynginx image contained "FROM debian:buster-slim". The mynginx image is based on debian:buster-slim. Docker pulled down the debian:buster-slim image first, so that layers could be created on top of it to end up with the resulting mynginx image.

Let's remove the "myblog_stable" version of nginx that we created earlier as a test. The command to do that is "docker rmi" followed by the image name.

```
docker rmi nginx:myblog_stable
```

Output:

```
Untagged: nginx:myblog_stable
```

Docker reports back that the image, with the tag "myblog_stable" after the colon, has been untagged, but not deleted. You may remember that this is because it was an alias.

Display your images again.

```
docker images
```

Output:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mynginx	latest	e66c6e8c939d	11 days ago	133MB
nginx	latest	4cdc5dd7eaad	11 days ago	133MB
debian	buster-slim	7c5871f12659	3 weeks ago	69.2MB

You can try the same with the image you downloaded from Docker Hub, using this command:

```
docker rmi nginx:latest
```

Output:

```
Untagged: nginx:latest
Untagged:
nginx@sha256:353c20f74d9b6aee359f30e8e4f69c3d7eaea2f610681c4a95849a2fd7c497f9
Deleted: sha256:4cdc5dd7eaadff5080649e8d0014f2f8d36d4ddf2eff2fdf577dd13da85c5d2f
Deleted: sha256:63d268dd303e176ba45c810247966ff8d1cb9a5bce4a404584087ec01c63de15
Deleted: sha256:b27eb5bbca70862681631b492735bac31d3c1c558c774aca9c0e36f1b50ba915
Deleted: sha256:435c6dad68b58885ad437e5f35f53e071213134eb9e4932b445eac7b39170700
Deleted: sha256:bdf28aff423adfe7c6cb938eced2f19a32efa9fa3922a3c5ddce584b139dc864
Deleted: sha256:2c78bcd3187437a7a5d9d8dbf555b3574ba7d143c1852860f9df0a46d5df056a
```

There is a difference between this output and the previous one. As well as untagging the image, a number of layers were deleted. This means you have freed up disk space that was being used by that image. You can check this again by listing the images to make sure it's gone.

```
docker images
```

Output:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mynginx	latest	e66c6e8c939d	11 days ago	133MB
debian	buster-slim	7c5871f12659	3 weeks ago	69.2MB

If you attempt to delete an image and a container exists on the system that uses that image, then you will get an error message. If you are certain you no longer need the image, then you can use the "-f" option, which will force

delete the image. Here's an example of using the force option:

```
docker rmi -f nginx:latest
```

In this particular case, the image has already been deleted. Make sure to use caution when force deleting an image.

As a reminder, you can use "--help" to see the available options.

```
docker rmi --help
```

Output:

```
Usage: docker rmi [OPTIONS] IMAGE [IMAGE...]
```

```
Remove one or more images
```

```
Options:
```

```
  -f, --force    Force removal of the image
```

```
  --no-prune    Do not delete untagged parents
```

Clearing Disk Space

A slightly unusual terminology in Docker is used to describe image layers that aren't being used by tagged images. These unused layers are called 'dangling images', and you may see them listed as "<none>" when listing images.

In older versions of Docker, relatively long commands were required to identify which images and layers weren't associated with a container and could be deleted to free up space. Luckily, Docker simplified this process and introduced a prune subcommand. You can imagine the pruning of these layers as a form of garbage collection. Try it out:

```
docker image prune
```

Output:

```
WARNING! This will remove all dangling images.
```

```
Are you sure you want to continue? [y/N]
```

Docker tells you what it's about to do, which is removing dangling images, and asks you for confirmation. In this particular case, I don't want to remove the dangling images, so I'll type "N". I just wanted you to be aware of the "docker image prune" command.

Docker can help with tidying up all the Docker objects, but be very careful when doing this. For example, to remove both unused images and dangling images, you can use this command:

```
docker system prune -a
```

Output:

```
WARNING! This will remove:
```

- all stopped containers
- all networks not used by at least one container
- all images without at least one container associated to them
- all build cache

```
Are you sure you want to continue? [y/N]
```

Please use this command with caution! Running this command will only keep those images not currently being used by a container. To abort the operation, type "N".

The "-a" option stands for 'all'. Let's run that command again, but this time without the "-a" option:

```
docker system prune
```

Output:

```
WARNING! This will remove:
```

- all stopped containers
- all networks not used by at least one container
- all dangling images
- all dangling build cache

```
Are you sure you want to continue? [y/N]
```

Note the differences between this command and the one with the "-a" option. Again, to abort type "N".

If you want to see how much disk space Docker is using, use the "docker system df" command:

```
docker system df
```

Output:

TYPE	TOTAL	ACTIVE	SIZE	RECLAIMABLE
Images	2	1	147.8MB	69.24MB (46%)
Containers	1	1	7.897MB	0B (0%)
Local Volumes	0	0	0B	0B (0%)
Build Cache	4	0	31.19MB	31.19MB

You can think of "df" as standing for "disk free." You're probably already familiar with the Linux "df" command. This command shows you how much space is being used by Docker. So, if you see that you have a lot of disk space being used by images, this is a good starting point to do some pruning and remove any unused or unwanted images.

Pruning Disk Space Manually

If you want to uninstall Docker and reclaim as much disk space as possible, then delete the /var/lib/docker directory. Of course, only do this after you have backed up anything you want to keep. For example, you can copy any of your local images to a remote registry, to save for later use.

Summary

In this chapter, you learned:

- How to download as well as delete Docker images.
- How to view the history of an image.
- How layering works and that the way a Dockerfile is structured determines how many layers will be created for the resulting image.

- How to view the storage space being used by Docker.
- How to prune Docker objects to free up storage space.

EXERCISE: MANAGING IMAGES

Note: You can download all the exercises contained in this book at <https://www.LinuxTrainingAcademy.com/docker>.

Goal:

The goal of this exercise is for you to become familiar with downloading and deleting images. In this exercise, you'll focus on two popular applications: Memcached and Apache.

Instructions:

Download the Images

In order to download – or "pull" – the "memcached" container image, use the following command:

```
docker pull memcached:latest
```

(NOTE: the ":latest" tag is optional because if no tag is supplied, the "latest" tag is assumed.)

Next, download the "Apache" image. This image is actually called "httpd." You can determine the name of the image by searching for it on the command line with "docker search TERM", or by searching on <https://hub.docker.com>.

Download the "2-alpine" version of this image instead of the default "latest" image:


```
docker pull httpd:2-alpine
```

This image contains Apache version 2, running on the Alpine Linux distribution as its base. Alpine is heavily used in containers, due to its small size.

If that tag becomes outdated and doesn't work for you, visit this page on Docker Hub to get a working tag: https://hub.docker.com/_/httpd?tab=tags

Check that the Images are Present

List the two images that you downloaded:

```
docker images
```

Check to see how much disk space all images combined are using.

```
docker system df
```

Delete the Images

Now delete each of the two images.

To delete Memcached, use this command:

```
docker rmi memcached
```

Note, there is no tag because you have the "latest" tag of Memcached.

For Apache, you have to declare the tag explicitly, or the "rmi" command won't work.

```
docker rmi httpd:2-alpine
```

Check that the Images are Deleted

Make sure the images aren't present with this command:

docker images

RUNNING AND MANAGING DOCKER CONTAINERS

Running Containers

In this chapter, you will learn the most important options for the "docker run" command. You'll learn how to give a container a name and then manage that container using its name. Additionally, you'll learn how to view a container's state, regardless of whether it is currently running or if it has stopped. You'll learn how to configure a container so that it restarts any time it exits, or even if the host system gets rebooted. Next, you'll learn how to view the output generated by a container that is running in the background. Finally, you'll learn how to quickly clean up and remove old containers that have already stopped.

Core Concepts

Sometimes, those who are new to Docker attempt to run a container only to find that it doesn't appear to be running! In those cases, the container starts and then immediately stops. Most people want to start a container and keep it running in the background. You need to explicitly tell Docker to do so.

Docker treats the software inside a container just like any other system process. Once the process has completed, Docker will then stop the container. Therefore, you have to "detach" it to keep it running in the background.

Running in the foreground by default has its advantages if a container is doing a specific single task, such as downloading a file, processing it and saving it somewhere. However, a container will usually keep running and provide some sort of ongoing service or processing. It might help you to remember when to use it, by thinking of the "-d" option used for "detaching"

as setting a container to "daemonize".

You may remember this command from earlier:

```
docker run -dit debian
```

Note that we used the "d" option along with the "i" and "t" options. These options mean “allow interactivity” and “assign a TTY”, respectively. In other words, using the "i" and "t" options together allows for an interactive shell to be made available to access the container. It’s common to use the "dit" options together.

Naming Containers Correctly

Earlier in the book, we mentioned referencing containers using their hash, but that isn't exactly human friendly. It's similar to using IP addresses; instead of remembering a string of numbers, we deploy DNS so that us humans can use a name to access an IP address. Docker allows us to do something similar. You can give a container a name and refer to it this way, instead of its container ID. This means you can give your container a meaningful name.

Let’s run a Debian container now with its own name and compare the command to the previous example. I'm going to pretend that this container will provide a web service, so I'm going to name it "web". Again, the name is completely up to you, so you can choose something that is meaningful and/or easy for you.

```
docker run -dit --name=web debian
```

You can use "--name=web" or "--name web" and get the same result. Both are valid syntax.

The main thing to note in this example is the image name "debian" is at the end of the command.

Check that the container is running with this command:

```
docker ps
```

Output:

CONTAINER					
ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
NAMES					
4c4abfbe221a	debian	"bash"	29 seconds ago	Up 28 seconds	
web					

As you can see in the "STATUS" column, the container is running successfully and has been for some time. You'll also see "web" in the "NAME" column. This is because we specified "--name=web" in the "docker run" command.

If you also wanted to check stopped containers, use the "-a" option, which stands for "all".

```
docker ps -a
```

To show you another useful command, let's start a second Debian container using a different name:

```
docker run -dit --name=third_container debian
```

You can list the latest container that was started with this command:

```
docker ps -l
```

Output:

CONTAINER					
ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
NAMES					
767f12ae3e8e	debian	"bash"	9 seconds ago	Up 8 seconds	
third_container					

The "-l" option stands for "latest" and it shows us the latest created container, regardless of its state. That means when you run "docker ps -l", it will give you information on the latest container that is running or that has already stopped.

To show both containers that are running, use the "docker ps" command again:

```
docker ps
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
767f12ae3e8e	debian	"bash"	38 seconds ago	Up	37 seconds
third_container					
4c4abfbe221a	debian	"bash"	2 minutes ago	Up	2 minutes
web					

If you're having issues with the syntax of the "docker run" command, work backwards from the right-hand side. For example, if the debian image hadn't existed locally or in an image registry, such as Docker Hub, then our command would fail irrespective of the other options.

The "docker run" command can become quite complex, but if you can remember the basic syntax above, it's possible to work around it, with a little trial and error. Not adding the "-d" option is also a very common mistake. Failing to do so will most likely result in a container starting and then immediately stopping.

Stopping a Container

Docker won't let you start another container with the same name of an already existing container. Let me demonstrate that now:

```
docker run -dit --name=web debian
```

Output:

```
docker: Error response from daemon: Conflict. The container name "/web" is  
already in use by container  
"4c4abfbe221a38239649580dfc46f016edaac8036616663018c01ba38fecb28"
```

You have to remove (or rename) that container to be able to reuse that name. See 'docker run --help'.

If you want to reuse the name, you will have to delete the old container. The command to delete a container is "docker rm".

First, stop your container.

```
docker stop web
```

Output:

```
web
```

You've now stopped your container, and can view it using the "-a" option to "docker ps".

```
docker ps -a
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
767f12ae3e8e	debian	"bash"	24 minutes ago	Up	24 minutes	third_container
4c4abfbe221a	debian	"bash"	29 minutes ago	Exited (0)	57 seconds ago	web

Now remove the stopped container named "web".

```
docker rm web
```

Output:

```
web
```

Unlike deleting an image, use "rm" to delete a stopped container, rather than

"rmi". Again, this has the same name as the "rm" Linux command used for removing files.

List all the containers again.

```
docker ps -a
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
767f12ae3e8e	debian	"bash"	25 minutes ago	Up	25 minutes
third_container					

Now there are no stopped containers with the name "web". The previously used name is now freed up. At this point, you can start another container with that same name.

By the way, you can also start a stopped container using the "docker start" command. This can be useful to add some more "run" command options to a stopped container. Let me demonstrate by restarting a container:

```
docker run -dit --name=web debian
```

Output:

```
5f8065b6dec65fd32520de70b771219e8a6d52e6b7a8b1e7a46fec2230dd2587
```

Now stop the running container.

```
docker stop web
```

Output:

```
web
```

List all the containers.

```
docker ps -a
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
5f8065b6dec6	debian	"bash"	2 minutes ago	Exited (0)	2 minutes ago	web
767f12ae3e8e	debian	"bash"	31 minutes ago	Up	31 minutes	third_container

Now you can restart the web container.

```
docker start web
```

Output:

```
web
```

List the containers to see that it is running again.

```
docker ps
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
5f8065b6dec6	debian	"bash"	4 minutes ago	Up	16 seconds	web
767f12ae3e8e	debian	"bash"	31 minutes ago	Up	31 minutes	third_container

As you're done with this container for now, you can clean up by removing it. First, it needs to be stopped.

```
docker stop web
```

Output:

```
web
```

Now it can be removed.

```
docker rm web
```

Output:

```
web
```

Always Restart Policy

Imagine if the host machine running a Docker Engine rebooted, or if the Docker Engine itself restarted. There would likely be some containers that you would want to automatically restart. Equally, if a container crashed for some reason, you might want it to restart on its own instead of waiting for a human to manually restart it.

Docker offers that functionality with the "--restart" option:

```
docker run -dit --restart=always --name=fourth_container debian
```

You can use "--restart=always" or "--restart always" and get the same result. Both are valid syntax.

Output:

```
dc85aff323918234ccde68e73c37a1c8970b113c7db709e5cf9ebf52ad4a0d85
```

You can inspect a container to check that the restart policy has been applied. Note that you can also use the name of the container instead of its hash. Any command that takes a container ID also accepts the container's name.

```
docker inspect fourth_container | grep -A3 RestartPolicy
```

By the way, the "-A" option of "grep" tells grep to not only print the matching line, but also a number of lines after the match. So here, grep prints the line that matches "RestartPolicy", as well as the next three lines.

Output:

```
"RestartPolicy": {  
  "Name": "always",  
  "MaximumRetryCount": 0  
},
```

You can also pipe the output to the "less" command, to allow you to easily page through the output and search for a keyword.

```
docker inspect fourth_container | less
```

We will look at the "docker inspect" command's formatting more closely later, but the output from the command shows that the container has an "always" restart policy applied to it.

If a container stops for any reason, and no restart policy was specified, the container will remain stopped since the default restart policy is "no". If the host machine is rebooted, the container will not start after the system becomes available again. If the container stops due to an error, it will remain stopped. If the Docker Engine restarts, the container will remain stopped. Using the default restart policy, or not supplying a restart policy, means that a stopped container can only be restarted manually.

The "on-failure" restart policy will restart a container that stopped due to an error. Specifically, if a container with an "on-failure" restart policy exits with a non-zero exit status, Docker will automatically restart it.

The "unless-stopped" policy behaves much the same way as the "always" policy. The only difference is that if a container with the "unless-stopped" policy has been manually stopped before a system reboot or a Docker Engine restart, it will remain stopped when the system boots or the Docker Engine starts again. In the case of a container with the "always" restart policy, the container would be restarted upon system reboot or Docker Engine restart, even if it was manually stopped at the time of reboot or Docker Engine

restart.

You will find that the "always" or "unless-stopped" restart policy is used in the overwhelming majority of production use cases.

Like other commands, if you get stuck with the "docker run" command, you can use the "--help" option.

```
docker run --help
```

For easier navigation, you can also pipe the output to a pager.

```
docker run --help | less
```

Stopping a Container

As you've learned, you can use the "docker stop" command to stop a container by providing either the name you gave it at launch time, or by its container ID, or hash:

```
docker stop fourth_container
```

Output:

```
fourth_container
```

Occasionally, you might get an error when you attempt to stop a container. The explanations tend to be relatively straightforward, and you'll be able to fix them. However, sometimes you will need to use the "kill" command to force a container to stop:

```
docker kill third_container
```

Output:

```
third_container
```

The "kill" command can save the day, but be warned that it skips the graceful shutdown aspect that the "docker stop" command provides. It's rare, but by

using "docker kill" you might put your container in such a state that it will not be able to restart. Most of the time this doesn't matter, as containers are typically built to run in such a way as to make them disposable. We'll cover how to keep your data safe in a later chapter.

Tidying Stopped Containers

You can use the "docker system prune" command to delete stopped containers. This can be easier than manually running "docker rm" for each stopped container. When prompted, answer "Y", to allow Docker to perform the pruning operation.

```
docker system prune
```

Output:

```
WARNING! This will remove:
```

- all stopped containers
- all networks not used by at least one container
- all dangling images
- all dangling build cache

```
Are you sure you want to continue? [y/N] y
```

```
Deleted Containers:
```

```
dc85aff323918234ccde68e73c37a1c8970b113c7db709e5cf9ebf52ad4a0d85  
767f12ae3e8e37e20561c478138b89d6e87390cfdc013f3fab95ea0fa8445258
```

```
Total reclaimed space: 0B
```

Confirm that all the containers have been removed.

```
docker ps -a
```

Output:

CONTAINER							
ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES	

Automatically Deleting a Stopped Container

If you want to automatically tidy up after yourself, use the "--rm" option, in conjunction with the "docker run" command. Once the main process within a container has completed, Docker automatically deletes the stopped container when you use the "--rm" option.

Here's an example: I'm going to use an image called "hello-world". Docker sometimes uses this image for demonstrations in some of its documentation. I don't have a copy of the image locally, so Docker will download it before it launches the container.

```
docker run --rm hello-world
```

Output:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
109db8fad215: Pull complete
Digest: sha256:df5f5184104426b65967e016ff2ac0bfcd44ad7899ca3bbcf8e44e4461491a9e
Status: Downloaded newer image for hello-world:latest
```

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
(arm64v8)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:

<https://hub.docker.com/>

For more examples and ideas, visit:

<https://docs.docker.com/get-started/>

View a list of containers.

```
docker ps -a
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
-----------------	-------	---------	---------	--------	-------	-------

Notice that the container doesn't exist at all, either in a running or a stopped state. The container ran, produced some output, stopped, and then Docker deleted it because it stopped.

Now I'll try it without the "--rm" option:

```
docker run hello-world
```

Output:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (arm64v8)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent
   it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

View a list of containers.

```
docker ps -a
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
--------------	-------	---------	---------	--------	-------

```
NAMES
```

```
5318170d8dc2 hello-world "/hello" 36 seconds ago Exited (0) 35 seconds ago  
priceless_northcutt
```

Some people like using the "--rm" option to help with garbage collection and keeping things tidy on their systems.

When I executed the "docker run hello-world" command, the container ran in the foreground, and we saw all the output created by the container. But how can you see that output if you run the container in the background? Well, the answer is to use the "docker logs" command.

First, start a container in the background using the hello-world image.

```
docker run -dit hello-world
```

Output:

```
911ebd5a66b85860f7dc90ca9ebdf729dba5f7853b5ac83b72abb9af0c392041
```

The returned string is the container ID or container hash. It can be used to see the output generated by the container. The syntax is "docker logs", followed by enough of the container ID so that it is unique on the local system.

Typically, you can use the first three or four characters of the ID. In this example, with this specific container ID, you would run:

```
docker logs 911e
```

Output:

```
Hello from Docker!
```

```
This message shows that your installation appears to be working correctly.
```

```
To generate this message, Docker took the following steps:
```

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.

```
(arm64v8)
```

3. The Docker daemon created a new container from that image which runs

the

executable that produces the output you are currently reading.

4. The Docker daemon streamed that output to the Docker client, which sent it

to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:

<https://hub.docker.com/>

For more examples and ideas, visit:

<https://docs.docker.com/get-started/>

As always, you can see the options available to you by requesting a little help from Docker.

```
docker logs --help
```

The "-t" option is handy here, as it provides timestamps of when the output was generated. Without the "-t" option, you may not know at what time the output was generated. This is useful information when troubleshooting or investigating.

```
docker logs -t 911e
```

Output:

```
2021-07-20T07:18:05.503042220Z Hello from Docker!
```

```
2021-07-20T07:18:05.503045095Z This message shows that your  
installation appears to be working correctly.
```

```
2021-07-20T07:18:05.503047220Z
```

```
2021-07-20T07:18:05.503049262Z To generate this message, Docker took  
the following steps:
```

```
2021-07-20T07:18:05.503051178Z 1. The Docker client contacted the  
Docker daemon.
```

```
2021-07-20T07:18:05.503053137Z 2. The Docker daemon pulled the
```

```
"hello-world" image from the Docker Hub.
2021-07-20T07:18:05.503055220Z (arm64v8)
2021-07-20T07:18:05.503057012Z 3. The Docker daemon created a new
container from that image which runs the
2021-07-20T07:18:05.503058970Z executable that produces the output
you are currently reading.
2021-07-20T07:18:05.503060803Z 4. The Docker daemon streamed that
output to the Docker client, which sent it
2021-07-20T07:18:05.503066012Z to your terminal.
2021-07-20T07:18:05.503068095Z
2021-07-20T07:18:05.503069887Z To try something more ambitious, you
can run an Ubuntu container with:
2021-07-20T07:18:05.503071803Z $ docker run -it ubuntu bash
2021-07-20T07:18:05.503073595Z
2021-07-20T07:18:05.503075262Z Share images, automate workflows,
and more with a free Docker ID:
2021-07-20T07:18:05.503077137Z https://hub.docker.com/
2021-07-20T07:18:05.503078887Z
2021-07-20T07:18:05.503080595Z For more examples and ideas, visit:
2021-07-20T07:18:05.503082428Z https://docs.docker.com/get-started/
```

If you want to see the output generated by a container in real-time, use the "-f" option. If you're familiar with the common tail -f command in Linux, this will make you feel right at home. The -f option stands for "follow", and it displays any new output as it's generated.

```
docker logs -f <CONTAINER_ID>
```

To stop following the output, type Ctrl-C. Note: if you run this command on an already stopped container, you will not need to type Ctrl-C, as there will be no more output generated. This is why "docker logs -f" exits. If you run the command on a running container, "docker logs -f" will pause after the last time of output from the container, and will continue to wait for more output until Ctrl-C is typed.

You can combine these options, of course.

```
docker logs -tf <CONTAINER_ID>
```

With this command, you are following the generated output with timestamps.

Summary

In this chapter, you learned:

- The most important options to the "docker run" command, including how to start the container in the background with the "-d" option.
- How to give your containers a name with the "--name" option.
- How to view not only running containers, but containers that have already executed and have been stopped.
- How to use the always restart policy on a container, to make sure it starts when the host node gets rebooted or when the container exits for any reason.
- How to view the output generated by a container that is running in the background with the "docker logs" command.
- How to use "docker system prune" to remove stopped containers.

EXERCISE: RUNNING CONTAINERS

Note: You can download all the exercises contained in this book at <https://www.LinuxTrainingAcademy.com/docker>.

Goal:

The goal of this exercise is to familiarize yourself with the various options for the "docker run" command.

