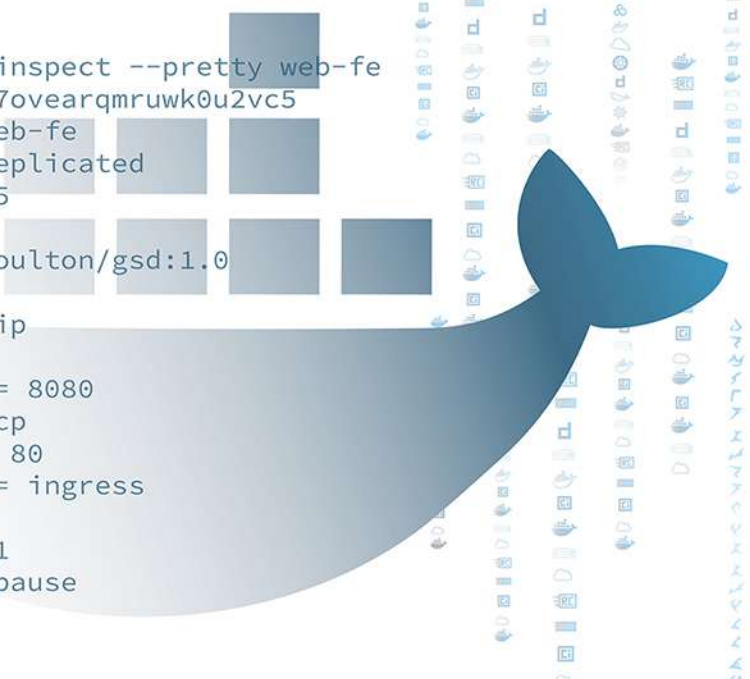


```
$ docker service inspect --pretty web-fe
ID:          z7ovearqmruwk0u2vc5
Name:        web-fe
Service Mode: Replicated
  Replicas:   5
ContainerSpec:
  Image:      nigelpoulton/gsd:1.0
  init:       false
Endpoint Mode: vip
Ports:
  PublishedPort = 8080
  Protocol      = tcp
  TargetPort    = 80
  PublishMode   = ingress
UpdateConfig:
  Parallelism:   1
  On failure:    pause
```



DOCKER DEEP DIVE

ZERO TO DOCKER IN A SINGLE BOOK

2023 Edition

By Docker Captain
Nigel Poulton

Docker Deep Dive

Zero to Docker in a single book

2023 Edition

Weapons-grade container learning

Nigel Poulton @nigelpoulton

About this edition

This edition was published in June 2023.

In writing this edition, I've gone over every word in every chapter ensuring everything is up-to-date with the latest editions of Docker and latest trends in the industry. I've also removed repetitions and made every chapter more concise.

Major changes include:

- Added sections on multi-platform builds with `buildx`
- Updated the Compose chapter to be in-line with the Compose Spec
- New example apps
- Updated all images to higher quality
- Added Multipass as a simple way to get a Docker lab

Enjoy the book and get ready to master containers!

A handwritten signature in black ink, appearing to read 'Nigel Poulton', with a stylized, flowing script.

Nigel Poulton

(c) 2023 Nigel Poulton Ltd

Huge thanks to my wife and kids for putting up with a geek in the house who genuinely thinks he's a bunch of software running inside of a container on top of midrange biological hardware. It can't be easy living with me!

Massive thanks as well to everyone who watches my Pluralsight videos. I love connecting with you and really appreciate all the feedback I've gotten over the years. This was one of the major reasons I decided to write this book! I hope it'll be an amazing tool to help you drive your careers even further forward.

About the author

Nigel is a technology geek who spends his life creating books, training videos, and online hands-on training. He is the author of best-selling books on Docker and Kubernetes and the most popular online training videos on the same topics. Nigel is a Docker Captain and always playing with new technology – his latest interest is WebAssembly on the server (Wasm). Previously, Nigel has held various senior infrastructure roles within large enterprises.

He is fascinated with technology and often daydreams about it. In his free time, he enjoys reading and watching science fiction. He wishes he lived in the future and could explore space-time, the universe, and other mind-bending phenomena. He is passionate about learning, cars, and football (soccer). He lives in England with his fabulous wife and three children.

Contents

0: About the book	1
Why should I read this book or care about Docker?	1
What if I'm not a developer	1
Should I buy the book if I've already watched your video training courses?	1
How the book is organized	2
Editions of the book	3
Having problems getting the latest updates on your Kindle?	3
Leave a review	3

Part 1: The big picture stuff **5**

1: Containers from 30,000 feet	7
The bad old days	7
Hello VMware!	7
VMwarts	8
Hello Containers!	8
Linux containers	8
Hello Docker!	9
Docker and Windows	9
Windows containers vs Linux containers	10
What about Mac containers?	10
What about Kubernetes	10
Chapter Summary	11
2: Docker	13
Docker - The TLDR	13
Docker, Inc.	13
The Docker technology	14
The Open Container Initiative (OCI)	16
Chapter summary	17
3: Installing Docker	19

Docker Desktop	19
Installing Docker with Multipass	23
Installing Docker on Linux	24
Play with Docker	25
Chapter Summary	25
4: The big picture	27
The Ops Perspective	27
The Dev Perspective	32
Chapter Summary	35
 Part 2: The technical stuff	 37
5: The Docker Engine	39
Docker Engine - The TLDR	39
Docker Engine - The Deep Dive	40
Chapter summary	47
6: Images	49
Docker images - The TLDR	49
Docker images - The deep dive	50
Images - The commands	71
Chapter summary	71
7: Containers	73
Docker containers - The TLDR	73
Docker containers - The deep dive	74
Containers - The commands	90
Chapter summary	91
8: Containerizing an app	93
Containerizing an app - The TLDR	93
Containerizing an app - The deep dive	94
Containerizing an app - The commands	114
Chapter summary	114
9: Multi-container apps with Compose	117
Deploying apps with Compose - The TLDR	117
Deploying apps with Compose - The Deep Dive	118
Deploying apps with Compose - The commands	132
Chapter Summary	133
10: Docker Swarm	135

Docker Swarm - The TLDR	135
Docker Swarm - The Deep Dive	136
Docker Swarm - The Commands	160
Chapter summary	161
11: Docker Networking	163
Docker Networking - The TLDR	163
Docker Networking - The Deep Dive	164
Docker Networking - The Commands	187
Chapter Summary	187
12: Docker overlay networking	189
Docker overlay networking - The TLDR	189
Docker overlay networking - The deep dive	189
Docker overlay networking - The commands	201
Chapter Summary	202
13: Volumes and persistent data	203
Volumes and persistent data - The TLDR	203
Volumes and persistent data - The Deep Dive	203
Volumes and persistent data - The Commands	213
Chapter Summary	213
14: Deploying apps with Docker Stacks	215
Deploying apps with Docker Stacks - The TLDR	215
Deploying apps with Docker Stacks - The Deep Dive	216
Deploying apps with Docker Stacks - The Commands	230
Chapter Summary	231
15: Security in Docker	233
Security in Docker - The TLDR	233
Security in Docker - The deep dive	234
Chapter Summary	254
16: What next	255
Feedback and reviews	255

0: About the book

This is a book about Docker, no prior knowledge required. In fact, the motto of the book is **Zero to Docker in a single book**.

So, if you're involved in the development and operations of cloud-native microservices apps and need to learn Docker, or if you want to be involved in that stuff, this book is dedicated to you.

Why should I read this book or care about Docker?

Docker is here and there's no point hiding. If you want the best jobs working on the best technologies, you need to know Docker and containers. Docker and containers are central to Kubernetes, and knowing how they work will help you learn Kubernetes. They're also positioned well for emerging cloud technologies such as WebAssembly on the server.

What if I'm not a developer

If you think Docker is just for developers, prepare to have your world turned upside-down.

Most applications, even the funky cloud-native microservices ones, need high-performance production-grade infrastructure to run on. If you think traditional developers are going to take care of that, think again. To cut a long story short, if you want to thrive in the modern cloud-first world, you need to know Docker. But don't stress, this book will give you all the skills you need.

Should I buy the book if I've already watched your video training courses?

The choice is yours, but I normally recommend people watch my videos **and** read my books. And no, it's not to make me rich. Learning via different mediums is a proven way to learn fast. So, I recommend you read my books, watch my videos, and get as much hands-on experience as possible.

Also, if you like my video courses¹ you'll probably like the book. If you don't like my video courses you probably won't like the book.

If you haven't watched my video courses, you should! They're fast-paced, lots of fun, and get *rave reviews*.

How the book is organized

I've divided the book into two sections:

1. The big picture stuff
2. The technical stuff

The big picture stuff covers things like:

- What is Docker
- Why do we have containers
- What does jargon like “cloud-native” and “microservices” mean...

It's the kind of stuff that you need to know if you want a rounded knowledge of Docker and containers.

The technical stuff is where you'll find everything you need to start working with Docker. It gets into the detail of *images*, *containers*, and the increasingly important topic of *orchestration*. It even covers the stuff that enterprises love — TLS, image signing, high-availability, backups, and more.

Each chapter covers theory and includes plenty of commands and examples.

Most of the chapters in the *technical stuff* section are divided into three parts:

- The TLDR
- The Deep Dive
- The Commands

The TLDR gives you two or three paragraphs that you can use to explain the topic at the coffee machine. They're also a great place to remind you what something is about.

The Deep Dive explains how things work and gives examples.

The Commands lists all the relevant commands in an easy-to-read list with brief reminders of what each one does.

I think you'll love that format.

¹<https://app.pluralsight.com/library/search?q=nigel+poulton>

Editions of the book

Docker and the cloud-native ecosystem is developing fast. As a result, I'm committed to updating the book approximately every year.

If that sounds excessive, welcome to the new normal.

We no-longer live in a world where a 4-year-old book on a technology like Docker is valuable. That makes my life as an author really hard, but I'm not going to argue with the truth.

Having problems getting the latest updates on your Kindle?

It's come to my attention that Kindle doesn't always download the latest version of the book. To fix this:

Go to <http://amzn.to/2l53jdg>

Under **Quick Solutions** (on the left) select **Digital Purchases**. Search for your purchase of **Docker Deep Dive** kindle edition and select **Content and Devices**. Your purchase should show up in the list with a button that says "Update Available". Click that button. Delete your old version on your Kindle and download the new one.

If this doesn't work, contact Kindle support and they'll resolve the issue for you.
https://kdp.amazon.com/en_US/self-publishing/contact-us/.

Leave a review

Last but not least... be a legend and write a quick review on Amazon and Goodreads. You can even do this if you bought the book from a different reseller.

That's everything. Let's get rocking with Docker!

Part 1: The big picture stuff

1: Containers from 30,000 feet

Containers have taken over the world!

In this chapter we'll get into things like; why we have containers, what they do for us, and where we can use them.

The bad old days

Applications are at the heart of businesses. If applications break, businesses break. Sometimes they even go bust. These statements get truer every day!

Most applications run on servers. In the past we could only run one application per server. The open-systems world of Windows and Linux just didn't have the technologies to safely and securely run multiple applications on the same server.

As a result, the story went something like this... Every time the business needed a new application, the IT department would buy a new server. Most of the time nobody knew the performance requirements of the new application, forcing the IT department to make guesses when choosing the model and size of the server to buy.

As a result, IT did the only thing it could do — it bought big fast servers that cost a lot of money. After all, the last thing anyone wanted, including the business, was under-powered servers unable to execute transactions and potentially losing customers and revenue. So, IT bought big. This resulted in over-powered servers operating as low as 5-10% of their potential capacity. **A tragic waste of company capital and environmental resources!**

Hello VMware!

Amid all of this, VMware, Inc. gave the world a gift — the virtual machine (VM). And almost overnight, the world changed into a much better place. We finally had a technology that allowed us to run multiple business applications safely on a single server. Cue wild celebrations!

This was a game changer. IT departments no longer needed to procure a brand-new oversized server every time the business needed a new application. More often than not, they could run new apps on existing servers that were sitting around with spare capacity.

All of a sudden, we could squeeze massive amounts of value out of existing corporate assets, resulting in a lot more bang for the company's buck (\$).

VMwarts

But... and there's always a *but!* As great as VMs are, they're far from perfect!

The fact that every VM requires its own dedicated operating system (OS) is a major flaw. Every OS consumes CPU, RAM and other resources that could otherwise be used to power more applications. Every OS needs patching and monitoring. And in some cases, every OS requires a license. All of this results in wasted time and resources.

The VM model has other challenges too. VMs are slow to boot, and portability isn't great — migrating and moving VM workloads between hypervisors and cloud platforms is harder than it needs to be.

Hello Containers!

For a long time, the big web-scale players, like Google, have been using container technologies to address the shortcomings of the VM model.

In the container model, the container is roughly analogous to the VM. A major difference is that containers do not require their own full-blown OS. In fact, all containers on a single host share the host's OS. This frees up huge amounts of system resources such as CPU, RAM, and storage. It also reduces potential licensing costs and reduces the overhead of OS patching and other maintenance. Net result: savings on the time, resource, and capital fronts.

Containers are also fast to start and ultra-portable. Moving container workloads from your laptop, to the cloud, and then to VMs or bare metal in your data center is a breeze.

Linux containers

Modern containers started in the Linux world and are the product of an immense amount of work from a wide variety of people over a long period of time. Just as one example, Google LLC has contributed many container-related technologies to the Linux kernel. Without these, and other contributions, we wouldn't have modern containers today.

Some of the major technologies that enabled the massive growth of containers in recent years include; **kernel namespaces**, **control groups**, **capabilities**, and of course **Docker**.

To re-emphasize what was said earlier — the modern container ecosystem is deeply indebted to the many individuals and organizations that laid the strong foundations that we currently build on. Thank you!

Despite all of this, containers remained complex and outside of the reach of most organizations. It wasn't until Docker came along that containers were effectively democratized and accessible to the masses.

Note: There are many operating system virtualization technologies similar to containers that pre-date Docker and modern containers. Some even date back to System/360 on the Mainframe. BSD Jails and Solaris Zones are some other well-known examples of Unix-type container technologies. However, in this book we are restricting our conversation to *modern containers* made popular by Docker.

Hello Docker!

We'll talk about Docker in a bit more detail in the next chapter. But for now, it's enough to say that Docker was the magic that made Linux containers usable for mere mortals. Put another way, Docker, Inc. made containers simple!

Docker and Windows

Microsoft has worked extremely hard to bring Docker and container technologies to the Windows platform.

At the time of writing, Windows desktop and server platforms support both of the following:

- Windows containers
- Linux containers

Windows containers run Windows apps that require a host system with a Windows kernel. Windows 10 and Windows 11, as well as all modern versions of Windows Server, have native support Windows containers.

Any Windows host running the WSL 2 (Windows Subsystem for Linux) can also run Linux containers. This makes Windows 10 and 11 great platforms for developing and testing Windows **and** Linux containers.

However, despite all of the work Microsoft has done developing *Windows containers*, the vast majority of containers are Linux containers. This is because Linux containers are smaller and faster, and the majority of tooling exists for Linux.

All of the examples in this edition of the book are Linux containers.

Windows containers vs Linux containers

It's vital to understand that a container shares the kernel of the host it's running on. This means containerized Windows apps need a host with a Windows kernel, whereas containerized Linux apps need a host with a Linux kernel. Only... it's not always that simple.

As previously mentioned, it's possible to run Linux containers on Windows machines with the WSL 2 backend installed.

What about Mac containers?

There is currently no such thing as Mac containers.

However, you can run Linux containers on your Mac using *Docker Desktop*. This works by seamlessly running your containers inside of a lightweight Linux VM on your Mac. It's extremely popular with developers, who can easily develop and test Linux containers on their Mac.

What about Kubernetes

Kubernetes is an open-source project out of Google that has quickly emerged as the de facto orchestrator of containerized apps. That's just a fancy way of saying *Kubernetes is the most popular tool for deploying and managing containerized apps*.

Note: A containerized app is an application running as a container.

Kubernetes used to use Docker as its default *container runtime* – the low-level technology that pulls images and starts and stops containers. However, modern Kubernetes clusters have a pluggable container runtime interface (CRI) that makes it easy to swap-out different container runtimes. At the time of writing, most new Kubernetes clusters use **containerd**. We'll cover more on containerd later in the book, but for now it's enough to know that containerd is the small specialized part of Docker that does the low-level tasks of starting and stopping containers.

Check out these resources if you need to learn Kubernetes. **Quick Start Kubernetes** is ~100 pages and will get you up-to-speed with Kubernetes in a day! **The Kubernetes Book** is a lot more comprehensive and will get you very close to being a Kubernetes expert.



★★★★☆ 75 ratings



★★★★☆ 1,503 ratings

Chapter Summary

We used to live in a world where every time the business needed a new application we had to buy a brand-new server. VMware came along and allowed us to drive more value out of new and existing IT assets. As good as VMware and the VM model is, it's not perfect. Following the success of VMware and hypervisors came a newer more efficient and portable virtualization technology called containers. But containers were initially hard to implement and were only found in the data centers of web giants that had Linux kernel engineers on staff. Docker came along and made containers easy and accessible to the masses.

Speaking of Docker... let's go find who, why, and what Docker is!

2: Docker

No book or conversation about containers is complete without talking about Docker. But when we say “Docker”, we can be referring to either of the following:

1. Docker, Inc. the company
2. Docker the technology

Docker - The TLDR

Docker is software that runs on Linux and Windows. It creates, manages, and can even orchestrate containers. The software is currently built from various tools from the *Moby* open-source project. Docker, Inc. is the company that created the technology and continues to create technologies and solutions that make it easier to get the code on your laptop running in the cloud.

That’s the quick version. Let’s dive a bit deeper.

Docker, Inc.

Docker, Inc. is a technology company based out of San Francisco founded by French-born American developer and entrepreneur Solomon Hykes. Solomon is no longer at the company.



Figure 2.1 Docker, Inc. logo.

The company started out as a platform as a service (PaaS) provider called *dotCloud*. Behind the scenes, the dotCloud platform was built on Linux containers. To help create and manage these containers, they built an in-house tool that they eventually nick-named “Docker”. And that’s how the Docker technology was born!

It’s also interesting to know that the word “Docker” comes from a British expression meaning **dock** worker — somebody who loads and unloads cargo from ships.

In 2013 they got rid of the struggling PaaS side of the business, rebranded the company as “Docker, Inc.”, and focussed on bringing Docker and containers to the world. They’ve been immensely successful in this endeavour.

Throughout this book we’ll use the term “Docker, Inc.” when referring to Docker the company. All other uses of the term “Docker” will refer to the technology.