Instructions:

Start a Container

Start up a container with a simple "docker run" command, using the popular key-value database, Redis:

```
docker run -dit redis
```

Check to see if it is running:

```
docker ps
```

Look closely at the left-hand side for the container hash under the CONTAINER ID column.

Also look at the arbitrary name that Docker has assigned the container, as you haven't explicitly named it. That information is on the right-hand side, under the NAMES column.

Start a Container With a Name

To stop the Redis container, you need to use its hash or arbitrary name.

Start a container with a name we can reference again later:

```
docker run -dit --name redis_container redis
```

NOTE: You can also use the "--name=CONTAINER_NAME" syntax.

Check that it's running:

```
docker ps
```

Stop a Named Container

Next, stop the container you named:

```
docker stop redis_container
```

Check the running containers on the system again:

```
docker ps
```

You should only see one Redis container, which received an arbitrary name from Docker.

Automatically Deleting a Stopped Container

Let's run a container with the "--rm" option. This option tells Docker to automatically delete the container once it stops.

We will be using an image called "hello-world", which Docker uses for demonstrations. If you don't have a local copy of the image, Docker will download it before starting the container.

```
docker run --rm hello-world
```

Check that the image has been deleted after completing its execution:

```
docker ps -a
```

Now run a container using the hello-world image without the "--rm" option, as shown here:

```
docker run hello-world
```

Check that it has not been deleted this time:

```
docker ps -a
```

Some users of Docker frequently use the "--rm" option to help with garbage collection.

Logs

The "docker logs" command provides information directly from the STDOUT (standard output) and STDERR (standard error) of a container.

The format of the command is:

```
docker logs CONTAINER_NAME
```

Create a container using the "hello-world" image:

```
docker run --name test_container -d hello-world
```

Examine the output of the "docker ps" command:

```
docker ps
```

Think about the commands you just executed and answer the following questions:

1. Why isn't the container visibly running with a "docker ps" command?
2. What command will show details of the container?
3. How would you read the output from that container, using the "docker logs" command?

The answer to the first question is because the container had already stopped by the time you executed the "docker ps" command. The hello-world image simply outputs some information and then stops. Unlike some other containers, it doesn't provide an ongoing service.

The answer to the second question is to use "docker ps -a". This command allows you to view all containers, including those that are stopped.

```
docker ps -a
```

The answer to the third question is to run "docker logs test_container" to see the output from the container.

Use the name you gave the container, "test_container" instead of using the container ID (hash).

```
docker logs test_container
```

Now display the output of the container with timestamps:

```
docker logs -t test_container
```

MAKING A CONTAINER PUBLICLY AVAILABLE

In this chapter, you will learn how to make an application that is running inside a container accessible from outside the Docker host machine. Additionally, you'll learn how to share data with containers running on your docker host system.

We will demonstrate these concepts with the popular and lightweight web server Nginx. We'll start up a container and open a specific network port so that we can view the container's web server in a web browser on our local machine.

If you want to find the "docker run" command syntax for a particular image, as well as other helpful tips about that image, then examine the image's details on Docker Hub. Ideally, only trust those marked as official images. This should offer a high degree of certainty that the image doesn't contain malware or other nefarious content.

As we touched on previously, the "docker run" command can grow lengthy and relatively complex. In the command that follows, we will use an example from the Nginx Docker Hub page. We will pick an arbitrary TCP port number on our host machine to expose our container's network port on. In this example, we will use TCP port 8080.

```
docker run --name our_nginx -d -p 8080:80 nginx
```

Output:

```
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
448f6bf000e3: Pull complete
28809c92a2a1: Pull complete
```



```
2a60e77f7ec5: Pull complete
1c1f693bbfe4: Pull complete
e155fdcac06b: Pull complete
08555aa0c452: Pull complete
Digest:
sha256:353c20f74d9b6aee359f30e8e4f69c3d7eaea2f610681c4a95849a2fd7c
Status: Downloaded newer image for nginx:latest
5e9530bf008ed98826a2afffcb1d8ea3650732f09df50966aa51f131a1f25df3
```

As you can see, we've named the container "our_nginx" and have just downloaded the stock nginx image. In a later section, we will get more involved in how networking is used with Docker. In short, "-p 8080:80" simply means open up TCP port 8080 on the Docker host machine and redirect traffic to and from it, to TCP port 80 inside the nginx container. You can think of port 8080 as being the outside world, while port 80 is the inside world of the container.

To check if the "docker run" syntax was correct, and we didn't, for example, just start the container only to have it die, we can list running containers on our system:

```
docker ps
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
5e9530bf008e	nginx	"/docker-entrypoint...."	6 minutes ago
Up 6 minutes	0.0.0.0:8080->80/tcp, :::8080->80/tcp	our_nginx	

We can see our container is running and the "STATUS" column confirms how long the container has been up. Pay close attention to the "PORTS" column. Those familiar with networking will see "0.0.0.0" mentioned before ":8080". This means that all local IP Addresses on our host machine are listening on TCP port 8080. The "->80/tcp" output means that traffic received on the host port 8080 is being directed to TCP port 80 inside the container.

Let's see what Nginx does when we talk to the container via the host. One

way to do that is with the "curl" command.

```
curl http://localhost:8080
```

Output:

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
  body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
  }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

What is returned is HTML, which is exactly what we would expect. Now congratulations is in order, as you just launched your own externally accessible Docker container!

You can use a full-blown web browser to access the Nginx web server running inside the container. If you are running Docker on your local

machine, then put "http://localhost:8080" into your web browser's address bar, just like you did with curl. However, if the Docker host machine is remote, and you want to connect to it from another system over the network, you will need the IP address of your Docker host machine.

For example, if you have Docker running on a Linux system on your private network, and you are accessing the Docker host via SSH from your laptop, you'll need to determine the IP address of the Docker host system. You can do that by running the "ip" command on the Linux Docker host system:

```
ip a
```

Output:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: ens3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9000 qdisc
pfifo_fast state UP group default qlen 1000
    link/ether 02:00:17:06:63:bf brd ff:ff:ff:ff:ff:ff
    altname enp0s3
    inet 10.4.8.15/24 brd 10.4.8.255 scope global ens3
        valid_lft forever preferred_lft forever
    inet6 fe80::17ff:fe06:63bf/64 scope link
        valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
qdisc noqueue state UP group default
    link/ether 02:42:71:c8:09:4a brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:71ff:fec8:94a/64 scope link
        valid_lft forever preferred_lft forever
```

In this example, the IP address of the Docker host system is 10.4.8.15. For

you, it will almost certainly be something entirely different, so don't expect to use this exact IP address. You have to find the IP address for your Docker host system, if you want to access it from outside the host.

Once you have determined the IP address of your Docker host system, type it along with the port into your web browser's address bar. In this example, I would visit "http://10.4.8.15:8080" and be presented with the Nginx welcome message.

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

If you've used web servers before, then you will no doubt be familiar with access logs, which are generated when a visitor requests a page from your web server. In Docker terms, a simple way of showing these access requests is by using the "docker logs" command that you've already been introduced to.

```
docker logs our_nginx
```

Output:

```
172.17.0.1 - - [20/Jul/2021:08:14:42 +0000] "GET / HTTP/1.1" 200 612 "-"
"curl/7.64.1" "-"
172.17.0.1 - - [20/Jul/2021:08:16:30 +0000] "GET / HTTP/1.1" 200 612 "-"
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:89.0)
Gecko/20100101 Firefox/89.0" "-"
2021/07/20 08:16:30 [error] 33#33: *2 open()
"/usr/share/nginx/html/favicon.ico" failed (2: No such file or directory),
client: 172.17.0.1, server: localhost, request: "GET /favicon.ico HTTP/1.1",
host: "localhost:8080", referrer: "http://localhost:8080/"
172.17.0.1 - - [20/Jul/2021:08:16:30 +0000] "GET /favicon.ico HTTP/1.1"
404 153 "http://localhost:8080/" "Mozilla/5.0 (Macintosh; Intel Mac OS X
```

```
10.15; rv:89.0) Gecko/20100101 Firefox/89.0" "-"
```

On the left-hand side, we can see the IP Address that requested a page, as well as other useful visitor information, thanks to the "docker logs" command. This is what Nginx is outputting, and the "docker logs" command is allowing us to see the output generated by the container.

Let's stop this container.

```
docker stop our_nginx
```

Output:

```
our_nginx
```

Having the default Nginx welcome page isn't necessarily much help. What we typically do when we have a web server is serve web content. So, let's make some HTML – or web content – for a container on our system to serve.

First, create a directory that will contain the HTML content.

```
mkdir webpages
```

Next, change into the newly created directory.

```
cd webpages
```

Now create a web page named "index.html" and add some content to it with this command:

```
echo 'Hi from inside the container!' > index.html
```

Now return to the original working directory. (Go back up a directory level.)

```
cd ..
```

Next, you'll run a slightly more advanced command than you've used before.

```
docker run -p 8080:80 --name another_nginx -v  
${PWD}/webpages:/usr/share/nginx/html:ro -d nginx
```

Output:

```
a74e8833a0c8518fec7e221fde5b207843aa760c4a95206bda695a76721b9661
```

Note the "PWD" in the command to use our subdirectory "webpages", which gives us the path from our current directory as a Bash variable.

```
echo ${PWD}
```

The "-v" option will be explored in further detail in a later section, but in brief, it creates a volume for sharing files. The syntax is "-v", followed by the path on the host machine you want the container to have access to, followed by a colon and where you want that data to be accessed in the container. If you want to supply options, then you'll need another colon and the options. In this case, we used "ro", which mounts the volume in read-only mode inside the container. This means that the files can be changed from the Docker host machine, but the container cannot alter the files. The container can only read data contained in that volume.

When you connect to Nginx, you will see the custom index.html page you created earlier.

```
curl http://localhost:8080
```

Output:

```
Hi from inside the container!
```

Summary

In this chapter, you learned:

- How to expose the port of an application running inside a container to the outside world.
- How to provide your container access to data stored on your Docker

host machine.

EXERCISE: MAKING A CONTAINER PUBLICLY AVAILABLE

Note: You can download all the exercises contained in this book at <https://www.LinuxTrainingAcademy.com/docker>.

Goal:

The goal of this exercise is to expose the port of a container to the outside world. You will be using the Apache HTTP Server.

Instructions:

Start a Container Using the Apache HTTP Server Image

The image name for the Apache HTTP Server is "httpd." Start an image named "apache_welcome", using the "httpd" image. Use port 9900 on the Docker host system to communicate with port 80 on the container.

```
docker run --name apache_welcome -d -p 9900:80 httpd:latest
```

Confirm the Port is Open

Use the "docker ps" command and examine the "PORTS" column for your apache_welcome container.

```
docker ps
```

Confirm that you can see "0.0.0.0:9900->80/tcp" in the "PORTS" column.

View the Application

Use the "curl" command to access Apache:

```
curl http://localhost:9900
```

You should see HTML returned.

If you are running Docker on your local system, you can check that Apache is accessible by typing "http://localhost:9900" into your web browser's address bar and hitting "Enter."

CONNECTING TO RUNNING CONTAINERS AND MANAGING CONTAINER OUTPUT

In this chapter, you'll learn how to connect to a running container using an interactive shell, as well as how to execute other commands inside a container.

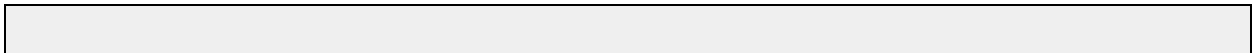
We have already discussed that it's uncommon to SSH into a container. We've also mentioned that instead of altering a running container, you would almost always adjust your Dockerfile and rebuild an image instead. However, it is sometimes helpful to be able to check exactly what a container is doing from within and to do so interactively, as opposed to just looking at the logs or the output generated by the container.

To access a container that is already running, Docker provides the "docker exec" command. But first, let's explore how to enter a container via a shell at launch time.

Entering a Container with the "run" Command

For this demonstration, I'm going to use the Apache HTTP Server image, named httpd. The image used here isn't important, but at least it gives us an application with a few processes running, as well as a filesystem that we can explore and interact with, if we choose to do so.

Use the following command to start an "Apache" container that can be interacted with directly using the Bash shell. After executing the command, you will be presented with a prompt inside the container.



```
docker run -it --name apache httpd /bin/bash
```

Output:

```
root@91b2a741c328:/usr/local/apache2#
```

Note that you will see a different prompt, as it includes the container host name, which is in turn based on the container ID.

Also, note that the "i" and "t" options were used. These options are required if you want to use an interactive shell. If you omit "-it" from the command, the container will start and then immediately stop without allowing you to interact with the shell.

Another thing to mention is that the "-d" option was not used, as that detaches the container and runs it in the background. Instead, we want to run this container in the foreground, so we can interact with it immediately.

As you already know, "--name apache" sets the name of the container to "apache". The image used to run this container is "httpd" and the command to run inside the container is "/bin/bash".

Now that you have a shell in the container, you can run commands inside the container that would normally work in a Bash shell. For example, you can display your current working directory by running "pwd".

```
pwd
```

Output:

```
/usr/local/apache2
```

To list the contents of the present working directory, run the "ls" command:

```
ls
```

Output:

```
bin build cgi-bin conf error htdocs icons include logs modules
```

To exit the container, type "CTRL-D" or "exit".

```
exit
```

At this point, you are dropped back to a prompt on your local Docker host machine.

List the running containers.

```
docker ps
```

Output:

CONTAINER	ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
-----------	----	-------	---------	---------	--------	-------	-------

Notice that you will not find this container running, as we did not supply the "-d" option to detach it and run it in the background. The container ran the command we provided, which was "/bin/bash", and when we exited Bash, the container stopped.

Try running through the commands again, this time adding the "-d" option. You can use the familiar "-dit" options to the "docker run" command. Give this container a unique name, so you don't have to delete the previous "apache" container.

```
docker run -dit --name second_apache httpd /bin/bash
```

Output:

```
c21a8a371dbd828cb3f5954d8c413d0f923884d69223bb9e7cdafcdd7bd87666
```

List the running containers.

```
docker ps
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			

Because you used the "-d" option, Docker detached the container as soon as it started. Although the container is still running, there's little point in adding "/bin/bash" to the end of the command line. This is because we're not attached to the container anyway, and so we can't type commands into the Bash shell.

Here, I want to pause and set your expectations when it comes to containers and shells. Many containers use a minimal Linux distribution called "Alpine Linux", which has an absolutely tiny footprint, making containers very small. Alpine does not use a fully-fledged version of Bash, so you need to use "/bin/sh" instead of "/bin/bash" to access a prompt. Several other images and distros do this as well. Some of them use a symlink from Bash to an old-style Bourne shell (sh), so be prepared that you may get quite frustrated without your usual Bash completions or Bash history commands being available to you inside a container.

Here is a handy trick to remember: Note that we were passing the full path of "/bin/bash" to the "docker run" command, which is a good practice, but you can also just pass "bash" or "sh" as the last parameter. This will work as long as the internal "PATH" environment variable is configured for the container and the specified command of "bash" or "sh" is in the path; most of the time, these commands are.

As previously mentioned, it doesn't help much to add "/bin/bash" to a container that will be detached immediately. This means we need another way of accessing the shell of a running container. Docker provides the "docker exec" command to achieve this.

Start another "httpd" container.

```
docker run -dit httpd
```

Output:

```
18a49466cdb16db98da79d2a0c8720b980a7d7d73ef0a980e9faa7edd7174d59
```

Now return to the "docker exec" command. You may have noticed we didn't name the container that is running the "httpd" image. This means you will have to use the hash, or container ID, to access it, or you can use the human-readable name which Docker randomly generated for the container.

Use the following command to display the list of running containers, along with their abbreviated hashes:

```
docker ps
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
18a49466cdb1	httpd	"httpd..."	2 minutes ago	Up 2 minutes
80/tcp	pedantic_cohen			
c21a8a371dbd	httpd	"/bin/bash"	4 minutes ago	Up 4 minutes
80/tcp	second_apache			
91b2a741c328	httpd	"/bin/bash"	6 minutes ago	Exited (127) 5 minutes ago
ago	apache			

Use the "docker exec" command to start a shell on that already running container. The format is "docker exec -it FIRST_THREE_CHARS_OF_CID /bin/bash".

```
docker exec -it 18a /bin/bash
```

Output:

```
root@18a49466cdb1:/usr/local/apache2#
```

To reiterate, because we want to access the container interactively, using a shell, we used the "-i" for "interactive" and "-t" for "TTY" options to "docker exec" to get a working terminal.

Exit the container.

```
exit
```

Now access that same container with the name it was given by Docker. In this example, that name is "pedantic_cohen". Of course, if you are following along on your own Docker host, Docker will most likely have assigned a different name to your container.

```
docker exec -it pedantic_cohen /bin/bash
```

Output:

```
root@18a49466cdb1:/usr/local/apache2#
```

Exit the container.

```
exit
```

Let's see if an old-style shell is also available in the container, by using "sh" at the end of the "docker exec" command.

```
docker exec -it 18a sh
```

Output:

```
#
```

Note that I didn't provide the explicit – or full – path to "/bin/sh". I simply used "sh". Again, not providing the full path to the shell works most of the time, but if it doesn't, remember to use the full path.

Stop your shell and detach from this running container.

```
exit
```

By the way, it's a common mistake to forget the "-it" options. If you do, the shell will execute and immediately exit. Let me demonstrate:

```
[root@docker_host ~]# docker exec 18a bash
[root@docker_host ~]# hostname
docker_host
```

Let's do it the right way:

```
[root@docker_host ~]# docker exec -it 18a bash
root@18a49466cdb1:/usr/local/apache2# hostname
18a49466cdb1
root@18a49466cdb1:/usr/local/apache2# exit
exit
[root@docker_host ~]#
```

Executing Commands

As you might expect, a subcommand called "exec" isn't used exclusively to open up a shell for us. It's capable of much more. Docker has designed the subcommand so that you can execute any available commands inside a container. Try it now by running a detached container:

```
docker run -dit --name execution httpd
```

Output:

```
d2966b471e601c4bc1cf7aac1b77770153f6f51cae4f13fee1f7e88d45e24b50
```

Check that the container is running correctly:

```
docker ps
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED
d2966b471e60	httpd	"httpd-foreground"	18 seconds ago
Up 17 seconds	80/tcp	execution	
18a49466cdb1	httpd	"httpd-foreground"	13 minutes ago
Up 13 minutes	80/tcp	pedantic_cohen	
c21a8a371dbd	httpd	"/bin/bash"	15 minutes ago
Up 15 minutes	80/tcp	second_apache	

Here's how to run a command to create an empty text file within the container

using "docker exec". Note that we're referring to our container using its name and detaching from it afterwards too.

```
docker exec -d execution touch /root/hello
```

You can check if that worked by entering the container using Bash:

```
docker exec -it execution /bin/bash
```

Output:

```
root@d2966b471e60:/usr/local/apache2#
```

You can see the "root" user prompt. Run the "ls" command like this:

```
ls /root
```

Output:

```
hello
```

Lo and behold, there's the "hello" file you created.

Exit the container:

```
exit
```

Of course, you could have done this as well:

```
docker exec -it execution ls /root
```

Output:

```
hello
```

By the way, the whole idea of containers is that they are very slimmed down images. They are not complete operating systems or full installations of Linux distributions. That means that some – or even many – of the

commands you might use in your day-to-day operations with Linux might not be available inside the container.

You can try several common commands now.

```
[root@docker_host ~]# docker exec -it execution bash
root@d2966b471e60:/usr/local/apache2# uptime
bash: uptime: command not found
root@d2966b471e60:/usr/local/apache2# man
bash: man: command not found
root@d2966b471e60:/usr/local/apache2# locate
bash: locate: command not found
root@d2966b471e60:/usr/local/apache2# vim
bash: vim: command not found
root@d2966b471e60:/usr/local/apache2# vi
bash: vi: command not found
root@d2966b471e60:/usr/local/apache2# nano
bash: nano: command not found
root@d2966b471e60:/usr/local/apache2# curl
bash: curl: command not found
```

If you want a command to be available inside your container, you can install it. This container image is based on Debian, so you can use "apt" commands to install a package.

```
root@d2966b471e60:/usr/local/apache2# apt update
root@d2966b471e60:/usr/local/apache2# apt install -y curl
root@d2966b471e60:/usr/local/apache2# which curl
/usr/bin/curl
```

Installing a package only alters that specific running container; it doesn't change the image that the container is based on. If you want new containers to have that piece of software installed, you'll need to create a new image with that software installed. You'll find the exact steps on how to do that, elsewhere in this book.

You might also find it frustrating that the "ps" command is not available in some of your containers.

```
root@d2966b471e60:/usr/local/apache2# ps
bash: ps: command not found
root@d2966b471e60:/usr/local/apache2# exit
```

The good news is that you can view the processes running inside the container with the "docker top" command.

```
docker top execution
```

Output:

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	2790	2765	0	16:09	?	00:00:00	httpd -DFOREGROUND
daemon	2829	2790	0	16:09	?	00:00:00	httpd -DFOREGROUND
daemon	2830	2790	0	16:09	?	00:00:00	httpd -DFOREGROUND
daemon	2832	2790	0	16:09	?	00:00:00	httpd -DFOREGROUND

You'll notice httpd processes running in the container, which provide the Apache web service.

Summary

In this chapter, you learned:

- The difference between the "docker run" command and the "docker exec" command.
- How to use the "docker exec" command to connect to a running container using an interactive shell.
- That the "docker exec" command can not only be used to start a shell, but also to run any command available inside a given container.
- That many containers are intentionally kept lightweight by excluding commands that are available on fully installed versions of the operating system.

EXERCISE: ENTERING AND CONNECTING TO CONTAINERS

Note: You can download all the exercises contained in this book at <https://www.LinuxTrainingAcademy.com/docker>.

Goal:

The goal of this exercise is to practice the two common ways of entering a container. The first method is entering the container at runtime. The second is entering or connecting to a container that is already running. In practice, the second method will be the one you'll use most often.

Instructions:

Enter a Container at Runtime

Launch a container based on the "redis" image and start an interactive shell. Name the container "enter_redis" and run the container in the foreground.

```
docker run -it --name enter_redis redis /bin/bash
```

Following that command, you are presented with a prompt that contains a username, a hostname, and a path. This most likely means that the Bash shell is available to you, as opposed to an older, simpler shell.

Check that this is a correct assumption with the "bash --version" command.

```
bash --version
```

The first line of output reports the version of Bash you are running.

Because the container is running in the foreground, when the process that started the container stops, the container itself stops. To stop the Bash shell, type "exit".

```
exit
```

Confirm that the container stopped.

```
docker ps -a
```

You should see that the status of the "enter_redis" container is "Exited."

Enter a Running Container

Most often, you'll want to connect to a container that is already running.

First, start a container named "exec_command_redis" based on the "redis" image. Run it in the background.

```
docker run -dit --name exec_command_redis redis
```

Check that it's running properly after starting it detached.

```
docker ps
```

Enter the container by running the Bash shell. Remember, that you'll need to use the "-it" option.

```
docker exec -it exec_command_redis /bin/bash
```

You are presented with a prompt again. End your shell by typing "Ctrl-D" or by executing the "exit" command.