The Docker technology

When most people talk about Docker, they’re referring to the technology that runs containers. However, there are at least three things to be aware of when referring to Docker as a technology:

1. The runtime
2. The daemon (a.k.a. engine)
3. The orchestrator

Figure 2.2 shows the three layers and will be a useful reference as we explain each component. We’ll get deeper into each later in the book.

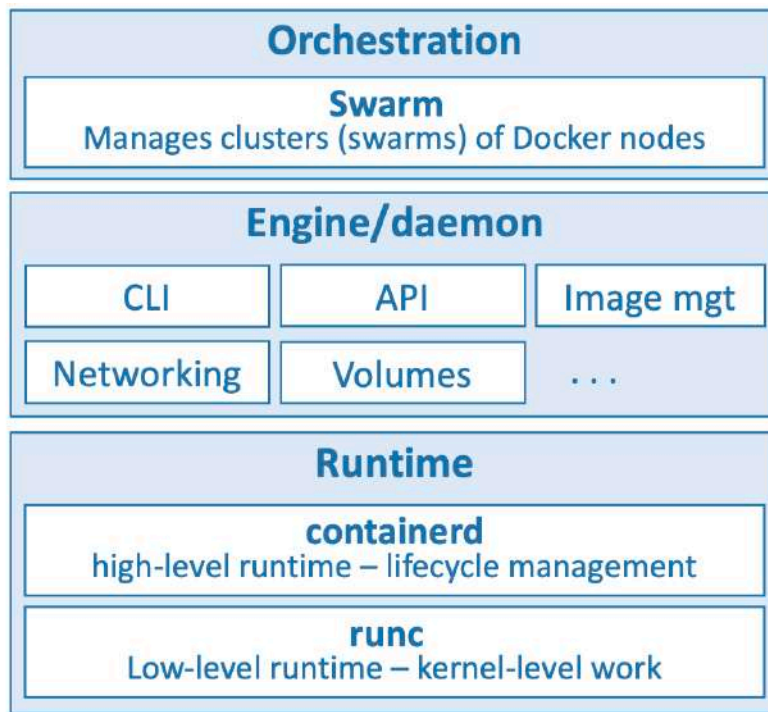


Figure 2.2 Docker architecture.

The runtime operates at the lowest level and is responsible for starting and stopping containers (this includes building all of the OS constructs such as namespaces and cgroups). Docker implements a tiered runtime architecture with high-level and low-level runtimes that work together.

The low-level runtime is called `runc` and is the reference implementation of Open Containers Initiative (OCI) runtime-spec. Its job is to interface with the underlying OS and start and stop containers. Every container on a Docker node was created and started by an instance of `runc`.

The higher-level runtime is called `containerd`. This manages the entire container lifecycle including pulling images and managing `runc` instances. `containerd` is pronounced “container-dee” and is a graduated CNCF project used by Docker and Kubernetes.

A typical Docker installation has a single long-running `containerd` process instructing `runc` to start and stop containers. `runc` is never a long-running process and exits as soon as a container is started.

The Docker daemon (`dockerd`) sits above `containerd` and performs higher-level tasks such as exposing the Docker API, managing images, managing volumes, managing networks, and more...

A major job of the Docker daemon is to provide an easy-to-use standard interface that abstracts the lower levels.

Docker also has native support for managing clusters of nodes running Docker. These clusters are called *swarms* and the native technology is called Docker Swarm. Docker Swarm is easy-to-use and many companies are using it in real-world production. It's a lot simpler to install and manage than Kubernetes but lacks a lot of the advanced features and ecosystem of Kubernetes.

The Open Container Initiative (OCI)

Earlier in the chapter we mentioned the Open Containers Initiative — OCI².

The OCI is a governance council responsible for standardizing the low-level fundamental components of container infrastructure. In particular it focusses on *image format* and *container runtime* (don't worry if you're not comfortable with these terms yet, we'll cover them in the book).

It's also true that no discussion of the OCI is complete without mentioning a bit of history. And as with all accounts of history, the version you get depends on who's doing the talking. So, this is container history according to Nigel :-D

From day one, use of Docker grew like crazy. More and more people used it in more and more ways for more and more things. So, it was inevitable that some parties would get frustrated. This is normal and healthy.

The TLDR of this *history according to Nigel* is that a company called CoreOS (acquired by Red Hat which was then acquired by IBM) didn't like the way Docker did certain things. So, they created an open standard called **appc**³ that defined things like image format and container runtime. They also created an implementation of the spec called **rkt** (pronounced "rocket").

This put the container ecosystem in an awkward position with two competing standards.

Getting back to the story, this threatened to fracture the ecosystem and present users and customers with a dilemma. While competition is usually a good thing, *competing standards* is usually not. They cause confusion and slowdown user adoption. Not good for anybody.

With this in mind, everybody did their best to act like adults and came together to form the OCI — a lightweight agile council to govern container standards.

At the time of writing, the OCI has published two specifications (standards) -

²<https://www.opencontainers.org>

³<https://github.com/appc/spec/>

- The image-spec⁴
- The runtime-spec⁵
- The distribution-spec⁶

An analogy that's often used when referring to these two standards is *rail tracks*. These two standards are like agreeing on standard sizes and properties of rail tracks, leaving everyone else free to build better trains, better carriages, better signalling systems, better stations... all safe in the knowledge that they'll work on the standardized tracks. Nobody wants two competing standards for rail track sizes!

It's fair to say that the OCI specifications have had a major impact on the architecture and design of the core Docker product. All modern versions of Docker and Docker Hub implement the OCI specifications.

The OCI is organized under the auspices of the Linux Foundation.

Chapter summary

In this chapter, we learned about Docker, Inc. the company, and the Docker technology.

Docker, Inc. is a technology company out of San Francisco with an ambition to change the way we do software. They were arguably the *first-movers* and instigators of the modern container revolution.

The Docker technology focuses on running and managing application containers. It runs on Linux and Windows, can be installed almost anywhere, and is currently the most popular container runtime used by Kubernetes.

The Open Container Initiative (OCI) was instrumental in standardizing low-level container technologies such as runtimes, image format, and registries.

⁴<https://github.com/opencontainers/image-spec>

⁵<https://github.com/opencontainers/runtime-spec>

⁶<https://github.com/opencontainers/distribution-spec>

3: Installing Docker

There are lots of ways and places to install Docker. There's Windows, Mac, and Linux. You can install in the cloud, on premises, and on your laptop. And there are manual installs, scripted installs, wizard-based installs...

But don't let that scare you. They're all really easy, and a simple search for "how to install docker on <insert your choice here>" will reveal up-to-date instructions that are easy to follow. As a result, we won't waste too much space here. We'll cover the following.

- Docker Desktop
 - Windows
 - MacOS
- Multipass
- Server installs on
 - Linux
- Play with Docker

Docker Desktop

Docker Desktop is a desktop app from Docker, Inc. that makes it super-easy to work with containers. It includes the Docker engine, a slick UI, and an extension system with a marketplace. These extensions add some very useful features to Docker Desktop such as scanning images for vulnerabilities and making it easy to manage images and disk space.

Docker Desktop is free for educational purposes, but you'll have to pay if you start using it for work and your company has over 250 employees or does more than \$10M in annual revenue.

It runs on 64-bit versions of Windows 10, Windows 11, MacOS, and Linux.

Once installed, you have a fully working Docker environment that's great for development, testing, and learning. It includes Docker Compose and you can even enable a single-node Kubernetes cluster if you need to learn Kubernetes.

Docker Desktop on Windows can run native Windows containers as well as Linux containers. Docker Desktop on Mac and Linux can only run Linux containers.

We'll walk through the process of installing on Windows and MacOS.

Windows pre-reqs

Docker Desktop on Windows requires all of the following:

- 64-bit version of Windows 10/11
- Hardware virtualization support must be enabled in your system's BIOS
- WSL 2

Be very careful changing anything in your system's BIOS.

Installing Docker Desktop on Windows 10 and 11

Search the internet or ask your AI assistant how to “install Docker Desktop on Windows”. This will take you to the relevant download page where you can download the installer and follow the instructions. You may need to install and enable the WSL 2 backend (Windows Subsystem for Linux).

Once the installation is complete you may have to manually start Docker Desktop from the Windows Start menu. It may take a minute to start but you can watch the start progress via the animated whale icon on the Windows task bar at the bottom of the screen.

Once it's up and running you can open a terminal and type some simple docker commands.

```
$ docker version
Client:
 Cloud integration: v1.0.31
 Version:          20.10.23
 API version:      1.41
 Go version:       go1.18.10
 Git commit:       7155243
 Built:            Thu Jan 19 01:20:44 2023
 OS/Arch:          linux/amd64
 Context:          default
 Experimental:     true
Server:
 Engine:
  Version:          20.10.23
<Snip>
 OS/Arch:          linux/amd64
 Experimental:     true
```

Notice the output is showing OS/Arch: linux/amd64 for the **Server** component. This is because a default installation assumes you'll be working with Linux containers.

You can easily switch to *Windows containers* by right-clicking the Docker whale icon in the Windows notifications tray and selecting `Switch to Windows containers....`

Be aware that any existing Linux containers will keep running in the background but you won't be able to see or manage them until you switch back to Linux containers mode.

Run another `docker version` command and look for the `windows/amd64` line in the `Server` section of the output.

```
C:\> docker version
Client:
  <Snip>

Server:
  Engine:
    <Snip>
    OS/Arch:      windows/amd64
    Experimental: true
```

You can now run and manage Windows containers (containers running Windows applications).

Congratulations. You now have a working installation of Docker on your Windows machine.

Installing Docker Desktop on Mac

Docker Desktop for Mac is like Docker Desktop on Windows — a packaged product with a slick UI that gets you a single-engine installation of Docker that's ideal for local development needs. You can also enable a single-node Kubernetes cluster.

Before proceeding with the installation, it's worth noting that Docker Desktop on Mac installs all of the Docker engine components in a lightweight Linux VM that seamlessly exposes the API to your local Mac environment. This means you can open a terminal on your Mac and use the regular Docker commands without ever knowing it's all running in a hidden VM. This is why Docker Desktop on Mac only work with Linux containers — it's all running inside a Linux VM. This is fine as Linux is where most of the container action is.

Figure 3.1 shows the high-level architecture for Docker Desktop on Mac.

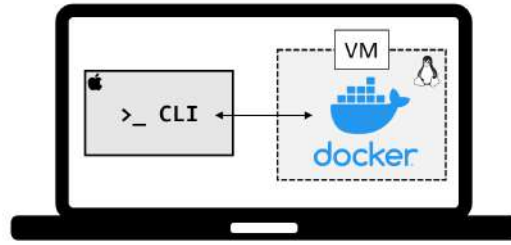


Figure 3.1

The simplest way to install Docker Desktop on your Mac is to search the web or ask your AI how to “install Docker Desktop on MacOS”. Follow the links to the download and then complete the simple installer.

Once the installation is complete you may have to manually start Docker Desktop from the MacOS Launchpad. It may take a minute to start but you can watch the animated Docker whale icon in the status bar at the top of your screen. Once it’s started you can click the whale icon to manage Docker Desktop.

Open a terminal window and run some regular Docker commands. Try the following.

```
$ docker version

Client:
 Cloud integration: v1.0.31
 Version:          23.0.5
 API version:      1.42
 <Snip>
 OS/Arch:          darwin/arm64
 Context:          desktop-linux

Server: Docker Desktop 4.19.0 (106363)
 Engine:
  Version:          dev
  API version:      1.43 (minimum version 1.12)
  <Snip>
  OS/Arch:          linux/arm64
  Experimental:     false
 containerd:
  Version:          1.6.20
  GitCommit:        2806fc1057397dbaefbea0e4e17bddfbd388f38
 runc:
  Version:          1.1.5
  GitCommit:        v1.1.5-0-gf19387a
 <Snip>
```

Notice that the OS/Arch: for the **Server** component is showing as linux/amd64 or linux/arm64. This is because the daemon is running inside the Linux VM mentioned

earlier. The **Client** component is a native Mac application and runs directly on the Mac OS Darwin kernel which is why it shows as either `darwin/amd64` or `darwin/arm64`.

You can now use Docker on your Mac.

Installing Docker with Multipass

Multipass is a free tool for creating cloud-style Linux VMs on your Linux, Mac, or Windows machine. It's my go-to choice for Docker testing on my laptop as it's incredibly easy to spin-up and tear-down Docker VMs.

Just go to <https://multipass.run/install> and install the right edition for your hardware and OS.

Once installed you'll only need the following three commands:

```
$ multipass launch
$ multipass ls
$ multipass shell
```

Let's see how to launch and connect to a new VM that will have Docker pre-installed.

Run the following command to create a new VM called **node1** based on the **docker** image. The docker image has Docker pre-installed and ready to go.

```
$ multipass launch docker --name node1
```

It'll take a minute or two to download the image and launch the VM.

List VMs to make sure it launched properly.

```
$ multipass ls
```

Name	State	IPv4	Image
node1	Running	192.168.64.37	Ubuntu 22.04 LTS
		172.17.0.1	
		172.18.0.1	

You'll use the 192 IP address when working with the examples later in the book.

Connect to the VM with the following command.

```
$ multipass shell node1
```

You're now logged on to the VM and can run regular Docker commands.

Just type `exit` to log out of the VM. Use `multipass delete node1` and then `multipass purge` to delete it.

Installing Docker on Linux

There are lots of ways to install Docker on Linux and most of them are easy. The recommended way is to search the web or ask your AI how to do it. The instructions in this section may be out of date and just for guidance purposes.

In this section we'll look at one of the ways to install Docker on Ubuntu Linux 22.04 LTS. The procedure assumes you've already installed Linux and are logged on.

1. Remove existing Docker packages.

```
$ sudo apt-get remove docker docker-engine docker.io containerd runc
```

2. Update the apt package index.

```
$ sudo apt-get update
$ sudo apt-get install ca-certificates curl gnupg
...
```

3. Add the Docker GPG key.

```
$ sudo install -m 0755 -d /etc/apt/keyrings
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
    sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
$ sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

4. Set-up the repository.

```
$ echo \
    "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg] \
    https://download.docker.com/linux/ubuntu \
    "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
    sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

5. Install Docker from the official repo.

```
$ sudo apt-get update
$ sudo apt-get install \
    docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

Docker is now installed and you can test by running some commands.

```
$ sudo docker --version
Docker version 24.0.0, build 98fdcd7

$ sudo docker info
Server:
 Containers: 1
  Running: 1
  Paused: 0
  Stopped: 0
 Images: 1
 Server Version: 24.0.0
 Storage Driver: overlay2
 ...
```

Play with Docker

Play with Docker (PWD) is a fully functional internet-based Docker playground that lasts for 4 hours. You can add multiple nodes and even cluster them in a swarm.

Sometimes performance can be slow, but for a free service it's excellent!

Visit <https://labs.play-with-docker.com/>

Chapter Summary

You can run Docker almost anywhere and most of the installation methods are simple.

Docker Desktop provides you a fully-functional Docker environment on your Linux, Mac, or Windows machine. It's easy to install, includes the Docker engine, has a slick UI, and has a marketplace with lots of extensions to extend its capabilities. It's a great choice for a local Docker development environment and even lets you spin-up a single-node Kubernetes cluster.

Packages exist to install the Docker engine on most Linux distros.

Play with Docker is a free 4-hour Docker playground on the internet.

4: The big picture

The aim of this chapter is to paint a quick big-picture of what Docker is all about before we dive in deeper in later chapters.

We'll break this chapter into two:

- The Ops perspective
- The Dev perspective

In the Ops Perspective section, we'll download an image, start a new container, log in to the new container, run a command inside of it, and then destroy it.

In the Dev Perspective section, we'll focus more on the app. We'll clone some app code from GitHub, inspect a Dockerfile, containerize the app, run it as a container.

These two sections will give you a good idea of what Docker is all about and how the major components fit together. **It's recommended that you read both sections to get the *dev* and the *ops* perspectives.** DevOps anyone?

Don't worry if some of the stuff we do here is totally new to you. We're not trying to make you an expert in this chapter. This is about giving you a *feel of things* — setting you up so that when we get into the details in later chapters, you have an idea of how the pieces fit together.

If you want to follow along, all you need is a single Docker host with an internet connection. I recommend Docker Desktop for your Mac or Windows PC. However, the examples will work anywhere that you have Docker installed. We'll be showing examples using Linux containers and Windows containers.

If you can't install software and don't have access to a public cloud, another great way to get Docker is Play With Docker (PWD). This is a web-based Docker playground that you can use for free. Just point your web browser to <https://labs.play-with-docker.com/> and you're ready to go (you'll need a Docker Hub or GitHub account to be able to login).

As we progress through the chapter, we may use the terms “Docker host” and “Docker node” interchangeably. Both refer to the system that you are running Docker on.

The Ops Perspective

When you install Docker, you get two major components:

- The Docker client
- The Docker engine (sometimes called the “Docker daemon”)

The engine implements the runtime, API and everything else required to run containers.

In a default Linux installation, the client talks to the daemon via a local IPC/Unix socket at `/var/run/docker.sock`. On Windows this happens via a named pipe at `npipes:///./pipe/docker_engine`. Once installed, you can use the `docker version` command to test that the client and daemon (server) are running and talking to each other.

```
> docker version
Client: Docker Engine - Community
 Version:           24.0.0
 API version:       1.43
 Go version:        go1.20.4
 Git commit:        98fcd7
 Built:             Mon May 15 18:48:45 2023
 OS/Arch:           linux/arm64
 Context:           default

Server: Docker Engine - Community
Engine:
 Version:           24.0.0
 API version:       1.43 (minimum version 1.12)
 Go version:        go1.20.4
 Git commit:        1331b8c
 Built:             Mon May 15 18:48:45 2023
 OS/Arch:           linux/arm64
 Experimental:      false
<Snip>
```

If you get a response back from the `Client` **and** `Server`, you’re good to go.

If you are using Linux and get an error response from the `Server` component, make sure that Docker is up and running. Also, try the command again with `sudo` in front of it – `sudo docker version`. If it works with `sudo` you will need to add your user account to the local `docker` group, or prefix **all** `docker` commands with `sudo`.

Images

It’s useful to think of a Docker image as an object that contains an OS filesystem, an application, and all application dependencies. If you work in operations, it’s like a virtual machine template. A virtual machine template is essentially a stopped virtual machine. In the Docker world, an image is effectively a stopped container. If you’re a developer, you can think of an image as a *class*.

Run the `docker images` command on your Docker host.

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

If you are working from a freshly installed Docker host, or Play With Docker, you'll have no images and it will look like the previous output.

Getting images onto your Docker host is called *pulling*. Pull the `ubuntu:latest` image.