```
exit
```

Check that the container is still running:

```
docker ps
```

You should still see your container running after disconnecting from it.

DOCKER LOGGING

In this short chapter, you'll learn where to check for logs on a host machine when using Docker. This will help you troubleshoot more effectively and keep an eye on any running resources. As you learn more about containers and orchestration, you are more likely to use logs to resolve issues. This is because of the large number of moving parts when only a few containers are being used at the same time. You will also need to access the standard output and standard error messages from your containers, just as you would see on other Linux systems.

Internal Container Logging

In earlier sections, you briefly used the "docker logs" command, and we'll look at that in a little more detail now.

This time, we'll be using the larger Jenkins application for our demonstration purposes, because it generates several logs.

To get Jenkins running, start a container with the name "busylogs" and make sure that you detach from it so that it will run in the background. To do that, run this command:

```
docker run -dit --name busylogs -p 8080:8080 -p 50000:50000  
jenkins/jenkins:lts
```

Output:

```
b8ee1d9404c2296d4b16bb773387efbef624561c8acaa42b0834a3dbd7f14390
```

This might be the first time you've seen a forward slash (/) in an image name. We'll dive deeper into image naming conventions in the section on registries. In short, what appears before the forward slash is the Docker user ID. Sometimes this is called the Docker user ID namespace. What appears after

the forward slash is the image name, and of course, what appears after the colon is the image tag.

Check that the container is running correctly:

```
docker ps
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
b8ee1d9404c2	jenkins/jenkins:lts	"/sbin/tini -- /usr/..."	47 seconds ago	Up
>50000/tcp, :::50000->50000/tcp busylogs				

You need a password to access Jenkins from a web browser. This image has the container generate a unique password the first time you launch the container. You can use the "docker logs" command to view the container's output, and you will find the password in that output.

```
docker logs busylogs
```

Output:

```
Jenkins initial setup is required. An admin user has been created and a
password generated.
Please use the following password to proceed to installation:

11da9895a92647ca93ad33ef65ceebf1

This may also be found at:
/var/jenkins_home/secrets/initialAdminPassword
```

Note: The password is randomly generated, so the one you see will be different from the example presented here.

If you want to follow any changes to the log, then you can simply add the "-f" option, which stands for 'follow'. It's a great time saver, and it's just like the "tail -f" command in Linux.


```
docker logs -f busylogs
```

Note: To stop tailing the container logs, type "Ctrl-C".

If any new output is generated by the container, it will be displayed to your screen.

Another handy tip is using the "-t" option. This gives you access to the timestamps for each entry written to the log. It's an ideal addition to assist with troubleshooting.

```
docker logs -t busylogs
```

Output:

```
2021-07-22T16:45:50.661597011Z Jenkins initial setup is required. An
admin user has been created and a password generated.
2021-07-22T16:45:50.661598803Z Please use the following password to
proceed to installation:
2021-07-22T16:45:50.661600595Z
2021-07-22T16:45:50.661602178Z 11da9895a92647ca93ad33ef65ceebf1
2021-07-22T16:45:50.661603928Z
2021-07-22T16:45:50.661605636Z This may also be found at:
/var/jenkins_home/secrets/initialAdminPassword
```

As you can see, each line is now preceded by a timestamp.

Of course, you can combine the options like this:

```
docker logs -tf busylogs
```

If you want to generate some logs, visit <http://localhost:8080> using a web browser on your Docker host system. You can also connect to port 8080 on the public IP address of your Docker host system. Exactly how to determine the IP address of your Docker host system was covered earlier.

It may take the Jenkins application several minutes to start, so be patient. If you receive an error, wait a few minutes and try connecting again.

You'll be prompted to enter a password, which you know by viewing the output of the "docker logs" command. Enter that password and click "Continue." Next, click "Install suggested plugins".

By the way, the goal of this chapter is not to teach the Jenkins application, but to teach you how to view Docker container logs, using Jenkins as an example. I won't be going into detail on what Jenkins does or how to use it; it's simply an example of a container that generates logs to view.

Return to your terminal and watch the output generated by the container.

Here is some sample output you might see generated from the container:

```
2021-07-22T17:00:13.402820133Z 2021-07-22 17:00:13.390+0000
[id=11] INFO hudson.PluginManager#install: Starting installation of a
batch of 19 plugins plus their dependencies
2021-07-22T17:00:13.410843008Z 2021-07-22 17:00:13.409+0000
[id=11] INFO hudson.model.UpdateSite$Plugin#deploy: Adding
dependent install of sshd for plugin cloudbees-folder
2021-07-22T17:00:13.416509966Z 2021-07-22 17:00:13.414+0000
[id=81] INFO hudson.model.UpdateCenter$DownloadJob#run: Starting the
installation of sshd on behalf of admin
```

To stop viewing the logs, type "Ctrl-C".

You can also use "--since" to only display entries after a certain time or date. Here's a simple timestamped example, using a date that's in the past.

```
docker logs busylogs -t --since 2021-07-01
```

If you are troubleshooting, you'll want to look at logs since a specific time, using the appropriate date for your needs.

Sometimes logs are recorded inside a container as well. In that case, examine the files in the /var/log directory as you would normally do on any Linux system. However, often containers will prevent files from being written to /var/log and instead redirect that output to standard output and standard error, which can be viewed with the "docker logs" command.

External systemd Logging

As well as logging the output of containers, the Docker Engine also creates logs of its own. As most daemons do, Docker sends its logs to the host machine's syslog system. On Red Hat Enterprise Linux derivatives, you can find some of those logs inside the file `/var/log/messages`. On Ubuntu or other Debian derivatives, these are usually found in `/var/log/syslog`.

If you start and stop a few containers, you will see a wealth of information that `dockerd` is generating. Sometimes you'll find "permission denied" errors, and sometimes other interesting errors about incorrect names being used, for example.

If you get stuck, the logging component of `systemd`, called `journald`, can also help.

This `journald` command displays the last 20 entries in the log from all the services on the system.

```
journalctl -n 20
```

You can display logs associated with a specific service by using the `-u` option. Here is how to display the last 20 logs for the Docker service:

```
journalctl -u docker.service -n 20
```

By the way, `journalctl` uses a pager by default. This means you may have to right-arrow over to see the full output of a given line. Personally, I prefer the content to be wrapped on my screen. To do that, tell `journalctl` to not use a pager, with the `--no-pager` option.

```
journalctl --no-pager -u docker.service -n 20
```

Summary

In this short chapter, you learned:

- How to view the logs created by a container.
- How to view the logs created by the Docker daemon.

DOCKER REGISTRIES

In this section, you're going to learn where container images are hosted, and the naming conventions used to access those images.

As mentioned in a previous chapter, Docker Hub is the default image registry that your Docker client is configured to connect to. The primary function of a registry is simply to store image files. This means that, as well as pulling images down from a registry, you can also push images back up to the registry, provided you have the correct permissions.

A repository within a registry is a collection of one or more images. A single repository can hold many Docker images, each stored as a tag. Typically, tags refer to a version of an image, but they can also refer to different variations of an image. For example, you may have a repository where you keep MySQL images, with some images based on Alpine Linux, while others are based on Debian Linux. You would use a tag for each one of those images.

Let's slowly break down exactly what you're connecting to when you pull an image from a repository stored in a registry. We'll use the Ubuntu image as an example, and the focal tag to pinpoint a specific image within the repository:

```
docker pull docker.io/ubuntu:focal
```

Output:

```
focal: Pulling from library/ubuntu
a31c7b29f4ad: Pull complete
Digest:
sha256:b3e2e47d016c08b3396b5ebe06ab0b711c34e7f37b98c9d37abe794b7
Status: Downloaded newer image for ubuntu:focal
docker.io/library/ubuntu:focal
```

This is the exact same command as simply running "docker pull ubuntu:focal". That's because the DNS name of the Docker Hub registry is docker.io. The official DNS name has changed a few times over the years, but you're currently encouraged to use docker.io. Older DNS entries still work, but this may change. At the time of publication, it is docker.io.

Let's perform another pull:

```
docker pull registry.hub.docker.com/library/ubuntu:focal
```

Output:

```
focal: Pulling from library/ubuntu
Digest:
sha256:b3e2e47d016c08b3396b5ebe06ab0b711c34e7f37b98c9d37abe794b7
Status: Downloaded newer image for registry.hub.docker.com/library/ubuntu
registry.hub.docker.com/library/ubuntu:focal
```

In this command, I've pulled another Ubuntu image with the same tag, but from a different DNS name. This time, the DNS name was registry.hub.docker.com and not docker.io.

Now let's check the local images:

```
docker images
```

Output:

REPOSITORY		TAG	IMAGE
ID	CREATED	SIZE	
ubuntu		focal	c29284518f49 9 days ago 72.8MB
registry.hub.docker.com/library/ubuntu		focal	c29284518f49 9 days ago 72.8MB

If you look closely, you'll see that the "IMAGE ID" column displays the same hash for both images. This means they are the same image, or were built with the same resulting ID. Also, the "TAG" column shows "focal" for both images, however under the "REPOSITORY" column, you can see these

images show different repository names. This means that if you wish to start a container with them, you would need to use a format such as this one:

```
docker run -dit registry.hub.docker.com/library/ubuntu:focal
```

Output:

```
23aaae69a6ef39e25dd70f93654c523911cf5be1bdd37efd5a90a153d3bfde51
```

The local image is actually named in this format: the registry address, followed by a forward slash (/), then the repository, followed by a forward slash (/), the image, a colon (:), and a tag.

Format:

Registry_address/Repository/Image:Tag

For this specific example:

registry.hub.docker.com/library/ubuntu:focal

The repository directly influences the name of the image. In the case of the docker.io example ("docker.io/ubuntu:focal"), the library portion of the name is obscured because of how Docker Hub handles official images. This means that usually no repository is shown. In this case, library is the namespace according to Docker's terminology – but don't worry, we'll get to that later on.

To confirm, delete the docker.io image which is just showing locally as ubuntu. Remember to use the correct tag because it's not the default "latest" tag, so you won't be able to delete it without specifying the tag "focal":

```
docker rmi ubuntu:focal
```

Output:

```
Untagged: ubuntu:focal
Untagged:
ubuntu@sha256:b3e2e47d016c08b3396b5ebe06ab0b711c34e7f37b98c9d37:
```

Confirm it has been deleted correctly:

```
docker images
```

Output:

REPOSITORY	TAG	IMAGE
registry.hub.docker.com/library/ubuntu	focal	c29284518f49
9 days ago	72.8MB	

As you can see, only the longer-named image is there.

Try this command:

```
docker run ubuntu:focal
```

Output:

```
root@docker_host:~# docker run ubuntu:focal
Unable to find image 'ubuntu:focal' locally
focal: Pulling from library/ubuntu
Digest:
sha256:b3e2e47d016c08b3396b5ebe06ab0b711c34e7f37b98c9d37abe794b7
Status: Downloaded newer image for ubuntu:focal
```

Docker reports that it's unable to find the image locally. These examples have shown how important it is to get the naming right. Even with the same hash ID for two images, the registry and repository name alters the name of the image.

To help you to remember the repository part of the naming convention, here's another example, using an unofficial image. This image might not exist when you try this command. However, Docker Hub is brimming with images, so just perform a search on hub.docker.com for popular images to test that you are getting this correct.

```
docker pull mysql/mysql-server
```

Output:

```
Using default tag: latest
latest: Pulling from mysql/mysql-server
8969f19fb2cc: Pull complete
18ff34a960f0: Pull complete
1059844cbb8f: Pull complete
3bd4cb0b78d1: Pull complete
901b41fa66ef: Pull complete
b33be9f4a1f3: Pull complete
38b3da6a86f7: Pull complete
Digest:
sha256:5241f7de0483a70f5856da995fea98904cfce8f1c51734b7f3836c1663e
Status: Downloaded newer image for mysql/mysql-server:latest
docker.io/mysql/mysql-server:latest
```

This image name follows the most common syntax. It is not listed as an official image and as a result, the namespace is mysql, followed by a forward slash (/), followed by the repository called mysql-server. Because we didn't specify a tag, we're just pulling the image with the default latest tag.

Format:

Namespace/Repository:Tag

For this specific example:

mysql/mysql-server:latest

Here is another example:

```
docker pull jasonc/centos:7.6
```

Output:

```
7.6: Pulling from jasonc/centos
d8d02d457314: Pull complete
Digest:
sha256:a36b9e68613d07eec4ef553da84d0012a5ca5ae4a830cf825bb68b9294
```



```
Status: Downloaded newer image for jasonc/centos:7.6
docker.io/jasonc/centos:7.6
```

In this command, the namespace is jasonc, which is my Docker Hub username, the repository is centos, and the image tag is 7.6. So, if or when you start to store your own images on Docker Hub, you'll use that format: your docker user-id, followed by a forward slash, followed by the repository, followed by a tag. Again, if you don't specify a tag, "latest" is the default.

Private Registries

You should be aware of the security risks involved in pulling images that aren't marked as official. Generally speaking, even official images have a high number of known software vulnerabilities that the vendor will likely address at some point in the future. However, malware is sadly common in images that appear to be official, so use these images with caution and isolate their access to other resources if you can. Ideally, build your own images from scratch, as carefully as possible.

In addition to Docker Hub, there are a number of popular container image registries, such as Amazon's Elastic Container Registry (known as ECR), Quay from Red Hat, and Google's Container Registry, which uses the gcr.io domain name.

For most business applications, it's common to have your own registry. It isn't difficult to set up a private Docker registry. As mentioned previously, Docker offers the option to make repositories private, requiring you to log in to gain access to the registry. Examining the trade-offs between running your own registry for complete control or using a vendor, will usually help businesses decide whether or not to host their images. By the way, you'll learn how to set up a private Docker registry later in this book.

Credential Storage

In this book, we will stick with Docker Hub as our main registry for most examples, as it is still the most popular public registry by far. However, we will still demonstrate how you would log in if a repository is set to private.

To access a private registry, use the "docker login" command. In order to create and access your own repositories, create an account on Docker Hub

and then run through some additional simple steps. In a future exercise, you'll be asked to create a Docker Hub account. If you want, you can do this on your own at the end of this chapter, as it's fairly straightforward, or you can wait for the upcoming exercise.

The default registry is Docker Hub and for this reason, you don't have to specify a registry name with the "docker login" command. In other words, you can use this command to access images hosted on Docker Hub.

```
docker login
```

This is what it looks like when you provide invalid credentials:

```
[root@docker_host ~]# docker login
Login with your Docker ID to push and pull images from Docker Hub. If
you don't have a Docker ID, head over to https://hub.docker.com to create
one.
Username: fakename
Password:
Error response from daemon: Get https://registry-1.docker.io/v2/:
unauthorized: incorrect username or password
```

Let's look at the help for this command.

```
docker login --help
```

Output:

```
Usage: docker login [OPTIONS] [SERVER]

Log in to a Docker registry.
If no server is specified, the default is defined by the daemon.

Options:
  -p, --password string  Password
      --password-stdin    Take the password from stdin
  -u, --username string  Username
```

From the "--help" output, you will find you can use the "-u" option to declare the username on the command line. Ideally, you wouldn't want to use the "-p" option to type your password on the command line, as that could potentially leave your password in the command line history. If you are the only one with access to your system, this isn't a big deal, simply something to keep in mind. Finally, you'll see where you can provide a server. You'll need to do this if you are using any other registry than Docker Hub.

Here, I'm using my personal Docker Hub account to demonstrate what it looks like to successfully log in. Of course, you'll use your Docker Hub username when you attempt to do the same.

```
[root@docker_host ~]# docker login
Login with your Docker ID to push and pull images from Docker Hub. If
you don't have a Docker ID, head over to https://hub.docker.com to create
one.
Username: jasonc
Password:
WARNING! Your password will be stored unencrypted in
/root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-
store

Login Succeeded
```

When you successfully log into a registry, Docker warns you that the credentials that you used to connect are stored locally in a format that could be hacked. Docker recommends that you configure a credential helper to avoid this. You will be warned about this when you first log in:

```
WARNING! Your password will be stored unencrypted in
/root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-
store
```

For personal development and testing, if you are cautious with your own

computer and its access, this is nothing to be greatly concerned with.

Summary

In this chapter, you learned:

- How to connect to a registry.
- The naming conventions used with Docker image repositories.
- That if your images contain sensitive information, you should make your repository private rather than public.
- Private repositories require credentials to access the images stored in them.

BUILDING IMAGES WITH DOCKERFILES

In this chapter, you will learn how to build an image using a Dockerfile, and how to upload that image to the Docker Hub registry.

Contents of a Dockerfile

In earlier chapters, we briefly looked at some Dockerfiles, including an abbreviated Dockerfile for Nginx. We will now cover a few more common instructions and then create a file of our own. Once we have a Dockerfile, we can build our image from it.

Here's a quick recap of the instructions we've already covered:

The "FROM" instruction tells Docker where to pull a base image from. As the name implies, the base image is going to be the base of your particular image.

The "CMD" instruction is for running a shell command inside your container. However, it can also be an argument to another command, so it's not strictly true that what follows CMD is a command, but it's a great way to simplify your thinking about this option. We'll look at this in more detail later.

The "RUN" instruction executes a command to help build your image.

"EXPOSE" is used for opening network ports.

The "VOLUME" instruction allows you to specify a disk share.

"COPY" is the instruction used for copying files into your image from a local disk. Each instruction creates a new, different layer in your image.

The "LABEL" instruction allows you to add metadata to an image. You provide "LABEL" a key-value pair. For example, you could use the key

"MAINTAINER" and the value "Jason Cannon." This would let someone know that Jason Cannon is the maintainer – or author – of the Dockerfile and the resulting image. You may see some older Dockerfiles that use the "MAINTAINER" instruction; this is an older, deprecated syntax. Currently, "LABEL" is the preferred way of declaring who the author or maintainer of an image is.

The "ENV" instruction introduces an environment variable to a container.

Another popular instruction you will see in Dockerfiles is called "ENTRYPOINT". There seems to be a bit of confusion by users at all levels regarding the "ENTRYPOINT" and "CMD" instructions. In short, there are two things to remember. First, "ENTRYPOINT" should be used almost at the end of your Dockerfile to specify which executable will run when your container starts. The following "CMD" entry after the "ENTRYPOINT" entry should then pass options to the executable, if required. Let's look at a Dockerfile to demonstrate how the "ENTRYPOINT" and "CMD" instructions relate.

Here are the contents of an example Dockerfile:

```
FROM debian:latest
LABEL maintainer="Jason Cannon"
ENTRYPOINT ["/bin/ping"]
CMD ["www.docker.com"]
```

Take a look at each line of this Dockerfile. The first line "FROM debian:latest" means that we are going to be using the latest Debian image as the base of the image.

The next line is a "LABEL" instruction, and contains a key-value pair. The key is "maintainer" and the value is "Jason Cannon". This adds metadata to the image.

The "ENTRYPOINT ["/bin/ping"]" line causes the container based on this image to execute the "/bin/ping" command when it starts.

The final line in this Dockerfile is "CMD ["www.docker.com"]". This is an argument to the ENTRYPOINT. When a container based on this image is

started, the command that will execute is `"/bin/ping www.docker.com"`.

When you create a Dockerfile, name it 'Dockerfile'. Be sure to start the file name with a capital "D" and leave all the other letters in lowercase. You can create a Dockerfile with any text editor you prefer. I would recommend vim, as that's my favorite editor, but you could use nano if you wanted to, for example.

Here is how to build an image from this Dockerfile:

```
docker build -t jasonc/dockerping:latest .
```

Output:

```
Sending build context to Docker daemon 2.048kB
Step 1/4 : FROM debian:latest
latest: Pulling from library/debian
627b765e08d1: Pull complete
Digest:
sha256:cc58a29c333ee594f7624d968123429b26916face46169304f0758064
Status: Downloaded newer image for debian:latest
---> 0980b84bde89
Step 2/4 : LABEL maintainer="Jason Cannon"
---> Running in f8aa739a5876
Removing intermediate container f8aa739a5876
---> 5c9ce5b6e788
Step 3/4 : ENTRYPOINT ["/bin/ping"]
---> Running in 13ccb1fb7580
Removing intermediate container 13ccb1fb7580
---> a3e4d36ffdf4
Step 4/4 : CMD ["www.docker.com"]
---> Running in e9093a53ed2f
Removing intermediate container e9093a53ed2f
---> cfe80f28cec0
Successfully built cfe80f28cec0
Successfully tagged jasonc/dockerping:latest
```

The command to build an image is "docker build". The "-t" option to the

"docker build" command allows you to specify a tag, or image name. The format I used for this image name was:

Docker_ID/Repository:Tag

If you are following along, you'll want to use your own Docker ID rather than mine. Creating a Docker Hub account is simple. If you haven't done so already, you'll be asked to in the exercise at the end of this chapter.

After the image name, I used a space and then a dot (or period). The "docker build" command expects you to give it a path to where it can find the Dockerfile that it will use to build the image. In Linux, a period indicates this directory or the present working directory. So by specifying a period, we're telling Docker to use the Dockerfile in our current directory.

When that command was running, we saw that Docker was busy copying files and building an image, layer by layer, from scratch.

To make sure that the image is available, use this command:

```
docker images
```

Output:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
jasonc/dockerping	latest	cfe80f28cec0	3 minutes ago	114MB
debian	latest	0980b84bde89	2 days ago	114MB

Notice we have two images. First, Docker downloaded the base image, which was debian:latest, which came from the "FROM" instruction in the Dockerfile. Next, Docker ran all the commands and instructions in the Dockerfile to create the resulting image of jasonc/dockerping:latest. The Docker ID, or name space, is jasonc, while dockerping is the repository.

Now that I have an image, I'm going to upload it to Docker Hub. To do that, I need to log in with my Docker ID.

```
docker login
```


Output:

```
Login with your Docker ID to push and pull images from Docker Hub. If
you don't have a Docker ID, head over to https://hub.docker.com to create
one.
Username: jasonc
Password:
WARNING! Your password will be stored unencrypted in
/root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-
store

Login Succeeded
```

If you haven't logged in before, you will be asked for your username and password. If you run the command after you've already been logged in, Docker will authenticate you with your existing credentials. In that case, you will see output similar to this:

```
Authenticating with existing credentials...
WARNING! Your password will be stored unencrypted in
/root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-
store

Login Succeeded
```

To push the image to Docker Hub, use this command format:

```
docker push Docker_ID/Repository:Tag
```

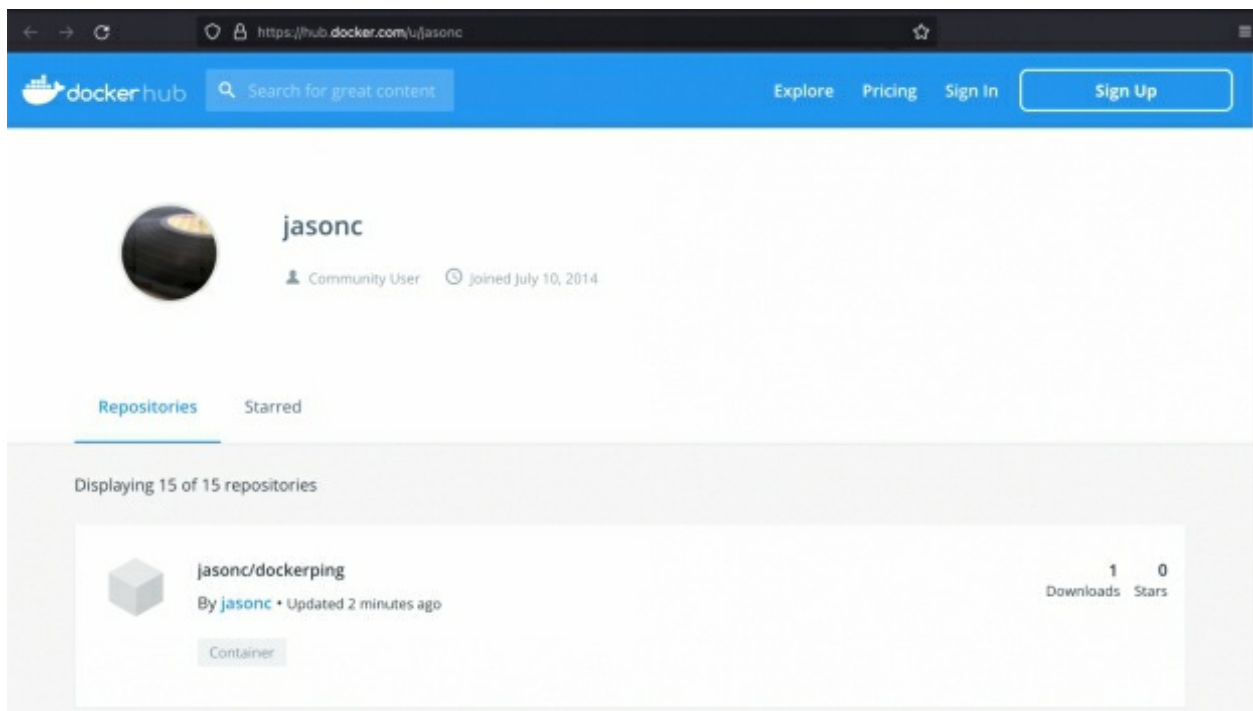
With my Docker ID and the dockering:latest image, the command is:

```
docker push jasonc/dockering:latest
```

Output:

The push refers to repository [docker.io/jasonc/dockerping]
afa3e488a0ee: Mounted from library/debian
latest: digest:
sha256:63675e08e5671e5faf6e954871b5a210aa04f954f473179bd245f2b23c
size: 529

Now look at Docker Hub and see if you can find the image listed under your user ID. Here is what my Docker Hub page looks like after I pushed the image. You can see the most recently pushed image appear at the top of the list.



The format to run the image is:

```
docker run Docker_ID/Repository:Tag
```

NOTE: Look out for the slash in the command above.

In my case, the exact command is:

```
docker run jasonc/dockerping:latest
```

Output:

```
PING d1syzps6kort6n.cloudfront.net (13.33.69.58) 56(84) bytes of data.  
64 bytes from server-13-33-69-58.phx50.r.cloudfront.net (13.33.69.58):  
icmp_seq=1 ttl=247 time=1.65 ms  
64 bytes from server-13-33-69-58.phx50.r.cloudfront.net (13.33.69.58):  
icmp_seq=2 ttl=247 time=1.67 ms  
64 bytes from server-13-33-69-58.phx50.r.cloudfront.net (13.33.69.58):  
icmp_seq=3 ttl=247 time=1.67 ms  
^C  
--- d1syzps6kort6n.cloudfront.net ping statistics ---  
3 packets transmitted, 3 received, 0% packet loss, time 4ms  
rtt min/avg/max/mdev = 1.648/1.662/1.670/0.009 ms
```

The container will ping the Docker website until you type "Ctrl-C".

```
CTRL-C
```

Let's remind ourselves of the contents of the Dockerfile used to create this image:

```
FROM debian:latest  
LABEL maintainer="Jason Cannon"  
ENTRYPOINT ["/bin/ping"]  
CMD ["www.docker.com"]
```

In this Dockerfile, the ENTRYPOINT instruction is set to "/bin/ping" and the CMD instruction is set to "www.docker.com". So, by default, when you start a container based on this image, the command that will execute is "/bin/ping www.docker.com". The default value specified by the CMD instruction can be overridden by supplying an option on the command line. For example, if you wanted to ping google.com, you would run this command:

```
docker run jasonc/dockerping google.com
```

Output:

```
PING google.com (142.250.217.142) 56(84) bytes of data.  
64 bytes from lax31s19-in-f14.1e100.net (142.250.217.142): icmp_seq=1  
ttl=117 time=8.89 ms  
64 bytes from lax31s19-in-f14.1e100.net (142.250.217.142): icmp_seq=2  
ttl=117 time=8.94 ms  
64 bytes from lax31s19-in-f14.1e100.net (142.250.217.142): icmp_seq=3  
ttl=117 time=8.97 ms  
^C  
--- google.com ping statistics ---  
3 packets transmitted, 3 received, 0% packet loss, time 5ms  
rtt min/avg/max/mdev = 8.894/8.934/8.973/0.032 ms
```

Remember to stop the execution by typing "Ctrl-C".

```
CTRL-C
```

In this example, the command that gets executed in the container is `"/bin/ping google.com"`. This is because our `ENTRYPOINT` is `"/bin/ping"`, and we overrode the default `CMD` value of `www.docker.com` and replaced it with `google.com`.

Hopefully this simple example demonstrates how the `ENTRYPOINT` and `CMD` instructions relate.

Now, let's build another image. In the first image we built, all the software we needed was already installed in the base image. Of course, the only software we used was the `"/usr/bin/ping"` command.

But what if you need a command in your image that isn't available by default in the base image? You would have to add those commands by installing a package or set of packages from the Linux distribution.

Here is another Dockerfile that will do this, which I've placed into a `nettools` subdirectory on my system.

```
cat nettools/Dockerfile
```

Output:

```
FROM debian:latest
LABEL maintainer="Jason Cannon"
RUN apt update && \
    apt install -y traceroute && \
    apt install -y curl && \
    apt clean
ENTRYPOINT ["/bin/bash"]
```

The first two lines are the same as the previous Dockerfile example. The same base image and maintainer are being used. However, what's new is the RUN instruction.

```
RUN apt update && \
```

The two ampersands (&&) are a way to concatenate commands. If you separate commands with double ampersands (&&), what happens is that the command that follows the ampersands will only get executed if the first command succeeds. By succeeding, that means the first command has an exit status of 0. So, if the "apt update" command fails, then none of the other commands get executed. However, if the "apt update" command succeeds, then it proceeds to the next command, which is "apt install -y traceroute".

Another thing to note here are the backslashes (\) that appear at the end of the line. Those are line continuation characters, which indicate, "this command line is not over, the rest of the command continues on the next line."

You can put these commands all on one line. Here is what that would look like:

```
RUN apt update && apt install -y traceroute && apt install -y curl && apt clean
```

However, if you want to make your Dockerfiles easier to read, then use line continuation characters.

By the way, all these commands are contained in one RUN instruction, so that only one layer is produced. In theory, you could have multiple RUN instructions, each performing a single "apt" command, but that would create a separate image layer for each "apt" command. That would look like this:

```
FROM debian:latest
LABEL maintainer="Jason Cannon"
RUN apt update
RUN apt install -y traceroute
RUN apt install -y curl
RUN apt clean
ENTRYPOINT ["/bin/bash"]
```

The standard convention is to use the line continuation character syntax.

```
FROM debian:latest
LABEL maintainer="Jason Cannon"
RUN apt update && \
    apt install -y traceroute && \
    apt install -y curl && \
    apt clean
ENTRYPOINT ["/bin/bash"]
```

Another thing to note is that "apt clean" is being run. That cleans up (removes) the packages that have been downloaded and successfully installed. There is no need for those packages to be lying around in the image, using disk space.

The idea of this Dockerfile and its resulting image is that we're installing a couple of network troubleshooting tools, those being traceroute and curl. That image is designed to be used interactively. This means a shell is needed to interact with the traceroute and curl commands. So, by default, we'll have a container based on this image execute "/bin/bash". As you can see, the Bash shell is used as the ENTRYPOINT. Once someone executes the container, they have access to all the commands within the container, including the traceroute and curl commands.

Let's build an image from the above Dockerfile.

```
docker build -t jasonc/nettools nettools/
```

Output, truncated:

```
Sending build context to Docker daemon 2.048kB
Step 1/4 : FROM debian:latest
---> 0980b84bde89
Step 2/4 : LABEL maintainer="Jason Cannon"
---> Using cache
---> 5c9ce5b6e788
Step 3/4 : RUN apt update && apt install -y traceroute && apt install
-y curl && apt clean
---> Running in c25127576d9f
...
Removing intermediate container c25127576d9f
---> ebff39ba3634
Step 4/4 : ENTRYPOINT ["/bin/bash"]
---> Running in 72d4a1c0291a
Removing intermediate container 72d4a1c0291a
---> 32ef1b7455cd
Successfully built 32ef1b7455cd
Successfully tagged jasonc/nettools:latest
```

The "-t" option stands for tag. Here, I'm using "jasonc/nettools". This will result in an image tagged as "jasonc/nettools:latest", because the "latest" tag is used unless you explicitly specify another in its place.

Because the Dockerfile is contained in the subdirectory of "nettools/", I've supplied that path to the "docker build" command. As you can see, the Dockerfile doesn't always have to be in your current directory.

Now that the image is built, run it.

```
root@docker_host:~# docker run -it jasonc/nettools
root@ffb01e3514d2:/# curl google.com
<HTML><HEAD><meta http-equiv="content-type"
content="text/html; charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="http://www.google.com/">here</A>.
</BODY></HTML>
```

```
root@ffb01e3514d2:/# traceroute google.com
traceroute to google.com (142.250.188.238), 30 hops max, 60 byte packets
 1 172.17.0.1 (172.17.0.1) 0.021 ms 0.107 ms 0.010 ms
 2 140.91.194.35 (140.91.194.35) 0.134 ms 140.91.194.122
   (140.91.194.122) 0.106 ms 140.91.195.35 (140.91.195.35) 0.089 ms
 3 oracle-svc075653-ic364445.ip.twelve99-cust.net
   (80.239.134.178) 1.659 ms 1.633 ms 1.583 ms
 4 phx-b6-link.ip.twelve99.net (80.239.134.177) 1.554 ms 1.531 ms 1.509
   ms
 5 * * *
 6 las-b22-link.ip.twelve99.net (62.115.125.72) 9.816 ms 9.573
   ms 10.141 ms
 7 google-ic344098-las-b24.ip.twelve99-cust.net (62.115.174.31) 9.257
   ms 8.976 ms 9.226 ms
 8 * * *
 9 142.250.226.114 (142.250.226.114) 9.414 ms 209.85.250.40
   (209.85.250.40) 9.089 ms 142.250.226.44 (142.250.226.44) 9.313 ms
10 142.251.60.129 (142.251.60.129) 9.286 ms 108.170.247.243
   (108.170.247.243) 9.945 ms 108.170.247.211 (108.170.247.211) 9.694
   ms
11 * lax31s15-in-f14.1e100.net (142.250.188.238) 9.063 ms
   108.170.230.123 (108.170.230.123) 11.210 ms
root@ffb01e3514d2:/# exit
exit
root@docker_host:~#
```

Because the image appears to be working as expected, I'm going to push it to Docker hub:

```
docker push jasonc/nettools
```

Output:

```
Using default tag: latest
The push refers to repository [docker.io/jasonc/nettools]
042c5c0f60e2: Pushed
afa3e488a0ee: Mounted from jasonc/dockerping
```



```
latest: digest:
sha256:bffe448ac6c265a019f6d886b01b296a5f1d1cf101d5da08646e16eb89
size: 741
```

Let's look at the Dockerfile again:

```
FROM debian:latest
LABEL maintainer="Jason Cannon"
RUN apt update && \
    apt install -y traceroute && \
    apt install -y curl && \
    apt clean
ENTRYPOINT ["/bin/bash"]
```

Notice that a CMD instruction was not used in the Dockerfile. That's because we didn't want to pass any arguments to Bash. In other words, there are no default arguments.

The downside of using an ENTRYPOINT, is that it is slightly more difficult to override. It's possible, but for this particular image, every time you execute it, /bin/bash is going to be executed, so you will have to use it interactively.

Let me demonstrate how to override an ENTRYPOINT. In this example, we'll change the ENTRYPOINT to "/usr/bin/curl".

```
docker run --entrypoint /usr/bin/curl -it jasonc/nettools google.com
```

Output:

```
<HTML><HEAD><meta http-equiv="content-type"
content="text/html; charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="http://www.google.com/">here</A>.
</BODY></HTML>
```

Note that anything that comes after the image name is an argument to the

ENTRYPOINT, so the command that gets executed in the container is: `"/usr/bin/curl google.com"`.

This is only one way to work with an image that has an ENTRYPOINT such as this one. However, there is a better way of doing the same thing: use the CMD instruction without the ENTRYPOINT.

Here are the contents of the "nettools/Dockerfile-allow-override" file:

```
FROM debian:latest
LABEL maintainer="Jason Cannon"
RUN apt update && \
    apt install -y traceroute && \
    apt install -y curl && \
    apt clean
CMD ["/bin/bash"]
```

Now, I'm going to build this image with the tag "allow-override". Since we are using a Dockerfile that has a different name than "Dockerfile", we need to provide Docker with the name of the file with the "-f" option.

```
cd nettools
docker build -t jasonc/nettools:allow-override -f Dockerfile-allow-override .
```

Output:

```
Sending build context to Docker daemon 3.072kB
Step 1/4 : FROM debian:latest
---> 0980b84bde89
Step 2/4 : LABEL maintainer="Jason Cannon"
---> Using cache
---> 5c9ce5b6e788
Step 3/4 : RUN apt update && apt install -y traceroute && apt install
-y curl && apt clean
---> Using cache
---> ebff39ba3634
Step 4/4 : CMD ["/bin/bash"]
---> Running in e5ff603896eb
Removing intermediate container e5ff603896eb
```

```
---> 8aef79c257d0
Successfully built 8aef79c257d0
Successfully tagged jasonc/nettools:allow-override
```

When you run a container based on this image, you get a Bash prompt, just like we did with the other image.

```
root@ubuntu:~# docker run -it jasonc/nettools:allow-override
root@71da89dd2323:/# exit
exit
```

The difference between this and the previous image is that it is easier to override. Let's say we want to override `"/bin/bash"` with the `"/usr/bin/curl google.com"` command.

```
docker run -it jasonc/nettools:allow-override curl google.com
```

Output:

```
<HTML><HEAD><meta http-equiv="content-type"
content="text/html; charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="http://www.google.com/">here</A>.
</BODY></HTML>
```

Now, push the image to Docker Hub

```
docker push jasonc/nettools:allow-override
```

Output:

```
The push refers to repository [docker.io/jasonc/nettools]
042c5c0f60e2: Layer already exists
afa3e488a0ee: Layer already exists
allow-override: digest:
```

```
sha256:813ebf34ebf0712ff32f641d7d50b58dc5780ddc6617e32c956cd0c098  
size: 741
```

Summary

In this chapter, you learned:

- The most common instructions found in Dockerfiles.
- How to build an image and push it to Docker hub. By doing so, you got a refresher on image naming conventions.

EXERCISE: BUILD AND PUSH AN IMAGE

Note: You can download all the exercises contained in this book at <https://www.LinuxTrainingAcademy.com/docker>.

Goal:

The goal of this exercise is to create an image based off a Dockerfile you make. You will also host this image on Docker Hub.

Instructions:

Create a Docker Hub Account

Create an account on Docker Hub. This account will allow you to host your own images.

Visit <https://hub.docker.com/signup> in your favorite web browser.

Follow the instructions on the screen by choosing a username (Docker ID) and a password. Be sure to accept any items such as the privacy policy and terms of service.

Create a Dockerfile

Using Debian as the base image, install the nano editor package. Also, have Bash start by default when the container based off your image is executed.

First, create a temporary directory to work in.

```
mkdir tempbuild
```

Change into the directory you created.

```
cd tempbuild
```

Create a Dockerfile by opening it with a text editor such as nano. (**NOTE:** Use your favorite text editor for the operating system you are working on. For example, nano is not installed by default on Windows and some Linux distributions. Instead, you can use notepad for Windows and TextEdit for Mac.)

```
nano Dockerfile
```

Enter the following contents into the file. **NOTE:** It is very important that the contents are exact. Capitalization, punctuation, and spacing matter!

```
FROM debian:stable  
LABEL maintainer="ENTER_YOUR_NAME_HERE"  
RUN apt update && apt install -y nano && apt clean  
CMD [ "/bin/bash" ]
```

Save the Dockerfile. If you are using nano, type Ctrl-o. Next, hit "Enter" to save the file. Finally, type Ctrl-x to exit the nano editor.

Build the Image from the Dockerfile

Build the image using your Docker Hub ID as the namespace, "nanotest" as the repository, and "latest" as the tag.

For example, if your Docker ID is "robertsmith", then use "robertsmith/nanotest:latest".

```
docker build -t YOUR_DOCKER_ID/nanotest:latest .
```

NOTE: You could also use "YOUR_DOCKER_ID/nanotest". Remember that the "latest" tag is used by default if no tag is specified.

After the build completes, check that the image is available on your local Docker host:

```
docker images
```

You should see that the nanotest image is present.

Start a Container Using the Image

Test that your image starts Bash and contains the nano package as expected.

```
docker run --name nanotest -it YOUR_DOCKER_ID/nanotest
```

You should be presented with a Bash prompt. Now, see if nano is available.

```
which nano
```

If nano is available, the above command will report the path to the nano executable: `"/bin/nano"`

Exit out of the container.

```
exit
```

Push the Image to Docker Hub

Log in to the Docker Hub registry using your Docker Hub ID.

```
docker login
```

Provide your ID and password when prompted.

Push your image to Docker Hub.

```
docker push YOUR_DOCKER_ID/nanotest:latest
```

DOCKER VOLUMES

In this chapter, you will learn how to persist and save data using Docker volumes. Additionally, you'll learn how to share the same data with multiple containers. You'll also learn how to make data available to a container in read-only mode. Finally, you'll learn how to use ephemeral volumes and how to quickly remove unused volumes.

Managing Docker Volumes

Images are built in such a way as to keep the containers based on them small, portable, and disposable. For example, images will typically contain only the packages required to provide the intended service. Also, small configuration files that rarely change are included in images. Ideally, it's best to be able to discard a container without worrying about losing important data. If possible, you don't want important data to only exist inside a container. This means if you want to persist – or save – data generated by or used by a container, you'll want to use a volume. A volume is also the ideal choice, if you want to share the same data between multiple containers.

Differing Command Options

Let's look at the different types of volumes that Docker supports.

In a previous chapter, we used the "-v" option to a "docker run" command to declare that we wanted to use a volume with a container. The abbreviated "-v" option is exactly the same as the "--volume" option. However, the newer and preferred way to mount volumes in a container is the "--mount" option. Docker recommends that you use "--mount" instead of "-v" or "--volume", as their research has shown "--mount" is easier to use. However, if you are an old Docker user like myself, then you're probably accustomed to using and seeing "-v". In any case, either works, but we will focus on "--mount", as it's the recommended way.

Creating Volumes

The "docker volume" subcommand is used to control volumes. Let's take a quick glance at its help to see what you can do with the Docker volume command.

```
docker volume --help
```

Output:

```
Usage: docker volume COMMAND
```

```
Manage volumes
```

```
Commands:
```

```
create    Create a volume
inspect   Display detailed information on one or more volumes
ls        List volumes
prune     Remove all unused local volumes
rm        Remove one or more volumes
```

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

This isn't too surprising. You can create, inspect, list, prune, and remove volumes. Now create a new volume.

```
docker volume create testdata
```

Output:

```
testdata
```

You can check that it's been correctly created by using the "docker volume ls" command.

```
docker volume ls
```

Output

DRIVER	VOLUME NAME
local	testdata

Go ahead and delete this volume, using the "docker volume rm" command.

```
docker volume rm testdata
```

Output:

```
testdata
```

Check that your volume has been deleted:

```
docker volume ls
```

Output

DRIVER	VOLUME NAME
--------	-------------

Create another volume to work with.

```
docker volume create mydata1
```

Output:

```
mydata1
```

Now list the volumes.

```
docker volume ls
```

Output:

DRIVER	VOLUME NAME
--------	-------------

```
local mydata1
```

For further information about a volume, you can also use the "inspect" command.

```
docker volume inspect mydata1
```

Output:

```
[
{
  "CreatedAt": "2021-08-06T19:30:20-06:00",
  "Driver": "local",
  "Labels": {},
  "Mountpoint": "/var/lib/docker/volumes/mydata1/_data",
  "Name": "mydata1",
  "Options": {},
  "Scope": "local"
}
```

Note: You're told where on the local host machine's filesystem the volume is actually located. In this case, the data stored in the mydata1 volume resides at /var/lib/docker/volumes/mydata1/_data on the Docker host machine.

Attaching a Volume

The next step that we'll look at is starting up a container and attaching a volume to it so that the container has access to the data in that volume.

```
docker run -d --name withvolume --mount
source=mydata1,destination=/root/volume nginx
```

Output:

```
3c3fd22d8460a8f2c63eff225875e099be0fbc0aaba4a5a2d8940d1f9efec1d9
```

Check that the container remained running after it was started:

```
docker ps
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PO
3c3fd22d8460	nginx	"/docker-entrypoint...."	17 seconds ago	Up 14 seconds	80/tcp withvolume

Sure enough, it started just like you wanted it to. Now, inspect the container and look at the "Mounts" section:

```
docker inspect withvolume | grep Mounts -A10
```

Output:

```
    "Mounts": [
      {
        "Type": "volume",
        "Source": "mydata1",
        "Target": "/root/volume"
      }
    ],
    "MaskedPaths": [
      "/proc/asound",
      "/proc/acpi",
      "/proc/kcore",
--
    "Mounts": [
      {
        "Type": "volume",
        "Name": "mydata1",
        "Source": "/var/lib/docker/volumes/mydata1/_data",
        "Destination": "/root/volume",
        "Driver": "local",
```

```
    "Mode": "z",  
    "RW": true,  
    "Propagation": ""  
}
```

As you can see, the "Mounts" section shows that `"/root/volume"` is the volume's destination inside our container, and it is set to read and write mode, as indicated by the `"RW": true` line. Here, "RW" stands for read and write. Pay attention to the long path beginning with `"/var/lib"`. It is showing the internal volume path that Docker uses on our host machine's disk. This is the source.

Abbreviations

To save you typing more than necessary, you can also use an abbreviation for the source and destination of a volume. These are `"src"` and `"dst"`, respectively. You can also use `"target"` for the destination. Sometimes, you'll see people using `source/src` and `destination/dst`, and other times you'll see people use `source/src` and `target`. Just know that `target` and `destination` are the same thing. I just want you to be aware of that, in case you see it.

Note that when you stop a container, you also need to delete the volume separately. You can do that by using `"docker volume rm"`, followed by the volume name. The good news is that when using volumes, even if a container is destroyed, the data it generated or used is still intact. Think of this as a feature of volumes, and not a bug. You want your precious data intact until you explicitly decide to delete it.

Try adding a file to the volume from the Docker host itself.

```
echo 'Hello from the mydata1 volume!' >  
/var/lib/docker/volumes/mydata1/_data/index.html
```

Make sure that the file was created.

```
cat /var/lib/docker/volumes/mydata1/_data/index.html
```

Output:

```
Hello from the mydata1 volume!
```

Now use the "exec" subcommand to enter the container that we called "withvolume", like this:

```
docker exec -it withvolume /bin/bash
```

Output:

```
root@3c3fd22d8460:/#
```

You can now see the prompt, as expected.

Check what directory you are in.

```
pwd
```

Output:

```
/
```

You are at "/", which is the root directory of the system.

When we started the container, you probably remember that we mounted a directory called "volume" off the root user's home directory. The full path to that is "/root/volume". Enter that directory and check its contents.