```
$ docker pull ubuntu:latest
latest: Pulling from library/ubuntu
dfd64a3b4296: Download complete
6f8fe7bffa0be: Download complete
3f5ef9003cef: Download complete
79d0ea7dc1a8: Download complete
docker.io/library/ubuntu:latest
```

Run the `docker images` command again to see the image you just pulled.

```
$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
ubuntu        latest    dfd64a3b4296   1 minute ago   106MB
```

We'll get into the details of where the image is stored and what's inside of it in later chapters. For now, it's enough to know that an image contains enough of an operating system (OS), as well as all the code and dependencies to run whatever application it's designed for. The `ubuntu` image that we've pulled has a stripped-down version of the Ubuntu Linux filesystem and a few of the common Ubuntu utilities.

If you pull an application container, such as `nginx:latest`, you'll get an image with a minimal OS **as well as** the code to run the app (NGINX).

It's also worth noting that each image gets its own unique ID. When referencing images, you can refer to them using either IDs or names. If you're working with image ID's, it's usually enough to type the first few characters of the ID — as long as it's unique, Docker will know which image you mean.

Containers

Now that we have an image pulled locally we can use the `docker run` command to launch a container from it.

```
$ docker run -it ubuntu:latest /bin/bash
root@6dc20d508db0:/#
```

Look closely at the output from the previous command. You'll see that the shell prompt has changed. This is because the `-it` flags switch your shell into the terminal of the container — your shell is now inside of the new container!

Let's examine that `docker run` command.

`docker run` tells Docker to start a new container. The `-it` flags tell Docker to make the container interactive and to attach the current shell to the container's terminal (we'll get more specific about this in the chapter on containers). Next, the command tells Docker that we want the container to be based on the `ubuntu:latest` image. Finally, it tells Docker which process we want to run inside of the container — a Bash shell.

Run a `ps` command from inside of the container to list all running processes.

```
root@6dc20d508db0:/# ps -elf
F S UID      PID  PPID  NI ADDR SZ WCHAN  STIME TTY      TIME CMD
4 S root       1     0    0 -  4560 -          13:38 pts/0    00:00:00 /bin/bash
0 R root       9     1    0 -  8606 -          13:38 pts/0    00:00:00 ps -elf
```

There are only two processes:

- PID 1. This is the `/bin/bash` process that we told the container to run with the `docker run` command.
- PID 9. This is the `ps -elf` command/process that we ran to list the running processes.

The presence of the `ps -elf` process in the Linux output can be a bit misleading as it is a short-lived process that dies as soon as the `ps` command completes. This means the only long-running process inside of the container is the `/bin/bash` process.

Press `Ctrl-PQ` to exit the container without terminating it. This will land your shell back at the terminal of your Docker host. You can verify this by looking at your shell prompt.

Now that you are back at the shell prompt of your Docker host, run the `ps` command again.

Notice how many more processes are running on your Docker host compared to the container you just ran.

Pressing `Ctrl-PQ` from inside a container will exit you from the container without killing it. You can see all running containers on your system using the `docker ps` command.

```
$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED   STATUS    NAMES
6dc20d508db0  ubuntu:latest  "/bin/bash"             7 mins   Up 7 min   vigilant_borg
```

The output shows a single running container. This is the one you created earlier and proves it's still running. You can also see it was created 7 minutes ago and has been running for 7 minutes.

Attaching to running containers

You can attach your shell to the terminal of a running container with the `docker exec` command. As the container from the previous steps is still running, let's make a new connection to it.

This example references a container called “vigilant_borg”. The name of your container will be different, so remember to substitute “vigilant_borg” with the name or ID of the container running on your Docker host.

```
$ docker exec -it vigilant_borg bash
root@6dc20d508db0:/#
```

Notice that your shell prompt has changed again. You are logged into the container again.

The format of the `docker exec` command is: `docker exec <options> <container-name or container-id> <command/app>`. We used the `-it` flags to attach our shell to the container's shell. We referenced the container by name and told it to run the bash shell. We could easily have referenced the container by its hex ID.

Exit the container again by pressing `Ctrl-PQ`.

Your shell prompt should be back to your Docker host.

Run the `docker ps` command again to verify that your container is still running.

```
$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED   STATUS    NAMES
6dc20d508db0  ubuntu:latest  "/bin/bash"             9 mins   Up 9 min   vigilant_borg
```

Stop the container and kill it using the `docker stop` and `docker rm` commands. Remember to substitute the names/IDs of your own containers.


```
$ docker stop vigilant_borg
vigilant_borg
```

It may take a few seconds for the container to gracefully stop.

```
$ docker rm vigilant_borg
vigilant_borg
```

Verify that the container was successfully deleted by running the `docker ps` command with the `-a` flag. Adding `-a` tells Docker to list all containers, even those in the stopped state.

```
$ docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS        NAMES
```

Congratulations, you've just pulled a Docker image, started a container from it, attached to it, executed a command inside it, stopped it, and deleted it.

The Dev Perspective

Containers are all about the apps.

In this section, we'll clone an app from a Git repo, inspect its Dockerfile, containerize it, and run it as a container.

The Linux app can be cloned from: <https://github.com/nigelpoulton/psweb.git>

Run all of the following commands from a terminal on your Docker host.

Clone the repo locally. This will pull the application code to your local Docker host ready for you to containerize it.

```
$ git clone https://github.com/nigelpoulton/psweb.git
Cloning into 'psweb'...
remote: Enumerating objects: 63, done.
remote: Counting objects: 100% (34/34), done.
remote: Compressing objects: 100% (22/22), done.
remote: Total 63 (delta 13), reused 25 (delta 9), pack-reused 29
Receiving objects: 100% (63/63), 13.29 KiB | 4.43 MiB/s, done.
Resolving deltas: 100% (21/21), done.
```

Change directory into the cloned repo's directory and list its contents.

```

$ cd psweb
$ ls -l
total 40
-rw-r--r--@ 1 ubuntu ubuntu 338 24 Apr 19:29 Dockerfile
-rw-r--r--@ 1 ubuntu ubuntu 396 24 Apr 19:32 README.md
-rw-r--r--@ 1 ubuntu ubuntu 341 24 Apr 19:29 app.js
-rw-r--r-- 1 ubuntu ubuntu 216 24 Apr 19:29 circle.yml
-rw-r--r--@ 1 ubuntu ubuntu 377 24 Apr 19:36 package.json
drwxr-xr-x 4 ubuntu ubuntu 128 24 Apr 19:29 test
drwxr-xr-x 3 ubuntu ubuntu 96 24 Apr 19:29 views

```

The app is a simple nodejs web app running some static HTML.

The *Dockerfile* is a plain-text document that tells Docker how to build the app and dependencies into a Docker image.

List the contents of the Dockerfile.

```

$ cat Dockerfile

FROM alpine
LABEL maintainer="nigelpoulton@hotmail.com"
RUN apk add --update nodejs nodejs-npm
COPY . /src
WORKDIR /src
RUN npm install
EXPOSE 8080
ENTRYPOINT ["node", "./app.js"]

```

For now, it's enough to know that each line represents an instruction that Docker uses to build the app into an image.

At this point we've pulled some application code from a remote Git repo and we've looked at the application's Dockerfile that contains the instructions Docker uses to build it as an image.

Use the `docker build` command to create a new image using the instructions in the Dockerfile. This example creates a new Docker image called `test:latest`.

Be sure to run the command from within the directory containing the app code and Dockerfile.

```
$ docker build -t test:latest .
[+] Building 36.2s (11/11) FINISHED
=> [internal] load .dockerignore                                0.0s
=> => transferring context: 2B                                  0.0s
=> [internal] load build definition from Dockerfile             0.0s
<Snip>
=> => naming to docker.io/library/test:latest                 0.0s
=> => unpacking to docker.io/library/test:latest               0.7s
```

Once the build is complete, check to make sure that the new `test:latest` image exists on your host.

```
$ docker images
REPO      TAG      IMAGE ID      CREATED      SIZE
test      latest   1ede254e072b   7 seconds ago 154MB
```

You have a newly-built Docker image with the app and dependencies inside.

Run a container from the image and test the app.