```
cd /root/volume  
ls
```

Output:

```
index.html
```

Here is the "index.html" that you created a moment ago.

Check to see that it is indeed the file by looking at its contents.

```
cat index.html
```

Output:

```
Hello from the mydata1 volume!
```

And of course the content is the same as what you just created because this is, in fact, the exact same file. You're just viewing the file from inside the container, instead of from the Docker host.

Detach from this container.

```
exit
```

Docker allows you to mount the same volume to multiple containers. Let's go through that process and demonstrate it with another container.

```
docker run -d --name withvolume2 --mount src=mydata1,dst=/root/volume  
nginx
```

Notice that I used the shortcuts of "src" and "dst", instead of typing out "source" and "destination".

Output:

```
2a9e326229ef28e64f8d5610e91bd0e5504a7e19b3c4dfea70038e1d171fcd19
```

Now that the container is running, you can attach to it.

```
docker exec -it withvolume2 /bin/bash
```

Output:

```
root@2a9e326229ef:/#
```

Now that you've entered the container, quickly check that your index.html file exists and contains the correct contents:

```
cat /root/volume/index.html
```

Output:

```
Hello from the mydata1 volume!
```

That certainly looks like your file.

Exit from this container.

```
exit
```

Read and Write Volumes

Let's say you don't want to allow the container to change the contents of a volume. In other words, you want the container to have read-only access to the volume. Here's how you would accomplish that:

```
docker run -d --name readcontainer --mount src=newestvolume,dst=/usr/share/nginx/html,readonly nginx
```

Output:

```
b7e69064ae4368a5a85af40d10ead8979ef8a98c441d08ffbd41b527517db7e7
```

By the way, you can use the abbreviated form of readonly, which is simply the two characters, "ro". Also, if the volume didn't exist before the container was started, Docker kindly creates it for you.

First, see if the container is up and running:

```
docker ps
```

Output:

CONTAINER						
ID	IMAGE	COMMAND	CREATED	STATUS	PO	
b7e69064ae43	nginx	"/docker-entrypoint...."	58 seconds ago	Up 57		
	seconds	80/tcp	readcontainer			
2a9e326229ef	nginx	"/docker-entrypoint...."	3 minutes ago	Up 3		


```
minutes 80/tcp withvolume2
3c3fd22d8460 nginx "/docker-entrypoint...." 14 minutes ago Up 14
minutes 80/tcp withvolume
```

It is, as there is a container named "readcontainer" in the output.

Let's see if it created your volume as well:

```
docker volume ls
```

Output:

```
DRIVER  VOLUME NAME
local   mydata1
local   newestvolume
```

Docker did create the volume, as "newestvolume" is displayed in the output.

Now let's look at the "docker inspect" output for this container and check the mounts section:

```
docker inspect readcontainer | grep Mounts -A10
```

Output:

```
    "Mounts": [
      {
        "Type": "volume",
        "Source": "newestvolume",
        "Target": "/usr/share/nginx/html",
        "ReadOnly": true
      }
    ],
    "MaskedPaths": [
      "/proc/asound",
      "/proc/acpi",
--
    "Mounts": [
```

```
{
    "Type": "volume",
    "Name": "newestvolume",
    "Source":
"/var/lib/docker/volumes/newestvolume/_data",
    "Destination": "/usr/share/nginx/html",
    "Driver": "local",
    "Mode": "z",
    "RW": false,
    "Propagation": ""
}
```

Note that the bottom section reports "RW" as false. RW, as you might expect, stands for read-write. So, if RW is false, it means that it is not "read and write". More simply stated, it means that the volume is read-only from the container. You can use this volume in read and write mode with other containers, or directly from the Docker host itself, but for this particular container, it is read-only for that volume.

Using this read-only option is a good security practice when read-write isn't required. Should a container become compromised in a security breach, the files in your volume might be left intact afterwards.

If you wish, you can also allow some containers full access to a volume and force other containers to only have read-only access to the same volume. The ability to write to the volume is set at the container level, not at the volume level.

Let's prove that the volume is indeed mounted read-only inside the container.

```
docker exec -it readcontainer bash
```

Output:

```
root@b7e69064ae43:/#
```

The "touch" command creates a file if it doesn't exist, or updates a file's timestamp if it does exist. Let's try to create an empty file using that

command.

```
touch /usr/share/nginx/html/test
```

Output:

```
touch: cannot touch '/usr/share/nginx/html/test': Read-only file system
```

Sure enough, here is a message stating that the file system is read-only.

Now exit the container.

```
exit
```

Ephemeral Volumes

From a security perspective, there's an even better way to run volumes that are destroyed after they are no longer needed, but it's not always convenient. These are known as ephemeral volumes, or short-lived volumes, and they are automatically destroyed when the container that started them is stopped.

Have a closer look at the command that uses "tmpfs" as the "type" of volume:

```
docker run -dit --name ephemeral --mount type=tmpfs,destination=/root/volume nginx
```

Output:

```
db9b8d5fe797b99fe73c2f7cba69455b86bbdd4a2b1ee59d504d5557c18ba362
```

Inspect this container and look at its mounts section.

```
docker inspect ephemeral | grep Mounts -A10
```

Output:

```
    "Mounts": [  
      {  
        "Type": "tmpfs",  
        "Target": "/root/volume"
```

```

    }
  ],
  "MaskedPaths": [
    "/proc/asound",
    "/proc/acpi",
    "/proc/kcore",
--
  "Mounts": [
    {
      "Type": "tmpfs",
      "Source": "",
      "Destination": "/root/volume",
      "Mode": "",
      "RW": true,
      "Propagation": ""
    }
  ],
  "Config": {

```

As you can see under "type", this container is connected to a "tmpfs" volume as opposed to the "type" reporting "volume" as before.

You can also add a static size to the volume after the "type=tmpfs" entry to keep an eye on disk space limits. Here's an example of how to do just that:

```
docker run -dit --name ephemeral2 --mount type=tmpfs,tmpfs-size=256M,destination=/root/volume nginx
```

Output:

```
a3171a555c0df72fc201265433440ea2288f233ffb96657d5c982bd63719f06a
```

Confirm that you are indeed restricted to just 256 MB of disk usage in that volume:

```
docker exec -it ephemeral2 df -h /root/volume
```

Output:

Filesystem	Size	Used	Avail	Use%	Mounted on
tmpfs	256M	0	256M	0%	/root/volume

Sure enough, the volume is correctly being limited in size, as the size of /root/volume is 256 MB. From inside the container – from the container's perspective – no more than 256M of data can be stored in /root/volume.

As you can see, using ephemeral volumes is a powerful way of cleaning up after containers that need extra storage space, but only temporarily. One limitation is that you cannot share "tmpfs" volumes between containers. Also, since these types of volumes disappear after the container stops, they can make it more difficult to diagnose issues, should they occur. Just so you know, at the time of publication, Docker is available on Windows, however this tmpfs feature is currently only available on Linux Docker hosts.

Other Volume Types

Even though the "--mount" option is the new, preferred way to handle volumes, in a previous section, we served content via an Nginx web server using "docker run" with a "-v" option. The "-v" option is short for "--volume". Even though it's not the preferred method of using volumes, it's still valid syntax at the time of publication, and you may see other people use it, or documentation that refers to it. So, it's better to know it than not.

With that said, here's a quick reminder of how it works. When using the "-v" option, the source and destination are separated by a colon.

```
docker run -p 8080:80 --name nginx-with-vol -v
${PWD}/webpages:/usr/share/nginx/html:ro -d nginx
```

Output:

```
2603e34b999f0eeecb021ed37253c3be2f14e8110de017e933c8b43090f8b00d
```

List the running containers.

```
docker ps
```

Output:

CONTAINER						
ID	IMAGE	COMMAND	CREATED	STATUS	PO	
2603e34b999f	nginx	"/docker-entrypoint...."	51 seconds ago	Up 49 se		
:::8080->80/tcp	nginx-with-vol					
a3171a555c0d	nginx	"/docker-entrypoint...."	9 minutes ago	Up 9		
minutes 80/tcp		ephemeral2				
db9b8d5fe797	nginx	"/docker-entrypoint...."	10 minutes ago	Up 10		
minutes 80/tcp		ephemeral				
b7e69064ae43	nginx	"/docker-entrypoint...."	15 minutes ago	Up 15		
minutes 80/tcp		readcontainer				
2a9e326229ef	nginx	"/docker-entrypoint...."	18 minutes ago	Up 18		
minutes 80/tcp		withvolume2				
3c3fd22d8460	nginx	"/docker-entrypoint...."	29 minutes ago	Up 29 mi		
80/tcp		withvolume				

Now inspect the container.

```
docker inspect nginx-with-vol | grep Mount -A10
```

Output:

```

    "MountLabel": "",
    "ProcessLabel": "",
    "AppArmorProfile": "docker-default",
    "ExecIDs": null,
    "HostConfig": {
        "Binds": [
            "/root/webpages:/usr/share/nginx/html:ro"
        ],
        "ContainerIDFile": "",
        "LogConfig": {
            "Type": "json-file",
--
    "Mounts": [
        {
            "Type": "bind",
            "Source": "/root/webpages",

```

```
        "Destination": "/usr/share/nginx/html",  
        "Mode": "ro",  
        "RW": false,  
        "Propagation": "rprivate"  
    },  
    ],  
    "Config": {
```

Using "docker run -v" and supplying a path creates a type of mount called a bind mount. You can see that it says "bind" as the "type" in the output above.

These types of volumes are less functional than using the "--mount" option and are an older way of mounting files. The bind mount target is a directory – or even just a file – on a host machine which isn't managed by Docker. That's one of the things that makes it less flexible. If paths didn't exist for the host machine's target, then you might have spent time looking into why files weren't being shared correctly. If you use Docker volumes, you can gain insight into the storage being used with the "docker volume" command set, as well as being able to manage it with those Docker commands.

We've looked at a couple of different prune commands in previous chapters. It should be no surprise that Docker gives us a prune command, along with the volumes subcommand. Let's take a look at that now.

First, stop all of your containers.

```
docker stop nginx-with-vol ephemeral2 ephemeral readcontainer  
withvolume2 withvolume
```

Output:

```
nginx-with-vol  
ephemeral2  
ephemeral  
readcontainer  
withvolume2  
withvolume
```

Next, remove those containers.

```
docker rm nginx-with-vol ephemeral2 ephemeral readcontainer withvolume2 withvolume
```

Output:

```
nginx-with-vol
ephemeral2
ephemeral
readcontainer
withvolume2
withvolume
```

Confirm that there are no running or even stopped containers on the system:

```
docker ps -a
```

Output:

CONTAINER	ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
-----------	----	-------	---------	---------	--------	-------	-------

Look at the volumes that are still available:

```
docker volume ls
```

Output:

DRIVER	VOLUME NAME
local	mydata1
local	newestvolume

Let's get some help for the prune option:

```
docker volume prune --help
```

Output:


```
Usage: docker volume prune [OPTIONS]
```

```
Remove all unused local volumes
```

```
Options:
```

```
--filter filter  Provide filter values (e.g. 'label=<label>')
```

```
-f, --force      Do not prompt for confirmation
```

It says it will "Remove all unused local volumes." Since there are no running containers, none of your volumes are being used, so let's prune those volumes.

```
docker volume prune
```

Output:

```
WARNING! This will remove all local volumes not used by at least one  
container.
```

```
Are you sure you want to continue? [y/N]
```

Docker gives you a strong warning that if you proceed, all local volumes not used by at least one container will be removed. Make sure you know what you're doing before running this command, or you will lose data. So, please be careful, especially when working in production environments. You have been warned!

In this case, we know we don't need the data on these volumes, so we can answer yes by typing "y".

```
y
```

Output:

```
Deleted Volumes:
```

```
mydata1
```

```
newestvolume
```

```
Total reclaimed space: 1.137kB
```

It shows the volumes that have been deleted and the amount of space reclaimed by the system.

To confirm all the volumes are gone, list the volumes on the Docker host.

```
docker volume ls
```

Output:

DRIVER	VOLUME NAME
--------	-------------

Summary

In this chapter, you learned:

- How to use the newer and more adaptable "--mount" option for connecting volumes to containers.
- How to share files between containers.
- How to manage the access rights to a volume and make the volume read-only by a container.
- How to use disposable or ephemeral volumes.
- How to clean up unused volumes with the "docker volume prune" command.

EXERCISE: MANAGING DOCKER VOLUMES

Note: You can download all the exercises contained in this book at <https://www.LinuxTrainingAcademy.com/docker>.

Goal:

The goal of this exercise is to create and use volumes with containers.

Instructions:

Create a Volume

Create a volume named "localvolume".

```
docker volume create localvolume
```

Use the "docker volume inspect" command to determine where Docker will store the contents of the volume on your local Docker host's filesystem.

```
docker volume inspect localvolume
```

Note the "Mountpoint" path shown, as you'll need it for the next step.

Create a text file named "file.txt" in the volume with the contents of "This file exists".

```
echo "This file exists" > /var/lib/docker/volumes/localvolume/_data/file.txt
```

NOTE: Use the output from the above inspect command to determine the path. It will probably be the same, but there is a slight chance that it may be

different on your system.

Run a Container and Attach the Volume to It

Start a detached container using the Apache image named "httpd". Mount the volume named "localvolume" inside the container at "/data".

```
docker run -d --name mountvolume --mount src=localvolume,dst=/data  
httpd
```

NOTE: The abbreviation of source ("src") and destination ("dst") was used. You could also have used this for the "--mount" option: "source=localvolume,destination=/data"

Attach to the "mountvolume" container.

```
docker exec -it mountvolume /bin/bash
```

Check that the volume is available inside the container and that the "file.txt" exists.

```
df -h  
cat /data/file.txt
```

Ensure that the contents of "/data/file.txt" is "This file exists".

Create a new file on the volume named "from-container.txt" with the contents of "Created from inside the container".

```
echo "Created from inside the container" > /data/from-container.txt
```

Confirm the file was created and that its contents are as expected:

```
cat /data/from-container.txt
```

Detach from the container.

```
exit
```

Confirm that you can see the file from the Docker host.

```
cat /var/lib/docker/volumes/localvolume/_data/from-container.txt
```

Run a Container with an Ephemeral Volume

Start a container named "tempvolume" in the background. Use the "httpd" image. Also, attach an ephemeral volume to the container at "/tempdata". You'll need to use "tmpfs" as the volume type.

```
docker run -d --name tempvolume --mount type=tmpfs,dst=/tempdata httpd
```

Check that the temporary volume is correctly created by inspecting the running container.

```
docker inspect tempvolume | grep Mounts -A10
```

Ensure that the "Type" field shows "tmpfs" and that the "Destination" field shows "/tempdata" as the path.

DOCKER NETWORKING AND DOCKERIZING APPLICATIONS

In this chapter, you will learn how Docker utilizes the existing features of the Linux kernel to control network traffic to and from Docker containers. You'll also learn about the default network used for containers started on a Docker host system. From there, you'll learn how to gather the details of all the networks in use on a Docker host. Additionally, you'll be able to determine the exact IP address used by each container. Next, you'll learn how to create user-defined networks that allow for network separation among various containers. Along the way, you'll learn how to Dockerize PHP web applications. Even though the main focus of this chapter is Docker networking, you'll see practical examples of how to deploy a web application using the best-practices regarding Docker networking. Plus, you'll learn how Docker employs an embedded DNS server with built-in service discovery for each of its user-defined networks and how to use it to your advantage when communicating between containers or deploying applications.

Linux Firewall

Docker makes extensive use of the Linux kernel's built-in firewall, Netfilter, whose user-space interface is called iptables. This may change in the future, as iptables is being replaced with nftables. Don't worry about the exact internal tool being used, just know that Docker uses whatever mechanism the Linux kernel provides for firewalling and network routing.

Let's start a container and have a quick look at what is going on behind the scenes.

```
docker run -dit -p 8080:80 php:apache
```

Output:

```
96f6f4964aaad2dfd0dba83004212b0a84cbce51588735199e95f090d67c3da9
```

The php:apache image contains Apache with PHP. Specifically, mod_php is used with httpd. If you need to Dockerize a PHP web application, consider using this image as your base.

Check that the container is running as expected and that Docker has indeed opened TCP port 8080 on your host machine and pointed it at TCP port 80 in your container.

```
docker ps
```

Output:

CONTAINER	ID	IMAGE	COMMAND	CREATED	STATUS
	96f6f4964aaa	php:apache	"docker-php-entrypoi..."	About a minute ago	
			>80/tcp, :::8080->80/tcp	infallible_chaplygin	

If you look at the "PORTS" section, you can see that Docker opened port 8080 on the host machine and pointed to port 80 inside the container.

Let's look at the DOCKER Chain with iptables.

```
iptables -nL "DOCKER"
```

Output:

```
Chain DOCKER (2 references)
target prot opt source      destination
ACCEPT tcp  --  0.0.0.0/0    172.17.0.2    tcp dpt:80
```

You can see an entry for TCP port 80 as a destination port (dpt:80). This means that Docker is applying a local, non-routable on the Internet, IP Address to our running container. To prove it, access Apache running inside the container by that IP address.

```
curl http://172.17.0.2
```

Output:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
<p>You don't have permission to access this resource.</p>
<hr>
<address>Apache/2.4.38 (Debian) Server at 172.17.0.2 Port 80</address>
</body></html>
```

Here you can see some HTML produced by the Apache HTTPD server. This means you have directly connected to TCP port 80, which is being served by our container.

To see Docker's networking model in action, connect to the host machine's IP Address on TCP port 8080 with this command:

```
curl http://127.0.0.1:8080
```

Output:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
<p>You don't have permission to access this resource.</p>
<hr>
<address>Apache/2.4.38 (Debian) Server at 172.17.0.2 Port 80</address>
</body></html>
```

There's the HTML from the web server again. In other words, Docker's internal routing is seamlessly forwarding traffic.

Let's stop this container for now.

```
docker stop 96f
```

Output:

```
96f
```

Host-Based Networking

Sometimes, you may want to switch off Docker's sophisticated networking for compatibility reasons, or to simplify existing iptables configurations. Be aware that this is not secure as there is no isolation for the host machine, in terms of a container's own networking being able to affect the host's networking stack. Also, this is a rather uncommon thing to do, though it is an option. I want to demonstrate it before we move on to some more practical networking use-cases.

You can enable this feature by using "--network host" as an option in your "docker run" command. This allows the container to share the host's networking namespace. Also, the container does not get its own IP-address, as you can see here:

```
docker run -dit --network host php:apache
```

Output:

```
dcc30a9db0620d73bfd8b4d22059db521ea055e5777938c6ff989bb768259a91
```

If you look at the iptables' Docker chain, you'll find it has no rules to route traffic.

```
iptables -nL "DOCKER"
```

Output:

```
Chain DOCKER (2 references)
```

target	prot	opt	source	destination
--------	------	-----	--------	-------------

Let's see if we can access nginx running in that container.

```
curl http://127.0.0.1
```

Output:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
<p>You don't have permission to access this resource.</p>
<hr>
<address>Apache/2.4.38 (Debian) Server at 127.0.0.1 Port 80</address>
</body></html>
```

Notice that I accessed port 80 directly on the Docker host. Again, using the "--network host" option makes the container share the Docker host's network. If port 80 opens up on the container, then using "--network host", port 80 gets opened directly up on the host. Again, I don't recommend doing this by default, but it's there if you need it.

Stop the container.

```
docker stop dcc
```

Output:

```
dcc
```

The Default Docker Network Bridge

One point I'd like to make is if you are using an orchestrator, you most likely won't have to worry about the details of how Docker handles networking. However, it's good to know so that you can do it manually when you need or

want to, and so you can also troubleshoot issues if they arise, even if you are using an orchestrator. With that said, let's get into some of these details. First, to check the networks in use by Docker, use the "docker network ls" command:

```
docker network ls
```

Output:

NETWORK ID	NAME	DRIVER	SCOPE
439b37cc0e40	bridge	bridge	local
66791c99cca3	host	host	local
da67138fbdb1	none	null	local

The first network listed is the default "bridge" network. Its name is "bridge", and its type is bridge. Again, this is the default network and that means that if you do not specify a network for a given container, it will use the network named "bridge". In broad terms, a bridge is a networking device that creates a single aggregate network from multiple communication networks or network segments. In simpler terms, a bridge connects – or bridges – everything that is attached to it. So, if you have a web server container attached to the default bridge network, and a database container also attached to the default bridge network, they can communicate with each other.

The second and third networks listed in the "docker network ls" output knit together the container and the host machine's own networking stacks. For the most part, you can ignore these networks.

Again, the most important thing to keep in mind is that the network named "bridge" is the default network for containers, unless you override this default. We'll be talking about how to do that shortly.

To investigate a particular network in more detail, use the "docker network inspect" command and pass it a network, like so:

```
docker network inspect bridge
```

Output:

```
[
  {
    "Name": "bridge",
    "Id":
"439b37cc0e406dca2bcc80bc6efde3936f7af67d20d7561f93fea30076561ea3",
    "Created": "2021-08-06T19:27:46.831104422-06:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "172.17.0.0/16",
          "Gateway": "172.17.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": 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",
      "com.docker.network.driver.mtu": "1500"
    },
    "Labels": {}
  }
]
```

```
]
```

There is a lot of very useful debugging information being displayed here. Under "Containers", for example, you can view which containers are currently using the bridge to connect back to the network stack on the host machine, as well as the container's associated IP Address.

Let me start a container and show you.

```
docker run --name w1 -dit php:apache
docker network inspect bridge
```

Output:

```
[
  {
    "Name": "bridge",
    "Id": "439b37cc0e406dca2bcc80bc6efde3936f7af67d20d7561f93fea30076561ea3",
    "Created": "2021-08-06T19:27:46.831104422-06:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "172.17.0.0/16",
          "Gateway": "172.17.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    }
  }
]
```

```
    },
    "ConfigOnly": 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",
        "com.docker.network.driver.mtu": "1500"
    },
    "Labels": {}
}
]
```

As you can see, lots of useful debugging information is being displayed. Under "Containers," for example, you can view which containers are currently using the bridge to connect back to the network stack on the host machine, as well as the container's associated IP address. In this example, no containers are using the bridge network. Let me start a container to show you what that looks like. I'm going to name this container "w1", to represent web server number 1.

```
docker run --name w1 -dit php:apache
```

Output:

```
1d90bfa3b9ab222f0af7e75c23128c0ba0fe83c6a033cfab8255d95d70ff5345
```

Now let's inspect the bridge network.

```
docker network inspect bridge
```

Output:

```
[
```

```
{
  "Name": "bridge",
  "Id":
"439b37cc0e406dca2bcc80bc6efde3936f7af67d20d7561f93fea30076561ea3",
  "Created": "2021-08-06T19:27:46.831104422-06:00",
  "Scope": "local",
  "Driver": "bridge",
  "EnableIPv6": false,
  "IPAM": {
    "Driver": "default",
    "Options": null,
    "Config": [
      {
        "Subnet": "172.17.0.0/16",
        "Gateway": "172.17.0.1"
      }
    ]
  },
  "Internal": false,
  "Attachable": false,
  "Ingress": false,
  "ConfigFrom": {
    "Network": ""
  },
  "ConfigOnly": false,
  "Containers": {
    "1d90bfa3b9ab222f0af7e75c23128c0ba0fe83c6a033cfab8255"
  }
{
  "Name": "w1",
  "EndpointID":
"2f54c8ce915fd6418e58974ceb5d4576a8c93f2850d680d0f6758bec14685a5",
  "MacAddress": "02:42:ac:11:00:02",
  "IPv4Address": "172.17.0.2/16",
  "IPv6Address": ""
}
},
  "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",
        "com.docker.network.driver.mtu": "1500"
    },
    "Labels": {}
}
]
```

Here you can see the details of the container that is using the bridge network. Its Mac address is 02:42:ac:11:00:02 and its IP address is 172.17.0.2.

Let's start another container.

```
docker run --name w2 -dit php:apache
```

Output:

```
0ea4d5e68fc99b7f15133df4603bd27b28a055472af1643b70a6bcfb16978df9
```

Inspect the bridge network again.

```
docker network inspect bridge
```

Output:

```
[
  {
    "Name": "bridge",
    "Id":
"439b37cc0e406dca2bcc80bc6efde3936f7af67d20d7561f93fea30076561ea3
    "Created": "2021-08-06T19:27:46.831104422-06:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
```



```
"IPAM": {
  "Driver": "default",
  "Options": null,
  "Config": [
    {
      "Subnet": "172.17.0.0/16",
      "Gateway": "172.17.0.1"
    }
  ],
  "Internal": false,
  "Attachable": false,
  "Ingress": false,
  "ConfigFrom": {
    "Network": ""
  },
  "ConfigOnly": false,
  "Containers": {
    "0ea4d5e68fc99b7f15133df4603bd27b28a055472af1643b70a"
  {
    "Name": "w2",
    "EndpointID":
"2d537dce3d3d5dd9b8314a1dd26d84ca889d92b7d28870215e3da41fbdac2d
    "MacAddress": "02:42:ac:11:00:03",
    "IPv4Address": "172.17.0.3/16",
    "IPv6Address": ""
  },
    "1d90bfa3b9ab222f0af7e75c23128c0ba0fe83c6a033cfab8255"
  {
    "Name": "w1",
    "EndpointID":
"2f54c8ce915fd6418e58974ceb5d4576a8c93f2850d680d0f6758bec14685a57
    "MacAddress": "02:42:ac:11:00:02",
    "IPv4Address": "172.17.0.2/16",
    "IPv6Address": ""
  }
  },
}
```

```
    "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",
        "com.docker.network.driver.mtu": "1500"
    },
    "Labels": {}
}
]
```

Now you can see that there is another container using the bridge network. Its IP address is 172.17.0.3.

Let's prove that all the containers on the bridge network can communicate with each other. You can do that by attaching to one of the containers and then connecting to the other via the network.

```
docker exec -it w1 bash
```

Output:

```
root@1d90bfa3b9ab:/var/www/html#
```

Use the "ip" command to show the IP address inside the container. This command isn't installed by default in this image, so add it to this running container now.

First, download the latest package information.

```
apt update
```

Output:

```
Get:1 http://deb.debian.org/debian buster InRelease [122 kB]
Get:2 http://deb.debian.org/debian buster-updates InRelease [51.9 kB]
Get:3 http://security.debian.org/debian-security buster/updates InRelease
```

```
[65.4 kB]
Get:4 http://deb.debian.org/debian buster/main amd64 Packages [7907 kB]
Get:5 http://security.debian.org/debian-security buster/updates/main amd64
Packages [299 kB]
Get:6 http://deb.debian.org/debian buster-updates/main amd64 Packages
[15.2 kB]
Fetched 8460 kB in 4s (1925 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
4 packages can be upgraded. Run 'apt list --upgradable' to see them.
```

Install the package that provides the "ip" command.

```
apt install -y iproute2
```

Output:

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer
required:
  sensible-utils
Use 'apt autoremove' to remove it.
The following additional packages will be installed:
  libatm1 libcap2 libcap2-bin libelf1 libmnl0 libpam-cap libxtables12
Suggested packages:
  iproute2-doc
The following NEW packages will be installed:
  iproute2 libatm1 libcap2 libcap2-bin libelf1 libmnl0 libpam-cap
libxtables12
0 upgraded, 8 newly installed, 0 to remove and 4 not upgraded.
Need to get 1211 kB of archives.
After this operation, 4126 kB of additional disk space will be used.
Get:1 http://deb.debian.org/debian buster/main amd64 libcap2 amd64
1:2.25-2 [17.6 kB]
```

Get:2 http://deb.debian.org/debian buster/main amd64 libelf1 amd64 0.176-1.1 [161 kB]
Get:3 http://deb.debian.org/debian buster/main amd64 libmnl0 amd64 1.0.4-2 [12.2 kB]
Get:4 http://deb.debian.org/debian buster/main amd64 libxtables12 amd64 1.8.2-4 [80.0 kB]
Get:5 http://deb.debian.org/debian buster/main amd64 libcap2-bin amd64 1:2.25-2 [28.8 kB]
Get:6 http://deb.debian.org/debian buster/main amd64 iproute2 amd64 4.20.0-2+deb10u1 [826 kB]
Get:7 http://deb.debian.org/debian buster/main amd64 libatm1 amd64 1:2.5.1-2 [71.0 kB]
Get:8 http://deb.debian.org/debian buster/main amd64 libpam-cap amd64 1:2.25-2 [14.3 kB]
Fetched 1211 kB in 0s (4185 kB/s)
debconf: delaying package configuration, since apt-utils is not installed
Selecting previously unselected package libcap2:amd64.
(Reading database ... 13547 files and directories currently installed.)
Preparing to unpack .../0-libcap2_1%3a2.25-2_amd64.deb ...
Unpacking libcap2:amd64 (1:2.25-2) ...
Selecting previously unselected package libelf1:amd64.
Preparing to unpack .../1-libelf1_0.176-1.1_amd64.deb ...
Unpacking libelf1:amd64 (0.176-1.1) ...
Selecting previously unselected package libmnl0:amd64.
Preparing to unpack .../2-libmnl0_1.0.4-2_amd64.deb ...
Unpacking libmnl0:amd64 (1.0.4-2) ...
Selecting previously unselected package libxtables12:amd64.
Preparing to unpack .../3-libxtables12_1.8.2-4_amd64.deb ...
Unpacking libxtables12:amd64 (1.8.2-4) ...
Selecting previously unselected package libcap2-bin.
Preparing to unpack .../4-libcap2-bin_1%3a2.25-2_amd64.deb ...
Unpacking libcap2-bin (1:2.25-2) ...
Selecting previously unselected package iproute2.
Preparing to unpack .../5-iproute2_4.20.0-2+deb10u1_amd64.deb ...
Unpacking iproute2 (4.20.0-2+deb10u1) ...
Selecting previously unselected package libatm1:amd64.
Preparing to unpack .../6-libatm1_1%3a2.5.1-2_amd64.deb ...

```
Unpacking libatm1:amd64 (1:2.5.1-2) ...
Selecting previously unselected package libpam-cap:amd64.
Preparing to unpack .../7-libpam-cap_1%3a2.25-2_amd64.deb ...
Unpacking libpam-cap:amd64 (1:2.25-2) ...
Setting up libatm1:amd64 (1:2.5.1-2) ...
Setting up libcap2:amd64 (1:2.25-2) ...
Setting up libcap2-bin (1:2.25-2) ...
Setting up libmnl0:amd64 (1.0.4-2) ...
Setting up libxtables12:amd64 (1.8.2-4) ...
Setting up libelf1:amd64 (0.176-1.1) ...
Setting up libpam-cap:amd64 (1:2.25-2) ...
debconf: unable to initialize frontend: Dialog
debconf: (No usable dialog-like program is installed, so the dialog based
frontend cannot be used. at /usr/share/perl5/Debconf/FrontEnd/Dialog.pm
line 76.)
debconf: falling back to frontend: Readline
Setting up iproute2 (4.20.0-2+deb10u1) ...
debconf: unable to initialize frontend: Dialog
debconf: (No usable dialog-like program is installed, so the dialog based
frontend cannot be used. at /usr/share/perl5/Debconf/FrontEnd/Dialog.pm
line 76.)
debconf: falling back to frontend: Readline
Processing triggers for libc-bin (2.28-10) ...
```

Finally, you can use the "ip a" command to display the container's IP address from within.

```
ip a
```

Output:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
93: eth0@if94: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu
```

```
1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever
```

As you can see, the IP address of this container is 172.17.0.2. The other container's IP address is 172.17.0.3. So, let's see if we can curl to the IP address of the other container, which is 172.17.0.3.

```
curl http://172.17.0.3
```

Output:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
<p>You don't have permission to access this resource.</p>
<hr>
<address>Apache/2.4.38 (Debian) Server at 172.17.0.3 Port 80</address>
</body></html>
```

Sure enough, HTML is returned from the other container. Let's detach from this container.

```
exit
```

Look at the logs produced by the w2 container.

```
docker logs w2
```

Output:

```
AH00558: apache2: Could not reliably determine the server's fully
qualified domain name, using 172.17.0.3. Set the 'ServerName' directive
globally to suppress this message
```

```
AH00558: apache2: Could not reliably determine the server's fully
qualified domain name, using 172.17.0.3. Set the 'ServerName' directive
globally to suppress this message
[Sat Aug 07 21:11:29.548687 2021] [mpm_prefork:notice] [pid 1]
AH00163: Apache/2.4.38 (Debian) PHP/8.0.9 configured -- resuming
normal operations
[Sat Aug 07 21:11:29.548741 2021] [core:notice] [pid 1] AH00094:
Command line: 'apache2 -D FOREGROUND'
[Sat Aug 07 21:17:28.320623 2021] [autoindex:error] [pid 20] [client
172.17.0.2:57512] AH01276: Cannot serve directory /var/www/html/: No
matching DirectoryIndex (index.php,index.html) found, and server-
generated directory index forbidden by Options directive
172.17.0.2 - - [07/Aug/2021:21:17:28 +0000] "GET / HTTP/1.1" 403 436
 "-" "curl/7.64.0"
```

Here, the last line of output states there was a network connection from 172.17.0.2, which is the IP address of the container named "w1".

Network Isolation for a Dockerized Application

Let's say you want to provide network separation for a group of containers. For example, you may have two containers that will host a blog. One container will act as the web server, and the other container will act as the database server. You want to allow the web server to talk to the database server, but no other containers should be able to access the database server. Also, no one from outside the Docker host needs to talk to the database either. In other words, you want to make sure the database is inaccessible to the public. To do this, you would create a network for your blog and attach these two containers to it. Let's do that now.

```
docker network create blog
```

Output:

```
1f30513b9e9eea557258f3e07f0573b9e7140e871977586a29ab4285f77bd7b9
```

Now list the networks.

```
docker network ls
```

Output:

NETWORK ID	NAME	DRIVER	SCOPE
1f30513b9e9e	blog	bridge	local
439b37cc0e40	bridge	bridge	local
66791c99cca3	host	host	local
da67138fbdb1	none	null	local

You can now see a new network named "blog" that is of type "bridge."

Start a container using the php:apache image and assign it to the blog network. First, create a volume to store the web data for this container.

```
docker volume create blog_web_data
```

Output:

```
blog_web_data
```

Name this container "web" and use the "--network" option to tell Docker to put this container on the "blog" network, instead of the default "bridge" network. Also, publish port 80 on the Docker host and send that traffic to port 80 in the container. Finally, mount your volume to "/var/www/html" inside the container.

```
docker run --name web --network blog -p 80:80 --mount  
src=blog_web_data,dst=/var/www/html -dit php:apache
```

Output:

```
5622d67998948ca2567282283b4919b43120496006122d7b8a48eaa6503c2dc
```

Now inspect your blog network.

```
docker network inspect blog
```


Output:

```
[
  {
    "Name": "blog",
    "Id":
"1f30513b9e9eea557258f3e07f0573b9e7140e871977586a29ab4285f77bd7b",
    "Created": "2021-08-07T15:20:37.483805353-06:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.18.0.0/16",
          "Gateway": "172.18.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {
      "5622d67998948ca2567282283b4919b43120496006122d7b8"
    }
  },
  {
    "Name": "web",
    "EndpointID":
"6da1d0b84abc6bd66cbd164d1b8efe7e0c35db6417a2cf3057dc2371d65f7c9",
    "MacAddress": "02:42:ac:12:00:02",
    "IPv4Address": "172.18.0.2/16",
    "IPv6Address": ""
  }
]
```

```
        }  
    },  
    "Options": {},  
    "Labels": {}  
}  
]
```

This container has an IP address of 172.18.0.2. To prove that the containers on one network cannot communicate with containers on a different network, you can attach to the container named "w1" that's on the bridge network, and attempt to access the container that's running on the blog network.

```
docker exec -it w1 bash
```

Output:

```
root@1d90bfa3b9ab:/var/www/html#
```

Use the "-m 2" option that tells curl to time out after waiting for two seconds. Otherwise, you would have to wait a couple of minutes to let curl time out, if you didn't specify this option.

```
curl -m 2 http://172.18.0.2
```

Output:

```
curl: (28) Connection timed out after 2000 milliseconds
```

If curl doesn't respond immediately, then you know you can't connect over the network to the other container. You have effectively isolated your web container by putting it on a different network, so this is working as expected.

Let's exit this container.

```
exit
```

The next step in creating your blog is to create a container that will act as its database server. For this example, we're going to use the MariaDB database

server image. By the way, MariaDB is an enhanced, drop-in replacement for MySQL.

The MariaDB image uses a volume for the `"/var/lib/mysql"` directory. We know this by looking at the documentation on Docker hub. Plus, we can confirm this by looking at the Dockerfile used to create the image. In the Dockerfile, you'll find a `VOLUME` instruction with `"/var/lib/mysql"` as the argument to that `VOLUME` instruction. You can either create your own volume and mount it at `"/var/lib/mysql"`, or you can just start the container and Docker will create a volume for you and mount it there. I prefer the first option of explicitly creating a volume. This way you can name it what you want and be very clear as to what is happening.

Ok, let's create the volume for your database.

```
docker volume create blog_db_data
```

Output:

```
blog_db_data
```

List the volumes.

```
docker volume ls
```

Output:

```
DRIVER    VOLUME NAME
local     blog_db_data
local     blog_web_data
```

Here, there are two volumes: one for the web container and one for the database container.

At the time of publication, the MariaDB image page on Docker Hub is located at https://hub.docker.com/_/mariadb. You can scroll down to read the "Environment Variables" section. As you go over the documentation for this image, you'll find that you need to provide at least one environment variable

at run time. You can use the "-e" option to the "docker run" command to set the MYSQL_ROOT_PASSWORD environment variable. Provide the root password you want to use, and when the container starts, it will configure the root password accordingly.

Also, if you set the MYSQL_DATABASE environment variable, the container will create a database when it starts, with the name you provide. We can take advantage of this and create a database named "wordpress". I want to mention one little “gotcha” with this: it can take a couple of minutes before the database creation process completes. So, if the database isn't immediately available after starting the container, wait a couple of minutes before trying again.

Let's name this container "db", which stands for database. We're going to attach this container to the blog network that we previously created, using the "-e" option to set an environmental variable. We will set the MYSQL_ROOT_PASSWORD environmental variable to be the very insecure password of "pw123". We'll also set the MYSQL_DATABASE environment variable to "wordpress". Note: Do NOT use spaces when setting name-value pairs such as these environment variables. If you do, you will get an error. Do not use a space before or after the equals sign (=).

We'll also use the "--mount" option to attach the volume to this container. Finally, we'll use the name of the image, which is "mariadb".

```
docker run --name db --network blog -e
MYSQL_ROOT_PASSWORD=pw123 -e
MYSQL_DATABASE=wordpress --mount
src=blog_db_data,dst=/var/lib/mysql -d mariadb
```

Output:

```
27d1f5d6aa52615d0bc490e80a8b066889cacee61d359774519898e90d66bf50
```

Of course, if you don't have the image locally, Docker will pull it down for you before starting the container.

By the way, notice that I didn't use a "-p" option to publish the MySQL port of 3306. This is so the port isn't exposed to the world, as only the web server

will be able to access it.

Let's inspect our blog network.

```
docker network inspect blog
```

Output:

```
[
  {
    "Name": "blog",
    "Id": "1f30513b9e9eea557258f3e07f0573b9e7140e871977586a29ab4285f77bd7b",
    "Created": "2021-08-07T15:20:37.483805353-06:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.18.0.0/16",
          "Gateway": "172.18.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {
      "27d1f5d6aa52615d0bc490e80a8b066889cacee61d35977451": {
        "Name": "db",
```

```

        "EndpointID":
        "b8bcda14cd55d4c09cd3bed3595a94e4028871183f58e95e099b3587a879e40",
        "MacAddress": "02:42:ac:12:00:03",
        "IPv4Address": "172.18.0.3/16",
        "IPv6Address": ""
    },
    "5622d67998948ca2567282283b4919b43120496006122d7b8"
{
    "Name": "web",
    "EndpointID":
    "6da1d0b84abc6bd66cbd164d1b8efe7e0c35db6417a2cf3057dc2371d65f7c9",
    "MacAddress": "02:42:ac:12:00:02",
    "IPv4Address": "172.18.0.2/16",
    "IPv6Address": ""
}
    },
    "Options": {},
    "Labels": {}
}
]

```

The db container was given an IP address of 172.18.0.3.

Although this chapter is primarily focused on networking, let's quickly look at the Docker volume we used for this database container.

```
docker volume inspect blog_db_data
```

Output:

```

[
  {
    "CreatedAt": "2021-08-07T15:48:27-06:00",
    "Driver": "local",
    "Labels": {},
    "Mountpoint": "/var/lib/docker/volumes/blog_db_data/_data",
    "Name": "blog_db_data",
    "Options": {},

```

```
    "Scope": "local"
  }
]
```

Grab the mount point `"/var/lib/docker/volumes/blog_db_data/_data/"` and perform an `"ls"` command there.

```
ls /var/lib/docker/volumes/blog_db_data/_data/
```

Output:

```
aria_log.00000001  ddl_recovery.log  ib_logfile0  ibtmp1      mysql
aria_log_control  ib_buffer_pool  ibdata1     multi-
master.info       performance_schema  wordpress
```

If you're familiar with MariaDB or MySQL, you won't be surprised by what you see here. Ultimately, this is the data being created by and used by the database server. If this container were to be destroyed, its data would still reside in this volume. That means you can start a new container using this same mount point and your data is preserved.

Be aware that none of the environment variables will have any effect if you start a container with a data directory that already contains a database. So, any pre-existing database will always be left untouched on container startup.

Okay, enough about volumes, let's get back to networking!

We can connect to the database server from our web server.

```
docker exec -it web bash
```

Output:

```
root@5622d6799894:/var/www/html#
```

A MySQL client is required, so go ahead and install it now.

```
apt update
apt install -y default-mysql-client
```

We'll use the "-h" option to the "mysql" client to provide a host name or IP address of the host you want to connect to, and "-p" to make the client prompt you for the password "pw123". (When prompted, enter the password "pw213".)

```
mysql -h 172.18.0.3 -p
```

Output:

```
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 3
Server version: 10.6.3-MariaDB-1:10.6.3+maria~focal mariadb.org binary
distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

Great! You can now access the database server from the web container on the blog network. For example, you can list the databases that reside in the database container.

```
show databases;
```

Output:

```
+-----+
| Database      |
+-----+
| information_schema |
| mysql          |
| performance_schema |
| sys            |
| wordpress      |
```



```
+-----+  
5 rows in set (0.034 sec)
```

You can see that there is a database named "wordpress" that was created at runtime because we set the `MYSQL_DATABASE` environment variable, which is used by the container to create the database.

Exit out of the MySQL client.

```
exit;
```

Another great thing about user-defined networks, is that Docker provides an embedded DNS server with built-in service discovery. This means that we can access containers by their name. So, instead of manually determining the IP address of the database container, you can access it by its name "db". Use the password "pw123" when prompted.

```
mysql -h db -p
```

Output:

```
Enter password:  
Welcome to the MariaDB monitor.  Commands end with ; or \g.  
Your MariaDB connection id is 6  
Server version: 10.6.3-MariaDB-1:10.6.3+maria~focal mariadb.org binary  
distribution  
  
Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.  
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.  
MariaDB [(none)]>
```

Now exit out of the MySQL client.

```
exit;
```

This is the best method because when a container restarts, Docker could assign it a different IP address. Use the container's name and let Docker

handle the translation from that name to the proper IP address.

It's important to note that currently, this embedded DNS feature only works with user-defined networks. It does not work on the default bridge network. Also, DNS does not work across networks. So, you can't resolve a container's name if it's on a different network. The exception is if a container is attached to both networks. To attach a container to another network, use the "docker network connect" command.

First, disconnect from the web container and return to the Docker host.

```
exit
```

Here is the help for the "docker network connect" command:

```
docker network connect --help
```

Output:

```
Usage: docker network connect [OPTIONS] NETWORK CONTAINER
```

```
Connect a container to a network
```

```
Options:
```

```
--alias strings      Add network-scoped alias for the container
--driver-opt strings  driver options for the network
--ip string           IPv4 address (e.g., 172.30.100.104)
--ip6 string          IPv6 address (e.g., 2001:db8::33)
--link list           Add link to another container
--link-local-ip strings Add a link-local address for the container
```

Provide the name of the network you want to attach a container to, followed by the container name.

Ok, let's finish setting up our blog. I'm going to be using WordPress, as you may have guessed based on the database name used. WordPress is written in PHP, which is why we are using the PHP image. WordPress, via PHP, needs a way to connect to the database. This means we need to install the mysqli PHP extension. Luckily, this container provides an easy way to do that.

Per its documentation on Docker Hub (http://hub.docker.com/_/php): "We provide the helper scripts `docker-php-ext-configure`, `docker-php-ext-install`, and `docker-php-ext-enable` to more easily install PHP extensions." They then continue to explain how to use these scripts in Dockerfiles. This is what you would want to do if you needed additional PHP modules or extensions. For demonstration purposes (and since I've already started my container) I'm going to do it manually for now, but I'll loop back around and show you how easy it is to create your own container image based on PHP, and bake in some of these extensions.

Connect to the web container.

```
docker exec -it web bash
```

Output:

```
root@5622d6799894:/var/www/html#
```

Use the helper script "`docker-php-ext-install`" to install the `mysqli` extension.

```
docker-php-ext-install mysqli
```

Now that the `mysqli` extension is installed, Apache needs to be restarted to see the change. Do that by simply stopping and starting the container.

First, exit the container and stop it.

```
exit  
docker stop web
```

Output:

```
web
```

Next, start the container back up, which will restart the Apache processes and load in the newly installed `mysqli` PHP extension.

```
docker start web
```

Output:

```
web
```

To ensure that Apache started, view the processes running in the container.

```
docker top web
```

Output:

UID	PID	PPID	C	STIME	T
root	2853814	2853791	0	14:43	p
-DFOREGROUND					
www-data	2853872	2853814	0	14:43	pts/
DFOREGROUND					
www-data	2853873	2853814	0	14:43	pts/
DFOREGROUND					
www-data	2853874	2853814	0	14:43	pts/
DFOREGROUND					
www-data	2853875	2853814	0	14:43	pts/
DFOREGROUND					
www-data	2853876	2853814	0	14:43	pts/
DFOREGROUND					

Let's finish setting up our blog by installing WordPress.

First, get the WordPress application files into the web server container.

```
docker volume inspect blog_web_data
```

Output:

```
[
```

```
{
  "CreatedAt": "2021-08-07T15:25:47-06:00",
  "Driver": "local",
  "Labels": {},
  "Mountpoint": "/var/lib/docker/volumes/blog_web_data/_data",
  "Name": "blog_web_data",
  "Options": {},
  "Scope": "local"
}
```

Add these files right into the volume from the Docker host.