```
$ docker run -d \
  --name web1 \
  --publish 8080:8080 \
  test:latest
```

Open a web browser and navigate to the DNS name or IP address of the Docker host that you are running the container from, and point it to port 8080. You will see the following web page.

If you're following along on Docker Desktop, you'll be able to connect to `localhost:8080` or `127.0.0.1:8080`. If you're following along on Play With Docker, you will be able to click the `8080` hyperlink above the terminal screen.

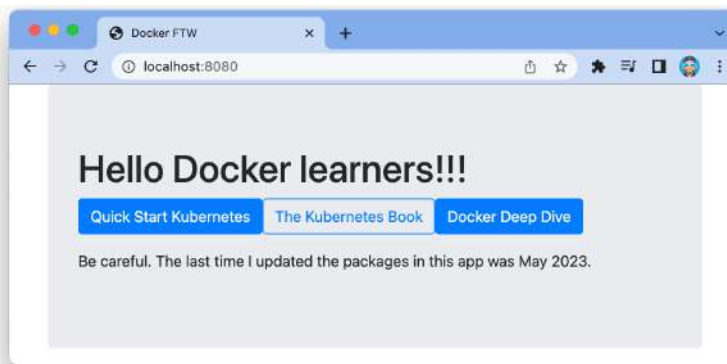


Figure 4.1

Well done. You've copied some application code from a remote Git repo, built it into a Docker image, and ran it as a container. We call this "containerizing an app".

Chapter Summary

In the Ops section of the chapter, you downloaded a Docker image, launched a container from it, logged into the container, executed a command inside of it, and then stopped and deleted the container.

In the Dev section you containerized a simple application by pulling some source code from GitHub and building it into an image using instructions in a Dockerfile. You then ran the containerized app.

This *big picture* view should help you with the up-coming chapters where we'll dig deeper into images and containers.

Part 2: The technical stuff

5: The Docker Engine

In this chapter, we'll take a look under the hood of the Docker Engine.

You can use Docker without understanding any of the things we'll cover in this chapter, so feel free to skip it. However, to be a real master of anything, you need to understand what's going on under the hood. So, to be a *real* Docker master, you need to know the stuff in this chapter.

This will be a theory-based chapter with no hands-on exercises.

As this chapter is part of the **technical section** of the book, we're going to employ the three-tiered approach where we split the chapter into three sections:

- **The TLDR:** Two or three quick paragraphs that you can read while standing in line for a coffee
- **The deep dive:** The really long bit where we get into the detail
- **The commands:** A quick recap of the commands we learned

Let's go and learn about the Docker Engine!

Docker Engine - The TLDR

The *Docker engine* is the core software that runs and manages containers. We often refer to it simply as *Docker*. If you know a thing or two about VMware, it might be useful to think of the Docker engine as being like ESXi.

The Docker engine is modular in design and built from lots of small specialised components. Most of these are pulled from the Moby project (<https://mobyproject.org/>) and implement open standards such as those maintained by the Open Container Initiative (OCI).

In many ways, the Docker Engine is like a car engine — both are modular and created by connecting many small specialized parts:

- A car engine is made from many specialized parts that work together to make a car drive — intake manifolds, throttle body, cylinders, pistons, spark plugs, exhaust manifolds etc.

- The Docker Engine is made from many specialized tools that work together to create and run containers — The API, image builder, high-level runtime, low-level runtime, shims etc.

At the time of writing, the major components that make up the Docker engine are the Docker daemon, the build system, containerd, runc, and various plugins such as networking and volumes. Together, these create and run containers.

Figure 5.1 shows a high-level view.

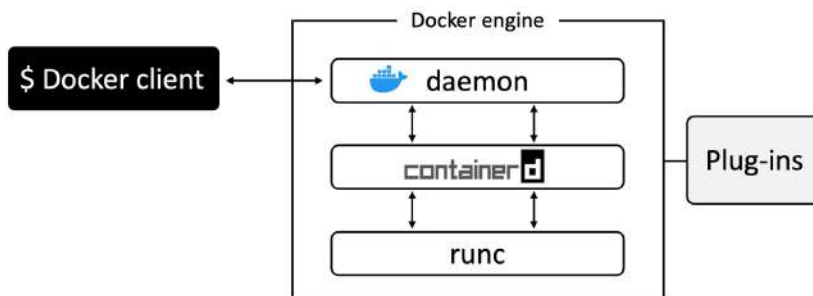


Figure 5.1

Throughout the book we'll refer to runc and containerd with lower-case "r" and "c". This means sentences starting with either runc or containerd will not start with a capital letter. This is intentional and not a mistake.

Docker Engine - The Deep Dive

When Docker was first released, the Docker engine had two major components:

- The Docker daemon (sometimes referred to as just "the daemon")
- LXC

The Docker daemon was a monolithic binary – it contained all the code for API, the runtime, image builds, and more.

LXC provided the daemon with access to the fundamental building-blocks of containers that existed in the Linux kernel. Things like *namespaces* and *control groups (cgroups)*.

Figure 5.2. shows how the daemon, LXC, and the OS, interacted in **older** versions of Docker.

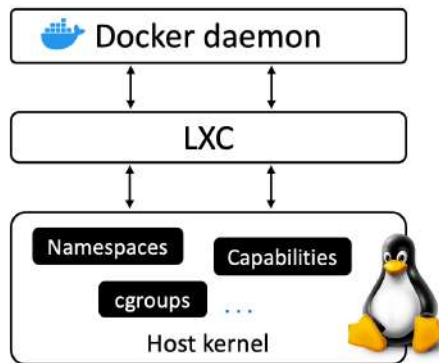


Figure 5.2 Old Docker architecture

Getting rid of LXC

The reliance on LXC was an issue from the start.

First up, LXC is Linux-specific. This was a problem for a project that had aspirations of being multi-platform.

Second up, being reliant on an external tool for something so core to the project was a huge risk.

As a result, Docker, Inc. developed their own tool called *libcontainer* as a replacement for LXC. The goal of *libcontainer* was to be a platform-agnostic tool that provided Docker with access to the fundamental container building-blocks that exist in the host kernel.

Libcontainer replaced LXC as the default *execution driver* a very long time ago in Docker 0.9.

Getting rid of the monolithic Docker daemon

Over time, the monolithic nature of the Docker daemon became more and more problematic:

1. It's hard to innovate on
2. It got slower
3. It wasn't what the ecosystem wanted

Docker, Inc. was aware of these challenges and began a huge effort to break apart the monolithic daemon and modularize it. The aim of this work was to break out as much of the functionality as possible from the daemon and re-implement it in smaller

specialized tools. These specialized tools can be swapped out, as well as easily re-used by third parties to build other tools. This plan followed the tried-and-tested Unix philosophy of building small specialized tools that can be pieced together into larger tools.

This work of breaking apart and re-factoring the Docker engine has seen **all of the container execution and container runtime code entirely removed from the daemon and refactored into small, specialized tools.**

Figure 5.3 shows a high-level view of the current Docker engine architecture with brief descriptions.

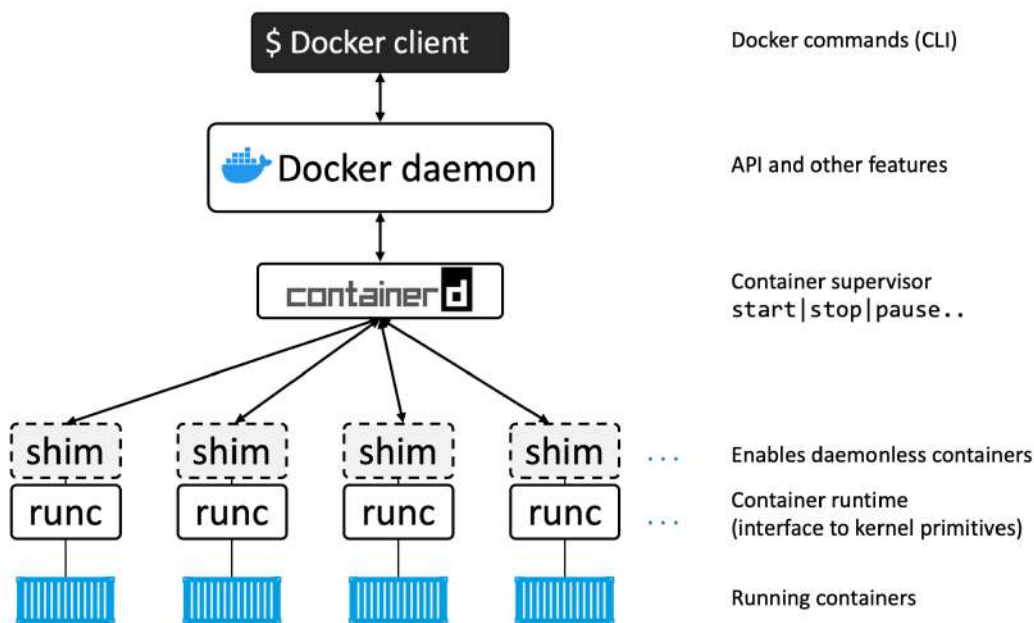


Figure 5.3

The influence of the Open Container Initiative (OCI)

While Docker, Inc. was breaking the daemon apart and refactoring code, the OCI⁷ was in the process of defining container-related standards:

1. Image spec⁸

⁷<https://www.opencontainers.org/>

⁸<https://github.com/opencontainers/image-spec>

2. Container runtime spec⁹

Both specifications were released as version 1.0 in July 2017 and we shouldn't see too much change, as stability is the name of the game here. At the time of writing a third spec has been added to standardise image distribution via registries.

Docker, Inc. was heavily involved in creating these specifications and contributed a lot of code.

All Docker versions since 2016 implement the OCI specifications. For example, the Docker daemon no longer contains any container runtime code — all container runtime code is implemented in a separate OCI-compliant layer. By default, Docker uses *runc* for this. *runc* is the *reference implementation* of the OCI container-runtime-spec. This is the *runc* container runtime layer in Figure 5.3.

As well as this, the *containerd* component of the Docker Engine makes sure Docker images are presented to *runc* as valid OCI bundles.

runc

As previously mentioned, *runc* is the OCI container-runtime-spec reference implementation. Docker, Inc. was heavily involved in defining the spec and developing *runc*.

If you strip everything else away, *runc* is a small, lightweight CLI wrapper for *libcontainer* – remember that *libcontainer* originally replaced *LXC* as the interface layer with the host OS in the early Docker architecture.

runc has a single purpose in life — create containers. And it's fast. But as it's a CLI wrapper, it's effectively a standalone container runtime tool. This means you can download and build the binary, and you'll have everything you need to build and play with *runc* (OCI) containers. But it's bare bones and very low-level, meaning you'll have none of the richness that you get with the full Docker engine.

We sometimes say that *runc* operates at “the OCI layer”. See Figure 5.3.

You can see *runc* release information at:

- <https://github.com/opencontainers/runc/releases>

containerd

As part of the effort to strip functionality out of the Docker daemon, all of the container execution logic was ripped out and refactored into a new tool called *containerd*

⁹<https://github.com/opencontainers/runtime-spec>

(pronounced container-dee). Its sole purpose in life is to manage container lifecycle operations such as `start` | `stop` | `pause` | `rm`...

containerd is available as a daemon for Linux and Windows, and Docker has been using it on Linux since the 1.11 release. In the Docker engine stack, containerd sits between the daemon and runc at the OCI layer.

As previously stated, containerd was originally intended to be small, lightweight, and designed for a single task in life — container lifecycle operations. However, over time it has branched out and taken on more functionality. Things like image pulls, volumes and networks.

One of the reasons for adding more functionality is to make it easier to use in other projects. For example, in Kubernetes it's beneficial for containerd to do things like push and pull images. However, all the extra functionality is modular and optional, meaning you can pick and choose which bits you want. So, it's possible to include containerd in projects such as Kubernetes, but only to take the pieces your project needs.

containerd was originally developed by Docker, Inc. and donated to the Cloud Native Computing Foundation (CNCF). At the time of writing, containerd is a fully graduated CNCF project, meaning it's stable and considered ready for production. You can see the latest releases here:

- <https://github.com/containerd/containerd/releases>

Starting a new container (example)

Now that we have a view of the big picture and some of the history, let's walk through the process of creating a new container.

The most common way of starting containers is using the Docker CLI. The following `docker run` command will start a simple new container based on the `alpine:latest` image.