First change into the directory.

```
cd /var/lib/docker/volumes/blog_web_data/_data
```

Now add the WordPress application archive. (Note: You can use the URL <https://wordpress.org/latest.tar.gz> to download the most recent version. However, I'm going to use a specific version, which works with this example at the time of publication.)

```
wget https://wordpress.org/wordpress-5.8.tar.gz
```

Output:

```
--2021-08-08 14:53:04-- https://wordpress.org/wordpress-5.8.tar.gz
Resolving wordpress.org (wordpress.org)... 198.143.164.252
Connecting to wordpress.org (wordpress.org)|198.143.164.252|:443... connect
HTTP request sent, awaiting response... 200 OK
Length: 15073609 (14M) [application/octet-stream]
Saving to: 'wordpress-5.8.tar.gz'

wordpress-5.8.tar.gz      100%
[=====]
22s

2021-08-08 14:53:26 (675 KB/s) - 'wordpress-5.8.tar.gz' saved [15073609/1
```

Now that the application archive has been downloaded, let's extract its contents.

```
tar xzf wordpress-5.8.tar.gz
```

Finally, chown the ownership of the WordPress application files to "www-data" and the group to "www-data".

```
chown -R www-data:www-data wordpress
```

If the user or group isn't available on the host system, you can simply run the "chown" command in the container.

```
docker exec -it web chown -R www-data:www-data  
/var/www/html/wordpress
```

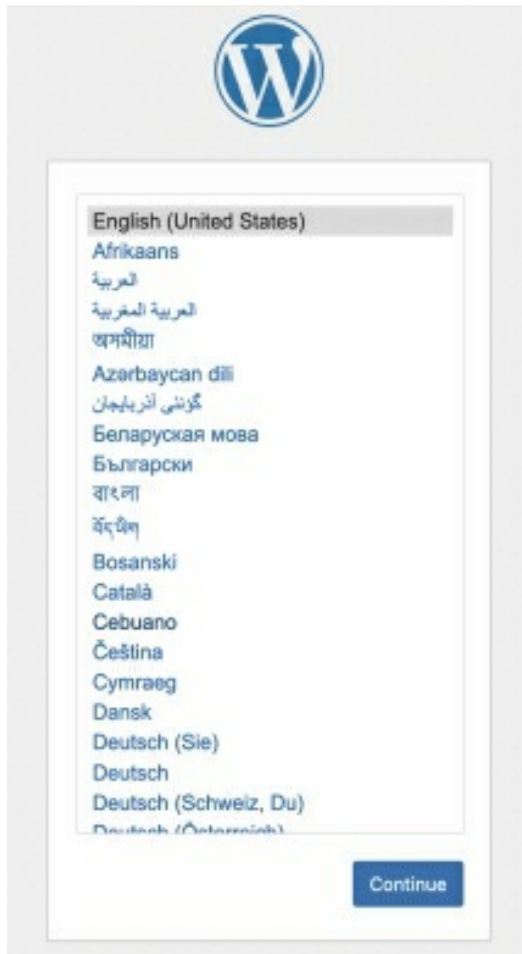
Now, I want to connect to the Docker host using my web browser. In my case, Docker is running on a Linux system on my private network. I'm accessing my Docker host via SSH from my laptop. So, to use the web browser on my laptop, I have to get the IP address of my Docker host. I can do that with the "ip a" command:

```
ip a
```

For me, it's 10.4.8.15. For you, it will most likely be something entirely different, so don't expect to use this exact IP address and get a result. You have to find the IP address for your Docker host if you want to access it from outside the host.

I'm going to put the IP address into my web browser. I don't have to specify a port because Docker is routing the default port for HTTP traffic, which is port 80, from the host to the web server container. However, I do need to supply the path "/wordpress", as I put the WordPress application files in that subdirectory.

When you connect to your Docker host via HTTP, you will be greeted with the first screen of the WordPress setup wizard:



Run through the setup steps. First, click "Continue." Next, click "Let's Go!". On the database connection details screen, use the following:

Database Name: wordpress

Username: root

Password: pw123

Database Host: db (This is the name of the container because "Docker is carrying out the name to IP address translation.)

Table Prefix: wp_



Below you should enter your database connection details. If you're not sure about these, contact your host.

Database Name	<input type="text" value="wordpress"/>	The name of the database you want to use with WordPress.
Username	<input type="text" value="root"/>	Your database username.
Password	<input type="text" value="pw123"/>	Your database password.
Database Host	<input type="text" value="db"/>	You should be able to get this info from your web host, if localhost doesn't work.
Table Prefix	<input type="text" value="wp_"/>	If you want to run multiple WordPress installations in a single database, change this.

Click "Submit" and on the next page click "Run the installation."

On the blog details page, I'm going to use the following information, but feel free to supply your own details.

Site Title: My Fun Blog


Username: redswingline

Password: stapler123

Your Email: root@localhost.localdomain

Information needed

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

Site Title	<input type="text" value="My Fun Blog"/>
Username	<input type="text" value="redswingline"/> <small>Usernames can have only alphanumeric characters, spaces, underscores, hyphens, periods, and the @ symbol.</small>
Password	<div><input type="password" value="stapler123"/> Medium</div> <div> Hide</div> <p>Important: You will need this password to log in. Please store it in a secure location.</p>
Your Email	<input type="text" value="root@localhost.localdomain"/> <small>Double-check your email address before continuing.</small>
Search engine visibility	<input type="checkbox"/> Discourage search engines from indexing this site <small>It is up to search engines to honor this request.</small>
<div>Install WordPress</div>	

Click "Install WordPress." Finally, click "Log in" to access WordPress.

Earlier, I mentioned that the right way to install the PHP modules is to create your own image using a Dockerfile. To be thorough, I'm going to demonstrate that now. This is going to be the simplest Dockerfile ever.

On the Docker host, change into your home directory.

```
cd
```

Now create a directory to hold the new Dockerfile.

```
mkdir php-mysqli
```

Change into the directory you created.

```
cd php-mysqli
```

Create a Dockerfile by opening it with a text editor such as nano. (NOTE: Use your favorite text editor for the operating system you are working on. For example, nano is not installed by default on Windows and some Linux distributions. Instead, you can use notepad for Windows and TextEdit for Mac.)

```
nano Dockerfile
```

Enter the following contents into the file. NOTE: It is very important that the contents are exact. Capitalization, punctuation, and spacing matter!

```
FROM php:apache  
RUN docker-php-ext-install mysqli
```

Save the Dockerfile. If you are using nano, type Ctrl-o. Next, hit "Enter" to save the file. Finally, type Ctrl-x to exit the nano editor.

By the way, mysqli stands for "MySQL improved," just in case you were wondering about the name of this particular PHP extension.

Now that you have a Dockerfile, you can build an image based on it.

```
docker build -t jasonc/php:mysqli .
```

Output:

```
Step 1/2 : FROM php:apache  
---> f935acc6d330  
Step 2/2 : RUN docker-php-ext-install mysqli  
---> Running in 62b8a32e2d75  
# Output continues...  
Removing intermediate container 62b8a32e2d75  
---> dd59fa94e573  
Successfully built dd59fa94e573  
Successfully tagged jasonc/php:mysqli
```

The naming convention used here is my Docker ID, or Docker Hub Login, or Docker Hub User ID, followed by a forward slash, followed by a repository.

In this case, that repository is "php". Following the repository is a colon and a tag. Let's choose to tag this particular image "mysql."

Now you can use your image locally, but if you want to use it on other Docker hosts, you need to push it to a repository, so those other Docker hosts can pull it down from there. Here, you can use Docker Hub as the default repository. Log in to Docker Hub with the "docker login" command.

```
docker login
```

Output:

```
Authenticating with existing credentials...  
WARNING! Your password will be stored unencrypted in  
/root/.docker/config.json.  
Configure a credential helper to remove this warning. See  
https://docs.docker.com/engine/reference/commandline/login/#credentials-store  
  
Login Succeeded
```

Now push your image to the registry.

```
docker push jasonc/php:mysql
```

Output:

```
The push refers to repository [docker.io/jasonc/php]  
6dd2fc27157b: Pushed  
9d780940e188: Mounted from library/php  
913e9b2f1f98: Mounted from library/php  
00e0c10a90fb: Mounted from library/php  
8676c125e011: Mounted from library/php  
d327808125b0: Mounted from library/php  
007070d11f9e: Mounted from library/php  
86b5e34374f3: Mounted from library/php  
c52f69d9a297: Mounted from library/php  
c24f05541085: Mounted from library/php
```

```
82f581b09510: Mounted from library/php
3b7e206db54c: Mounted from library/php
575b147c7c31: Mounted from library/php
814bff734324: Mounted from library/php
mysql: digest:
sha256:f3dd1260405fe8b0375d249e98ea255dc42e4d53c28396e136d66c578
size: 3244
```

Stop and remove the currently running container named "web".

```
docker stop web
```

Output:

```
web
```

Now destroy that container.

```
docker rm web
```

Output:

```
web
```

You're now ready to replace the function of that container, in my case, with the `jsonc/php:mysql` image. I'll start it with this command:

```
docker run --name web --network blog -p 80:80 --mount
src=blog_web_data,dst=/var/www/html -dit jsonc/php:mysql
```

Output:

```
7589b2ccd0ee948da51667b715a039be68226e287eed12196d71d4ec437eee2f
```

To verify that the new container works, visit the blog in your browser. For me, that address is `http://10.4.8.15/wordpress`. Remember that your Docker host IP will be different.

This demonstration should give you the theoretical concepts and practical steps for how you can deploy your own applications. If you are working on a custom written application, you'll have to Dockerize it on your own. However, if you are using a popular application, such as WordPress, the work has probably already been done for you.

For the purpose of this example, I knew an image for WordPress already existed; I just wanted to use a popular web application such as WordPress to teach the concepts. Here is the link to that image on Docker Hub:

https://hub.docker.com/_/wordpress

If you look at the Dockerfile for WordPress, you'll see it's an elaborate version of what we did in this chapter. They use the php:apache image as the base and then install all the required and desired PHP extensions. From there, they perform a bit of additional web server configuration, then finally install the WordPress application itself.

One final thing to mention is that there's an older, deprecated method that allows containers to communicate with each other. This involves using the "--link" option to the "docker run" command. Since it's deprecated, I didn't cover it in this chapter. I just wanted to mention it, so you know it exists and, more importantly, so you know how to use the methods you learned in this chapter, instead of using "--link".

Summary

In this chapter, you learned:

- How Docker utilizes the existing features of the Linux kernel to control network traffic to and from Docker containers.
- About the default bridge network used for containers started on a Docker host system.
- How to gather the details of all the networks in use on a Docker host and how to determine the exact IP address used by each container.
- How to create user-defined networks that allow for network separation among various containers.
- How to dockerize PHP web applications, using WordPress as an example.
- How Docker employs an embedded DNS server with built-in service

discovery for each of its user-defined networks.

EXERCISE: DOCKER NETWORKING

Note: You can download all the exercises contained in this book at <https://www.LinuxTrainingAcademy.com/docker>.

Goal:

The goal of this exercise is to create a user-defined network for a web application called Drupal.

Instructions:

Create a Network for the Drupal Web Application

Create a Docker bridged network named "drupal".

```
docker network create drupal
```

Create a Database Container

First, create a volume that will be used to store the database data. Name the volume "db".

```
docker volume create db
```

Next, start a container named "db", based on the "postgres" image with a tag of "11.5". Make sure it is connected to the "drupal" network. Also, mount the "db" volume to "/var/lib/postgresql/data" inside the container.

Create a database named "drupal" that can be accessed by a "drupal" user with the password "pw123". To do that, set the following environment

variables when starting the container:

POSTGRES_USER=drupal

POSTGRES_PASSWORD=pw123

POSTGRES_DB=drupal

(NOTE: The environment variables were determined by looking at the documentation for this image, as found on Docker Hub:

https://hub.docker.com/_/postgres)

Here is the command: (NOTE: Type this all on one line, as this is a single command.)

```
docker run -d --network drupal --name db -e
POSTGRES_DB=drupal
-e POSTGRES_USER=drupal -e
POSTGRES_PASSWORD=pw123
--mount src=db,dst=/var/lib/postgresql/data postgres:11.5
```

Inspect the "drupal" network to make sure the container is attached to it. You should see the "db" container listed in the "Containers" section of the output.

```
docker network inspect drupal
```

Create a Drupal Application Container

Start a container named "drupal" based on the "drupal" image with a tag of "8.7.7". Make sure it is connected to the "drupal" network. Also, publish port 80 on the host and map it to port 80 in the container.

```
docker run -d --network drupal --name drupal -p 80:80
```



```
drupal:8.7.7
```

Inspect the "drupal" network to make sure the container is attached to it. You should see the "drupal" container listed in the "Containers" section of the output.

```
docker network inspect drupal
```

Connect the Application to the Database

If you are running Docker on your local machine, then open up a web browser on your local machine and enter "http://localhost" into your web browser.

However, if the Docker host machine is a remote machine and you want to connect to from another system, then you'll need the IP address of your docker host machine. Determine the IP address of your Docker host machine and enter that into your web browser.

One way to get the IP address of your Docker host system is to use the "ip a" command:

```
ip a
```

(NOTE: If "ip a" doesn't work, try using the "ifconfig" command.)

Answer the installation prompts as follows.

Choose Language: English

Choose Profile: Standard

Set up Database:

Select "PostgreSQL"

Database name: drupal

Database username: drupal
Database password: pw123
Click "Advanced Options."
Host: db
Port number: 5432
It should look like this:

Database configuration

Database type *

- ☐ MySQL, MariaDB, Percona Server, or equivalent
- ☐ SQLite
- ☒ PostgreSQL

Database name *

Database username *

Database password

▼ ADVANCED OPTIONS

Host *

Port number

Table name prefix

If more than one application will be sharing this database, a unique table name prefix – such as *drupal_* – will prevent collisions.

Save and continue

Click "Save and continue".

Configure Site:

Site name: Test

Site email address: root@localhost.localdomain

Username: admin

Password: admin123

Confirm Password: admin123

Click "Save and continue".

You will now be presented with Drupal web application.

DOCKER SWARM

In this chapter, you're going to learn how to create and use a Docker Swarm, which is a cluster management and orchestration tool provided by Docker. You'll learn how to add nodes to the swarm and how to create services that run in the swarm.

Orchestrators

Before we begin, let's discuss what an orchestrator is and why you would want to use one when working with containers.

Orchestrators provide a number of different types of functionality, which assist in keeping a containerized application live and available to its users. They have many benefits, but two simple examples are: autoscaling, in which the orchestrator automatically adds more containers to deal with heavier demand and then removes the containers when the workload decreases, and garbage collection, where the orchestrator automatically terminates a container that has crashed. In short, orchestrators guarantee high availability by ensuring enough containers are running to provide the service.

In addition to the topics covered in this book so far, orchestrators can significantly enhance how you might use containers for serving your software or providing your application as a service.

Note that Docker sometimes refers to "Docker Swarm" as "swarm mode". So, if you see "Docker Swarm" or "swarm mode", know they are the same thing. For example, the documentation states: "Current versions of Docker include swarm mode for natively managing a cluster of Docker Engines called a *swarm*."

Docker allows users of "Docker Swarm" to decentralize the design of a cluster of host machines running "Docker Engine", which helps with maintaining higher levels of uptime for an application.

The terminology involves "nodes", which are the host machines, or more accurately, the "Docker Engines" in a "swarm", "managers" which assist with the orchestration but can also run services, and "workers" who, well, *work* to run those services.

Just like other orchestrators, "Docker Swarm" helps you automatically scale your application by introducing extra resources, such as containers, when required. These are called "replicas". Additionally, "swarm mode" will allow customizable resource quotas to stop single containers overloading a node, just as Docker does.

Docker also ensures that all communication between nodes is encrypted using TLS for security purposes. This means networking is safer by default, which Docker Swarm uses.

Using Docker Swarm will also allow you to create internal networks for specific communications between different aspects of an application. For example, this might be a back-end, private network, where two databases can talk to each other and update each other with their new data.

Another common benefit of running an orchestrator involves service discovery. With Docker Swarm, this involves a service being allocated a unique DNS name, which can assist with load balancing and resource naming. Names are automatically updated when services are altered, e.g., when a service scales up.

Example Initialization

On this working Docker CE installation, I will run the "init" command for Docker Swarm to demonstrate what you can expect if you want to set up nodes in swarm mode.

If you have multiple IP addresses, tell Docker Swarm which one to use. This particular system does have multiple IPs, so I'm going to have to specify which one. Here's what happens if you don't do that:

```
docker swarm init
```

Output:

```
Error response from daemon: could not choose an IP address to advertise
since this system has multiple addresses on different interfaces (10.0.2.15
on enp0s3 and 10.4.8.15 on enp0s8) - specify one with --advertise-addr
```

Docker is polite and tells you that you need to specify which IP address it should use. In my case, the public IP address of my system is 10.4.8.15. Docker Swarm will list the IP addresses it can use. Of course, you'll have to determine which IP address is assigned to your system.

```
docker swarm init --advertise-addr 10.4.8.15
```

Output:

```
Swarm initialized: current node (6kyhtq3ql0wgkrrwd01cnahu2) is now a
manager.
```

To add a worker to this swarm, run the following command:

```
docker swarm join --token SWMTKN-1-
2f45jyn2txhddvmcxfi9kme9li3z4bxyf7ps7mvk52n3dg5hk8-
czli52doszpr423ndph1clkvq 10.4.8.15:2377
```

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

As we can see from the output, the top line states that the current node has automatically become a manager and has been given a hashed name (shown in parentheses). If you are following along, you'll see a different hashed name.

Docker Swarm uses security tokens instead of passwords. In order to add a worker to the swarm, a token is needed to satisfy the requirements for credentials.

You can copy the "docker swarm join" command from the "docker swarm init" output. Then, go to another system where you have Docker CE installed. On that second Docker host system, have it join the swarm as a worker node.

```
docker swarm join --token SWMTKN-1-
```

```
2f45jyn2txhddvmcxfi9kme9li3z4bxyf7ps7mvk52n3dg5hk8-  
czli52doszpr423ndph1clkvq 10.4.8.15:2377
```

Output:

```
This node joined a swarm as a worker.
```

If you get an error such as "connection refused" or "no route to host", check there isn't a firewall blocking the required ports. If your Docker manager has a local firewall, make sure that TCP ports 2376, 2377, and 7946 are open, as well as UDP ports 7946 and 4789.

You can confirm you're part of the swarm by using the "docker info" command:

```
docker info
```

Output:

```
Client:
Context: default
Debug Mode: false
Plugins:
  app: Docker App (Docker Inc., v0.9.1-beta3)
  buildx: Build with BuildKit (Docker Inc., v0.6.1-docker)
  scan: Docker Scan (Docker Inc., v0.8.0)
Server:
Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
Images: 0
Server Version: 20.10.8
Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Native Overlay Diff: true
```

userxattr: false
Logging Driver: json-file
Cgroup Driver: cgroupfs
Cgroup Version: 1
Plugins:
 Volume: local
 Network: bridge host ipvlan macvlan null overlay
 Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk
syslog
Swarm: active
 NodeID: nsnjnarwowdbjaualuzzzhxpc
 Is Manager: false
 Node Address: 10.4.8.16
 Manager Addresses:
 10.4.8.15:2377
Runtimes: io.containerd.runc.v2 io.containerd.runtime.v1.linux runc
Default Runtime: runc
Init Binary: docker-init
containerd version: e25210fe30a0a703442421b0f60afac609f950a3
runc version: v1.0.1-0-g4144b63
init version: de40ad0
Security Options:
 apparmor
 seccomp
 Profile: default
Kernel Version: 4.15.0-153-generic
Operating System: Ubuntu 18.04.5 LTS
OSType: linux
Architecture: x86_64
CPUs: 2
Total Memory: 984.9MiB
Name: ubuntu2
ID:
GS5R:QDOI:3U7R:76WJ:SMLN:4TRP:SQB6:Y6LK:XFXM:LJTI:OIQV:P
Docker Root Dir: /var/lib/docker
Debug Mode: false
Registry: https://index.docker.io/v1/


```
Labels:
Experimental: false
Insecure Registries:
  127.0.0.0/8
Live Restore Enabled: false

WARNING: No swap limit support
```

You can see a line that states, "Swarm: active". The output also shows the node's ID and IP address, as well as the swarm manager's IP address.

Now, you can return to your first Docker system, your Docker Swarm manager system. List the nodes from the manager's perspective.

```
docker node ls
```

Output:

ID	HOSTNAME	STATUS	AVAILABILITY	MANAG
STATUS	ENGINE VERSION			
6kyhtq3ql0wgkrrwd01cnahu2				
* ubuntu	Ready	Active	Leader	20.10.8
nsnjnarwowdbjaualuzzzhxpc	ubuntu2	Ready	Active	20.10.8

This command only works on a manager in a swarm. Note that there are several commands that only work on a manager in a swarm. This is one reason why you might want to have multiple managers in your swarm. That way, if one manager becomes unavailable, you can still perform actions, such as creating services and listing nodes, on the remaining manager node(s) in the swarm. In any case, you can see the two nodes that are currently in your swarm.

You don't have to save this token information. Let's say you want to add another worker to your swarm later on. In that case, run this command on one of the managers in the swarm, and you'll get the required token.

```
docker swarm join-token worker
```

Output:

To add a worker to this swarm, run the following command:

```
docker swarm join --token SWMTKN-1-2f45jyn2txhddvmcxfi9kme9li3z4bxyf7ps7mvk52n3dg5hk8-czli52doszpr423ndph1clkvq 10.4.8.15:2377
```

Also, if you want to add another manager to the swarm, run this command to get the manager token.

```
docker swarm join-token manager
```

Output:

To add a manager to this swarm, run the following command:

```
docker swarm join --token SWMTKN-1-2f45jyn2txhddvmcxfi9kme9li3z4bxyf7ps7mvk52n3dg5hk8-dsmsqyv0kn2jhetnwjlrlrqz 10.4.8.15:2377
```

Now that we have a swarm, let's put it to use by creating a service using the "docker service create" command. You can supply a name for the service by using the "--name" option. In this example, we are going to name the service "web". We're also going to publish port 80 on the Docker hosts and direct that traffic to port 80 inside the containers for the service. Finally, supply an image name. In this example, we are going to use the "nginx:latest" image.

```
docker service create --name web -p 80:80 nginx:latest
```

Output:

```
pc50r1bofp5oms6ck7i39v3f7
overall progress: 1 out of 1 tasks
1/1:
running [=====]
verify: Service converged
```

The "docker service create" command is very similar to the "docker run" command. The "docker run" command starts a single container on a single Docker host, but the "docker service create" command creates one or more containers on one or more Docker hosts, all managed by Docker Swarm.

You can look at the services in the swarm with this command:

```
docker service ls
```

Output:

ID	NAME	MODE	REPLICAS	IMAGE	PORTS
pc50r1bofp5o	web	replicated	1/1	nginx:latest	*:80->80/tcp

You can also view information about the containers providing a service with "docker service ps", followed by the service name.

```
docker service ps web
```

Output:

ID	NAME	IMAGE	NODE	DESIRED
STATE	CURRENT	STATE	ERROR	PORTS
w1lkp39m5bh0	web.1	nginx:latest	ubuntu	Running
				Running 2 minutes ago

You can get details for the service with the "docker service inspect" command.

```
docker service inspect --pretty web
```

Output:

```
ID:          pc50r1bofp5oms6ck7i39v3f7
Name:        web
Service Mode: Replicated
Replicas:    1
Placement:
```

```
UpdateConfig:
Parallelism: 1
On failure:  pause
Monitoring Period: 5s
Max failure ratio: 0
Update order:  stop-first
RollbackConfig:
Parallelism: 1
On failure:  pause
Monitoring Period: 5s
Max failure ratio: 0
Rollback order:  stop-first
ContainerSpec:
Image:      nginx:latest@sha256:8f335768880da6baf72b70c701002b45f493
Init:      false
Resources:
Endpoint Mode: vip
Ports:
PublishedPort = 80
  Protocol = tcp
  TargetPort = 80
  PublishMode = ingress
```

The "--pretty" option makes Docker display the information in an easy-to-read format.

In the output, you'll find all the details for the service, including the ID, the name, etc.

Now that the service is running, you can access any of the nodes in the swarm on port 80 and that traffic will get directed to the service we just defined. It doesn't matter where in the swarm the actual container is running because Docker Swarm will route the traffic to the appropriate container. This is due to Docker Swarm's routing mesh. Let's try it out.

First, visit the IP address of the first node in the swarm:

```
curl http://10.4.8.15
```

Output:

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
  body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
  }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Here is the HTML output from the container providing the service.

Now run the same command, but against the other node in our swarm.

```
curl http://10.4.8.16
```

Output:

```
<!DOCTYPE html>
<html>
```

```
<head>
<title>Welcome to nginx!</title>
<style>
  body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
  }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

We get the exact same output. That's because Docker Swarm has redirected that traffic over its mesh network to the container that is running on the first Docker host. Docker Swarm does this all seamlessly.

Let's see what logs have been generated by our requests:

```
docker service logs web
```

Output:

```
web.1.w1lkp39m5bh0@ubuntu | 10.0.0.2 - - [09/Aug/2021:16:54:24
+0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.58.0" "-"
web.1.w1lkp39m5bh0@ubuntu | 10.0.0.3 - - [09/Aug/2021:16:54:39
+0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.58.0" "-"
```

By the way, Docker Swarm created an overlay network, which is doing all of this network routing of traffic.

```
docker network ls
```

Output:

NETWORK ID	NAME	DRIVER	SCOPE
8fd8f593aaa3	bridge	bridge	local
6fe6d58dd9a5	docker_gwbridge	bridge	local
59e0db25dd79	host	host	local
ix5br9f9klm6	ingress	overlay	swarm
da9fe47278be	none	null	local

Notice the network named "ingress", which is of type "overlay" and has a scope of "swarm". Let's take a closer look at that network.

```
docker network inspect ingress
```

Output:

```
[
  {
    "Name": "ingress",
    "Id": "ix5br9f9klm6g9aoh8biw0p53",
    "Created": "2021-08-09T16:26:04.328322594Z",
    "Scope": "swarm",
    "Driver": "overlay",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "10.0.0.0/24",
          "Gateway": "10.0.0.1"
        }
      ]
    }
  }
]
```

```

    ],
    "Internal": false,
    "Attachable": false,
    "Ingress": true,
    "ConfigFrom": {
        "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {
        "e77a04e8480bc876c62fffae25a4e56e12eecfeb346bf892855e
    {
        "Name": "web.1.w1lkp39m5bh0jhjg8bp4mneu8",
        "EndpointID":
"18431dbff1fb1a29cbd0dbe91374fcff2a6318751081f5cdb8f5f20a0c9ec359",
        "MacAddress": "02:42:0a:00:00:05",
        "IPv4Address": "10.0.0.5/24",
        "IPv6Address": ""
    },
    "ingress-sbox": {
        "Name": "ingress-endpoint",
        "EndpointID":
"5fb601fd806060c339db34b5e61960722b0cf5022a1c70dd5e6c5199f494f8f9",
        "MacAddress": "02:42:0a:00:00:02",
        "IPv4Address": "10.0.0.2/24",
        "IPv6Address": ""
    }
    },
    "Options": {
        "com.docker.network.driver.overlay.vxlanid_list": "4096"
    },
    "Labels": {},
    "Peers": [
        {
            "Name": "eb6e0c245e3d",
            "IP": "10.4.8.15"
        },
    ],

```



```
[
  {
    "Name": "cc5ba7657d52",
    "IP": "10.4.8.16"
  }
]
```

You can see the web container that is providing the service. In this example, it is named "web.1.w1lkp39m5bh0jhjg8bp4mneu8".

Let's say that this lone container isn't able to keep up with the workload. Well, it's easy to add another one by scaling the service.

```
docker service scale web=2
```

Output:

```
web scaled to 2
overall progress: 2 out of 2 tasks
1/2:
running [=====]
2/2:
running [=====]
verify: Service converged
```

Now we have two containers running the same image and providing the exact same service.

```
docker service ps web
```

Output:

ID	NAME	IMAGE	NODE	DESIRED	
STATE	CURRENT	STATE	ERROR	PORTS	
w1lkp39m5bh0	web.1	nginx:latest	ubuntu	Running	Running 20 minutes ago
go5lqhwyq4wk	web.2	nginx:latest	ubuntu2	Running	Running

45 seconds ago

As you can see, we have one container running on the Docker host named "ubuntu", while the other container is running on the Docker host named "ubuntu2".

If we list the Docker containers on the Docker host named "ubuntu", we will find one container running there.

```
docker ps
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
e77a04e8480b	nginx:latest	"/docker-entrypoint...."	21 minutes ago	Up 2 minutes
80/tcp	web.1.w1l	kp39m5bh0jhjg8bp4mneu8		

If you run the same command on the "ubuntu2" Docker host, you'll find another container running there.

```
docker ps
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	P
9e85ffaaee03	nginx:latest	"/docker-entrypoint...."	2 minutes ago	Up 2 minutes	
80/tcp	web.2.go5l	qhwyq4wkm8wd11z9i9o5z			

Docker Swarm does its best to spread out the containers amongst the nodes in the swarm. If you were to start more containers than nodes, then at least some nodes would have multiple containers running on them using the same image. Sometimes, this is exactly what you want. For example, if you were running an application in a container that could only handle one request at a time, and it blocks until it's finished with that request, then it would make sense to run multiple containers, even if they reside on the same node.

Let's move back to the Docker Swarm manager node. We'll go ahead and scale our service to more containers than we have nodes, and then inspect the result.

```
docker service scale web=4
```

Output:

```
web scaled to 4
overall progress: 4 out of 4 tasks
1/4:
running [=====
2/4:
running [=====
3/4:
running [=====
4/4:
running [=====
verify: Service converged
```

Let's look at all the containers that are providing the service.

```
docker service ps web
```

Output:

ID	NAME	IMAGE	NODE	DESIRED	
STATE	CURRENT	STATE	ERROR	PORTS	
w1lkp39m5bh0	web.1	nginx:latest	ubuntu	Running	Running 26 minutes ago
go5lqhwyq4wk	web.2	nginx:latest	ubuntu2	Running	Running 7 minutes ago
nfd9lvfqypn8	web.3	nginx:latest	ubuntu2	Running	Running 16 seconds ago
q6q4ip9hg46h	web.4	nginx:latest	ubuntu	Running	Running 17 seconds ago

Now we have multiple instances on each node in our swarm. Also, Swarm will try to distribute the workload amongst the containers, as it acts as a sort of load balancer. So, let's try this:

```
curl http://10.4.8.15
```

Output:

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
  body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
  }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Examine the logs for the web service.

```
docker service logs web
```

Output:

```
web.3.nfd9lvfqypn8@ubuntu2 | 10.0.0.2 - - [10/Aug/2021:16:19:53
+0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.58.0" "-"
```

That request was routed to web.3. Perform another request and see where it gets routed.

```
curl http://10.4.8.15
```

Output:

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
  body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
  }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Examine the logs.

```
docker service logs web
```

Output:

```
web.4.miwefjdyqtjo@ubuntu | 10.0.0.2 - - [10/Aug/2021:16:22:48  
+0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.58.0" "-"
```

That request was routed to web.4. As you can see, Docker is routing these requests to different containers in our swarm, even though we are accessing the one IP address on the manager node. Of course, we can access any of the IP addresses, of any node in the swarm.

If you need to take a Docker host offline for maintenance, you can update its availability status to drained. This moves all the containers off that host and onto other hosts in the swarm. Here is how to drain the worker node of ubuntu2 in our swarm:

```
docker node update --availability drain ubuntu2
```

Output:

```
ubuntu2
```

Now, let's look at our nodes.

```
docker node ls
```

Output:

ID	HOSTNAME	STATUS	AVAILABILITY	MANAG
STATUS	ENGINE	VERSION		
6kyhtq3ql0wgkrrwd01cna	ubuntu	Ready	Active	Leader
* ubuntu	Ready	Active	Leader	20.10.8
nsnjnarwowdbjaualuzzzhxpc	ubuntu2	Ready	Drain	20

You can see "Drain" in the "AVAILABILITY" column for the ubuntu2 node. All of those containers that were running on the ubuntu2 host are moving - or have moved - to this node in the swarm.

```
docker service ps web
```

Output:

ID	NAME	IMAGE	NODE	DESIRED	
STATE	CURRENT	STATE	ERROR	PORTS	
w1lkp39m5bh0	web.1	nginx:latest	ubuntu	Running	Running 26 minutes ago
wd1pmrpjj4ay	web.2	nginx:latest	ubuntu	Running	Running 25 seconds ago
go5lqhwyq4wk	_ web.2	nginx:latest	ubuntu2	Shutdown	Shutdown 27 seconds ago
qv49m24n185c	web.3	nginx:latest	ubuntu	Running	Running 24 seconds ago
nfd9lvfqypn8	_ web.3	nginx:latest	ubuntu2	Shutdown	Shutdown 27 seconds ago
q6q4ip9hg46h	web.4	nginx:latest	ubuntu	Running	Running 5 minutes ago

In this output, we can see that there are currently four running containers, all on the node named "ubuntu". Notice that the two containers that were running on the "ubuntu2" node are now showing as "Shutdown". Those were the containers that were moved over to the Docker manager node, called "ubuntu". Of course, if you had more nodes in the swarm, those containers could have ended up on those other nodes.

Once your maintenance on the node is done , you can set its availability back to active:

```
docker node update --availability active ubuntu2
```

Output:

```
ubuntu2
```

Check the status of the nodes in the swarm.

```
docker node ls
```

Output:

ID	HOSTNAME	STATUS	AVAILABILITY	MANAG
6kyhtq3ql0wgkrrwd01cnaHu2				
* ubuntu	Ready	Active	Leader	20.10.8
nsnjnarwowdbjaualuzzzhxpc	ubuntu2	Ready	Active	20

Note that when you bring a node back online or add a new node to the swarm, workloads don't automatically migrate to those new nodes. This is actually a design decision by the Docker team. They don't want to interrupt currently connected users, so they leave the containers wherever they are, if possible. New workloads will then go to the new nodes. So, if we scale our service up again, the new containers will start on the node we just brought back into the swarm.

Let's remind ourselves of the status of the containers in our swarm.

```
docker service ps web
```

Output:

ID	NAME	IMAGE	NODE	DESIRED
STATE	CURRENT	STATE	ERROR	PORTS
w1lkp39m5bh0	web.1	nginx:latest	ubuntu	Running
Running 26 minutes ago				
wd1pmprrpj4ay	web.2	nginx:latest	ubuntu	Running
Running 25 seconds ago				
go5lqhwyq4wk	_ web.2	nginx:latest	ubuntu2	Shutdown
Shutdown 27 seconds ago				
qv49m24n185c	web.3	nginx:latest	ubuntu	Running
Running 24 seconds ago				
nfd9lvfqypn8	_ web.3	nginx:latest	ubuntu2	Shutdown
Shutdown				


```
27 seconds ago
q6q4ip9hg46h web.4 nginx:latest ubuntu Running Running 5
minutes ago
```

There are four running containers on the "ubuntu" host. Let's scale this service and add two more containers.

```
docker service scale web=6
```

Output:

```
web scaled to 6
overall progress: 6 out of 6 tasks
1/6:
running [=====]
2/6:
running [=====]
3/6:
running [=====]
4/6:
running [=====]
5/6:
running [=====]
6/6:
running [=====]
verify: Service converged
```

Let's see where those newly created containers reside.

```
docker service ps web
```

Output:

ID	NAME	IMAGE	NODE	DESIRED	STATE	CURRENT STATE	ERROR	PORTS
w1lkp39m5bh0	web.1	nginx:latest	ubuntu	Running	Running	26 minutes ago		

wd1pmpjj4ay	web.2	nginx:latest	ubuntu	Running	Running 25 seconds ago
go5lqhwyq4wk	_ web.2	nginx:latest	ubuntu2	Shutdown	Shutdown 27 seconds ago
qv49m24n185c	web.3	nginx:latest	ubuntu	Running	Running 24 seconds ago
nfd9lvfqypn8	_ web.3	nginx:latest	ubuntu2	Shutdown	Shutdown 27 seconds ago
q6q4ip9hg46h	web.4	nginx:latest	ubuntu	Running	Running 5 minutes ago
5f5a8xtu37cj	web.5	nginx:latest	ubuntu2	Running	Running 26 seconds ago
2xilbwbhmpo1	web.6	nginx:latest	ubuntu2	Running	Running 25 seconds ago

Notice that both of the new containers are on the "ubuntu2" node. These two new containers are named "web.5" and "web.6".

If you know that a node in the swarm needs to go offline, the best strategy is to drain it, just like we demonstrated. However, unscheduled outages do happen. When a Docker node becomes unavailable, Docker Swarm redistributes the workload just like it did when a node was drained. Let's pretend one of our nodes has suffered a power outage. We'll do this by shutting down our worker node named "ubuntu2". I've switched to that host, and I am going to run this command to power off that host:

```
shutdown -h now
```

Back on the remaining node in the swarm named "ubuntu", I'm going to list the containers.

```
docker service ps web
```

Output:

ID	NAME	IMAGE	NODE	DESIRED	
STATE	CURRENT	STATE	ERROR	PORTS	
w1lkp39m5bh0	web.1	nginx:latest	ubuntu	Running	Running 26

```

minutes ago
wd1pmprrjj4ay web.2 nginx:latest ubuntu Running Running 25
seconds ago
go5lqhwyq4wk \_ web.2
nginx:latest ubuntu2 Shutdown Shutdown 27 seconds ago
qv49m24n185c web.3 nginx:latest ubuntu Running Running 24
seconds ago
nfd9lvfqypn8 \_ web.3 nginx:latest ubuntu2 Shutdown Shutdown
27 seconds ago
q6q4ip9hg46h web.4 nginx:latest ubuntu Running Running 5
minutes ago
o7r74pugnc3m web.5 nginx:latest ubuntu Running Running 10
seconds ago
5f5a8xtu37cj \_ web.5 nginx:latest ubuntu2 Shutdown Running 5
minutes ago
8pr8625fjezm web.6 nginx:latest ubuntu Running Running 9
seconds ago
2xilbwbhmpo1 \_ web.6 nginx:latest ubuntu2 Shutdown Running
5 minutes ago

```

You can see that the containers that were running on the "ubuntu2" host are now showing as "shutdown" and that they have restarted here on the "ubuntu" node.

Not only can you scale the service up, but you can also scale the service back down if you need or want to. Let's scale this service down to just one container.

```
docker service scale web=1
```

Output:

```

web scaled to 1
overall progress: 1 out of 1 tasks
1/1:
verify: Service converged

```

Check that only one container is running for our service.

```
docker service ps web
```

Output:

ID	NAME	IMAGE	NODE	DESIRED	
STATE	CURRENT	STATE	ERROR	PORTS	
w1lkp39m5bh0	web.1	nginx:latest	ubuntu	Running	Running 29 minutes ago
wd1pmrpjj4ay	web.2	nginx:latest	ubuntu2	Shutdown	Shutdown 25 seconds ago
qv49m24n185c	web.3	nginx:latest	ubuntu2	Shutdown	Shutdown 24 seconds ago
o7r74pugnc3m	web.5	nginx:latest	ubuntu2	Shutdown	Shutdown 10 seconds ago
8pr8625fjezm	web.6	nginx:latest	ubuntu2	Shutdown	Shutdown 9 seconds ago

In this output, you can see that there is only one container in a "Running" state in our swarm.

If you no longer need a service, you can remove it like so:

```
docker service rm web
```

Output:

```
web
```

You can confirm the service has been removed with this command:

```
docker service ps web
```

Output:

```
no such service: web
```

Summary

In this chapter, you learned:

- How to initialize a Docker Swarm.
- How to add nodes to the swarm.
- How to deploy a service using the swarm.
- How to scale the service, both up and down.
- How to remove the service from the swarm.

EXERCISE: DEPLOYING A PRIVATE DOCKER REGISTRY

Note: You can download all the exercises contained in this book at <https://www.LinuxTrainingAcademy.com/docker>.

Goal:

The goal of this exercise is to deploy a private Docker registry. Additionally, you will learn how to pull and push images to the registry you create.

Instructions:

Create and Start a Registry Container

First, create a volume that will be used to store the registry data. Name the volume "registry".

```
docker volume create registry
```

Next, start a container named "registry", based on the "registry" image with the tag "2". Mount the "registry" volume to "/var/lib/registry" inside the container. Finally, publish port 5000 on the host and map it to port 5000 in the container.

Here is the command: (NOTE: Type this all on one line, as this is a single command.)

```
docker run -d --name registry --mount
```

```
src=registry,dst=/var/lib/registry -p 5000:5000 registry:2
```

(NOTE: For more information about the official Docker registry image, visit https://hub.docker.com/_/registry.)

Copy an Image from Docker Hub to Your Registry

Pull down the "nginx:latest" image to your Docker host system.

```
docker pull nginx:latest
```

(NOTE: You could also use "docker pull nginx", as the "latest" tag is assumed if no tag is specified.)

Tag the image as "localhost:5000/nginx:latest". Note that the registry name is "localhost:5000", the repository is "nginx" and the tag is "latest".

```
docker tag nginx:latest localhost:5000/nginx:latest
```

Check that the image has been tagged.

```
docker images
```

You'll notice that the "nginx:latest" image and the "localhost:5000/nginx:latest" images have the same Image ID.

Push the retagged nginx image into the private registry.

```
docker push localhost:5000/nginx:latest
```

You can check that the image has been pushed to the private registry using the registry's HTTP API. (For details on the API, see <https://docs.docker.com/registry/spec/api/>.)

```
curl http://localhost:5000/v2/_catalog
```

You should see "nginx" listed as a repository.

```
{"repositories":["nginx"]}
```

Remove the Image from the Docker Host

Remove the "nginx:latest" image that you downloaded from Docker Hub:

```
docker rmi nginx:latest
```

Confirm that the image has been removed with the following command:

```
docker images
```

Note that deleting the "nginx:latest" image does not remove the other image with the same Image ID, which is localhost:5000/nginx:latest. Of course, it also does not remove it from the private registry.

Next, remove the localhost:5000/nginx:latest image.

```
docker rmi localhost:5000/nginx:latest
```

You'll notice that instead of only untagging the image, it also actively deletes the layers that make up the image.

Confirm that the image has been removed with the following command:

```
docker images
```

Even though the image does not exist on your local Docker host machine, it still exists in the registry.

Pull the Image from the Private Docker Registry

To prove that the image is stored in the private registry, pull it from the private registry to your Docker host system.


```
docker pull localhost:5000/nginx
```

You will now see the image listed:

```
docker images
```

Start a Container Using the Image

Next, start a container using the image.

```
docker run -d localhost:5000/nginx  
docker ps
```

OPTIONAL: Access the Private Docker Registry from a Remote Docker Host

First, determine the IP address of your Docker host machine. One way to get the IP address of your Docker host system is to use the "ip a" command:

```
ip a
```

(NOTE: If "ip a" doesn't work, try using the "ifconfig" command.)

Next, connect to another Docker host system. We'll call this one "Docker Host 2." (If you need to install Docker, see the exercises on installing Docker.)

Because the private registry that you configured isn't using a TLS certificate, you'll need to tell your Docker Host 2 system not to attempt to use HTTPS and instead use HTTP to communicate with the registry. To do that, add the following to the /etc/docker/daemon.json file:

```
echo '{ "insecure-registries" : ["10.4.8.15:5000"] }' >>  
/etc/docker/daemon.json
```

(NOTE: If your Docker host is a Windows Server, the location of the

daemon.json file is: C:\ProgramData\docker\config\daemon.json.)

Next, restart the Docker engine so that it picks up the new configuration:

```
systemctl restart docker
```

Now you are ready to pull the "nginx:latest" image from the original Docker Host system.

```
docker pull  
IP_ADDRESS_OF_YOUR_DOCKER_HOST:5000/nginx
```

In my case, the IP address of my original Docker host is 10.4.8.15, so I would use this command:

```
docker pull 10.4.8.15:5000/nginx
```

(NOTE: If you get an error such as "http: server gave HTTP response to HTTPS client", check the previous steps and ensure the contents of the /etc/docker/daemon.json file are correct and that the Docker engine has been restarted.)

Verify the image has been downloaded.

```
docker images
```

Next, start a container using this image.

```
docker run -d  
IP_ADDRESS_OF_YOUR_DOCKER_HOST:5000/nginx
```

Again, as my IP is 10.4.8.15, I would use this command:

```
docker run -d 10.4.8.15:5000/nginx
```

Check that the container is running.

```
docker ps
```

NOTE: Because no authentication is required to access the registry, be sure to only deploy it on a secure local network. Alternatively, you can add basic authentication as described in the Docker Registry documentation:

<https://docs.docker.com/registry/> Authentication was not covered in this lesson as it requires DNS configuration, which is beyond the scope of this book.

THE END AND THE BEGINNING

Even though this is the end of this book, I sincerely hope that it is just the beginning of your Docker journey. Docker has been growing steadily in popularity since its release, playing a major role in microservices architectures as well as in DevOps methodologies. The possibilities for learning, exploring, and growing are endless.

OTHER BOOKS BY THE AUTHOR

Linux for Beginners: An Introduction to the Linux Operating System and Command Line

<https://www.linuxtrainingacademy.com/linux>

Command Line Kung Fu: Bash Scripting Tricks, Linux Shell Programming Tips, and Bash One-liners

<http://www.linuxtrainingacademy.com/command-line-kung-fu-book>

Shell Scripting: How to Automate Command Line Tasks Using Bash Scripting and Shell Programming

<http://www.linuxtrainingacademy.com/shell-book>

Python Programming for Beginners: An Introduction to the Python Computer Language and Computer Programming

<https://www.linuxtrainingacademy.com/python-book>

COURSES BY THE AUTHOR

You'll find courses by the author at
<https://courses.linuxtrainingacademy.com>.

Docker: A Project-Based Approach to Learning
<https://courses.linuxtrainingacademy.com/course/docker>

Linux in the Real World
<https://courses.linuxtrainingacademy.com/lrw>

Learn Linux in 5 Days
<https://courses.linuxtrainingacademy.com/course/learn-linux-in-5-days>

Linux System Administration
<https://courses.linuxtrainingacademy.com/course/linux-admin>

Linux Shell Scripting Projects
<https://courses.linuxtrainingacademy.com/course/linux-shell-scripting>

Vim Masterclass
<https://courses.linuxtrainingacademy.com/course/vim-masterclass>

Linux Security and Hardening
<https://courses.linuxtrainingacademy.com/course/linux-security>

Python Programming
<https://courses.linuxtrainingacademy.com/course/python-programming>