```
$ docker run --name ctrl1 -it alpine:latest sh
```

When you type commands like this into the Docker CLI, the Docker client converts them into the appropriate API payload and POSTs them to the API endpoint exposed by the Docker daemon.

The API is implemented in the daemon and can be exposed over a local socket or the network. On Linux the socket is `/var/run/docker.sock` and on Windows it's `\pipe\docker_engine`.

Once the daemon receives the command to create a new container, it makes a call to containerd. Remember that the daemon no-longer contains any code to create containers!

The daemon communicates with `containerd` via a CRUD-style API over gRPC¹⁰.

Despite its name, *containerd* cannot actually create containers. It uses *runc* to do that. It converts the required Docker image into an OCI bundle and tells *runc* to use this to create a new container.

runc interfaces with the OS kernel to pull together all of the constructs necessary to create a container (namespaces, cgroups etc.). The container process is started as a child-process of *runc*, and as soon as it starts, *runc* will exit.

Voilà! The container is now started.

The process is summarized in Figure 5.4.

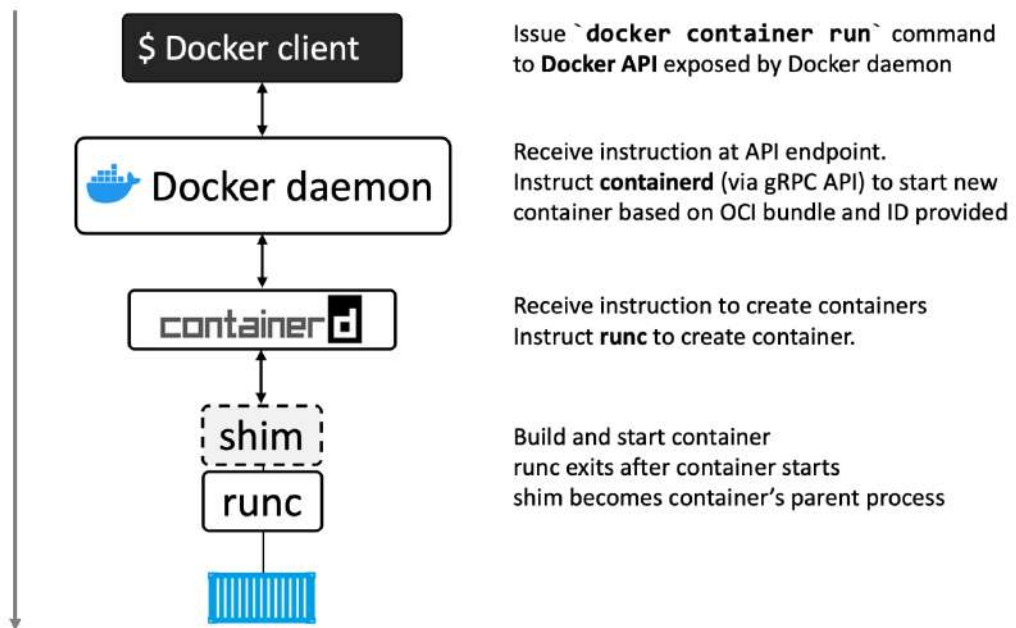


Figure 5.4

One huge benefit of this model

Having all of the logic and code to start and manage containers removed from the daemon means that the entire container runtime is decoupled from the Docker daemon. We sometimes call this “daemonless containers” and it makes it possible to perform

¹⁰<https://grpc.io/>

maintenance and upgrades on the Docker daemon without impacting running containers.

In the old model, where all of container runtime logic was implemented in the daemon, starting and stopping the daemon would kill all running containers on the host. This was a huge problem in production environments.

Fortunately, this is no longer a problem.

What's this shim all about?

Some of the diagrams in the chapter have shown a shim component.

The shim is integral to the implementation of daemonless containers — what we just mentioned about decoupling running containers from the daemon for things like daemon upgrades.

We mentioned earlier that *containerd* uses *runc* to create new containers. In fact, it forks a new instance of *runc* for every container it creates. However, once each container is created, the *runc* process exits. This means we can run hundreds of containers without having to run hundreds of *runc* instances.

Once a container's parent *runc* process exits, the associated *containerd*-shim process becomes the container's parent. Some of the responsibilities the shim performs as a container's parent include:

- Keeping any STDIN and STDOUT streams open so that when the daemon is restarted, the container doesn't terminate due to pipes being closed etc.
- Reports the container's exit status back to the daemon.

How it's implemented on Linux

On a Linux system, the components we've discussed are implemented as separate binaries as follows:

- `/usr/bin/dockerd` (the Docker daemon)
- `/usr/bin/containerd`
- `/usr/bin/containerd-shim-runc-v2`
- `/usr/bin/runc`

You can see all of these on a Linux system by running a `ps` command on the Docker host. Obviously, some of them will only be present when the system has running containers.

What's the point of the daemon

With all of the execution and runtime code stripped out of the daemon you might be asking the question: “what is left in the daemon?”.

Obviously, the answer to this question will change over time as more and more functionality is stripped out and modularized. However, the daemon is capable of pushing and pulling images, implementing the Docker API, authentication, security, and more.

At the time of writing, image management is in the process of being removed from the daemon and handled by containerd.

Chapter summary

The Docker engine is software that makes it easy to build, ship, and run containers. It implements the OCI standards and is a modular app comprising lots of small, specialised components.

The *Docker daemon* component implements the Docker API and can do things such as image management, networks, and volumes. However, image management is currently being removed from the daemon and implemented in containerd.

The **containerd** component oversees container execution and image management tasks. It was originally written by Docker, Inc. but then contributed to the CNCF. It's usually classified as a high-level runtime that acts as a container supervisor managing lifecycle operations. It is small and lightweight and is used by many other projects including Kubernetes.

containerd relies on a low-level runtime called **runc** to interface with the host kernel and build containers. runc is the reference implementation of the OCI runtime-spec and expects to start containers from OCI-compliant bundles. containerd talks to runc and ensures Docker images are presented to runc as OCI-compliant bundles.

runc can be used as a standalone CLI tool to create containers. It's based on code from libcontainer and is used almost everywhere that containerd is used.

6: Images

In this chapter we'll dive deep into Docker images. The aim is to give you a **solid understanding** of what Docker images are, how to perform basic operations, and how they work under-the-hood.

We'll see how to build new images for our own applications in a later chapter.

We'll split this chapter into the usual three parts:

- The TLDR
- The deep dive
- The commands

Docker images - The TLDR

Image, *Docker image*, *container image*, and *OCI image* all mean the same thing. We'll use the terms interchangeably.

A container image is read-only package that contains everything you need to run an application. It includes application code, application dependencies, a minimal set of OS constructs, and metadata. A single image can be used to start one or more containers.

If you're familiar with VMware, you can think of images as similar to VM templates. A VM template is like a stopped VM — a container image is like a stopped container. If you're a developer you can think of them as similar to *classes*. You can create one or more objects from a class — you can create one or more containers from an image.

You get container images by *pulling* them from a *registry*. The most common registry is Docker Hub¹¹ but others exist. The *pull* operation downloads an image to your local Docker host where Docker can use it to start one or more containers.

Images are made up of multiple *layers* that are stacked on top of each other and represented as a single object. Inside of the image is a cut-down operating system (OS) and all of the files and dependencies required to run an application. Because containers are intended to be fast and lightweight, images tend to be small (Windows images tend to be huge).

That's the elevator pitch. Let's dig a little deeper.

¹¹<https://hub.docker.com>

Docker images - The deep dive

We've mentioned a couple of times already that **images** are like stopped containers. In fact, you can stop a container and create a new image from it. With this in mind, images are considered *build-time* constructs, whereas containers are *run-time* constructs.

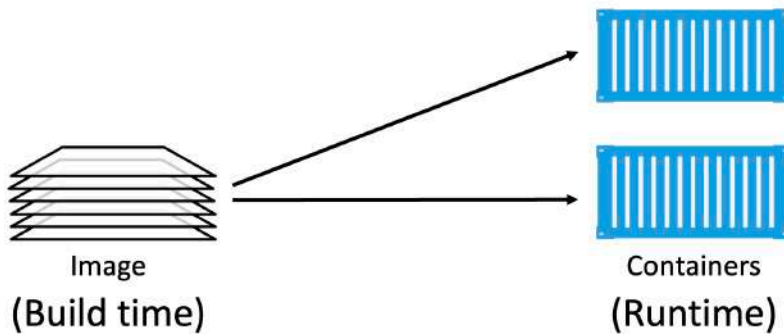


Figure 6.1

Images and containers

Figure 6.1 shows high-level view of the relationship between images and containers. We use the `docker run` and `docker service create` commands to start one or more containers from a single image. Once you've started a container from an image, the two constructs become dependent on each other, and you cannot delete the image until the last container using it has been stopped and destroyed.

Images are usually small

The whole purpose of a container is to run a single application or service. This means it only needs the code and dependencies of the app it's running — it doesn't need anything else. This means images are also small and stripped of all non-essential parts.

For example, at the time of writing the official Alpine Linux image is 7MB. This is because it doesn't ship with 6 different shells, three different package managers and more... In fact, a lot of images ship without a shell or a package manager — if the application doesn't need it, it doesn't get included in the image.

Images don't include a kernel. This is because containers share the kernel of the host they're running on. It's normal for the only OS components included in an image to be

a few important filesystem components and other basic constructs. This is why you'll sometimes hear people say "images contain just enough OS".

Windows-based images tend to be a lot bigger than Linux-based images because of the way the Windows OS works. It's not uncommon for Windows images to be several gigabytes and take a long time to push and pull.

Pulling images

A cleanly installed Docker host has no images in its local repository.

The local image repository on a Linux host is usually in `/var/lib/docker/<storage-driver>`. If you're using Docker on your Mac or PC with Docker Desktop, everything runs inside of a VM.

You can use the following command to check if your Docker host has any images in its local repository.

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

The process of getting images onto a Docker host is called *pulling*. So, if you want the latest Busybox image on your Docker host, you'd have to *pull* it. Use the following commands to *pull* some images and then check their sizes.

If you are following along on Linux and haven't added your user account to the local docker Unix group, you may need to add `sudo` to the beginning of all the following commands.

Linux example:

```
$ docker pull redis:latest
latest: Pulling from library/redis
b5d25b35c1db: Pull complete
6970efae6230: Pull complete
fea4afd29d1f: Pull complete
7977d153b5b9: Pull complete
7945d827bd72: Pull complete
b6aa3d1ce554: Pull complete
Digest: sha256:ea30bef6a1424d032295b90db20a869fc8db76331091543b7a80175cede7d887
Status: Downloaded newer image for redis:latest
docker.io/library/redis:latest

$ docker pull alpine:latest
latest: Pulling from library/alpine
```

```
08409d417260: Pull complete
Digest: sha256:02bb6f428431fbc2809c5d1b41eab5a68350194fb508869a33cb1af4444c9b11
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
```

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
alpine	latest	44dd6f223004	9 days ago	7.73MB
redis	latest	2334573cc576	2 weeks ago	111MB

Windows example:

```
> docker pull mcr.microsoft.com/powershell:latest
latest: Pulling from powershell
5b663e3b9104: Pull complete
9018627900ee: Pull complete
133ab280ee0f: Pull complete
084853899645: Pull complete
399a2a3857ed: Pull complete
6c1c6d29a559: Pull complete
d1495ba41b1c: Pull complete
190bd9d6eb96: Pull complete
7c239384fec8: Pull complete
21aee845547a: Pull complete
f951bda9026b: Pull complete
Digest: sha256:fb9555...123f3bd7
Status: Downloaded newer image for mcr.microsoft.com/powershell:latest
mcr.microsoft.com/powershell:latest
```

```
> docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mcr.microsoft.com/powershell	latest	73175ce91dff	2 days ago	495MB
mcr.microsoft.com/.../iis	latest	6e5c6561c432	3 days ago	5.05GB

As you can see, the images are now present in the Docker host's local repository. You can also see that the Windows images are a lot larger and comprise many more layers.

Image naming

When pulling an image, you have to specify the name of the image you're pulling. Let's take a minute to look at image naming. To do that we need a bit of background on how images are stored.

Image registries

We store images in centralised places called *registries*. Most modern registries implement the OCI distribution-spec and we sometimes call them *OCI registries*. The job of a

registry is to securely store container images and make them easy to access from different environments. Some registries offer advanced services such as image scanning and integration with build pipelines.

The most common registry is Docker Hub, but others exist including 3rd-party registries and secure on-premises registries. However, the Docker client is opinionated and defaults to using Docker Hub. We'll be using Docker Hub for the rest of the book.

The output of the following command is snipped, but you can see that Docker is configured to use `https://index.docker.io/v1/` as its default registry. This automatically redirects to `https://index.docker.io/v2/`.

```
$ docker info
<Snip>
Default Runtime: runc
containerd version: 2806fc1057397dbaeefbea0e4e17bddfbd388f38
runc version: v1.1.5-0-gf19387a
Registry: https://index.docker.io/v1/
<Snip>
```

Image registries contain one or more *image repositories*. In turn, image repositories contain one or more images. That might be a bit confusing, so Figure 6.2 shows a picture of an image registry with 3 repositories, and each repository has one or more images.

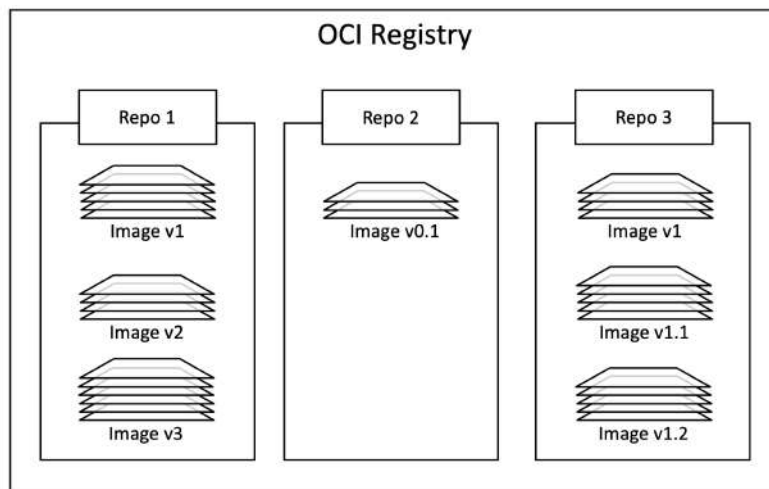


Figure 6.2

Official repositories

Docker Hub has the concept of *official repositories*.

As the name suggests, *official repositories* are the home to images that have been vetted and curated by the application vendor and Docker, Inc. This means they *should* contain up-to-date, high-quality code, that is secure, well-documented, and in-line with best practices.

If a repository isn't an *official repository* it can be like the wild-west — you should **not** assume they are safe, well-documented or built according to best practices. That's not saying the images they contain are bad. There's some excellent stuff in *unofficial repositories*. You just need to be very careful before trusting code from them. To be honest, you should never trust software from the internet — even images from *official repositories*.

Most of the popular applications and operating systems have *official repositories* on Docker Hub. They're easy to spot because they live at the top level of the Docker Hub namespace and have a green “Docker official image” badge. The following list shows a few *official repositories* and their URLs that exist at the top-level of the Docker Hub namespace:

- **nginx**: https://hub.docker.com/_/nginx/
- **busybox**: https://hub.docker.com/_/busybox/
- **redis**: https://hub.docker.com/_/redis/
- **mongo**: https://hub.docker.com/_/mongo/

On the other hand, my own personal images live in the wild west of *unofficial repositories* and should **not** be trusted. Here are some examples of images in my repositories:

- nigelpoulton/tu-demo — <https://hub.docker.com/r/nigelpoulton/tu-demo/>
- nigelpoulton/pluralsight-docker-ci — <https://hub.docker.com/r/nigelpoulton/pluralsight-docker-ci/>

Not only are images in my repositories **not** vetted, **not** kept up-to-date, **not** secure, and **not** well documented... they also don't live at the top-level of the Docker Hub namespace. My repositories all live within the `nigelpoulton` second-level namespace.

After all of that, we can finally look at how we address images on the Docker command line.

Image naming and tagging

Addressing images from official repositories is as simple as providing the repository name and tag separated by a colon (:). The format for `docker pull`, when working with an image from an official repository is:

```
$ docker pull <repository>:<tag>
```

In the Linux examples from earlier, we pulled an Alpine and a Redis image with the following two commands:

```
$ docker pull alpine:latest
$ docker pull redis:latest`
```

These pulled the images tagged as “latest” from the top-level “alpine” and “redis” repositories.

The following examples show how to pull various different images from *official repositories*:

```
$ docker pull mongo:4.2.24
//This will pull the image tagged as `4.2.24` from the official `mongo` repository.

$ docker pull busybox:glibc
//This will pull the image tagged as `glibc` from the official `busybox` repository.

$ docker pull alpine
//This will pull the image tagged as `latest` from the official `alpine` repository.
```

A couple of points about those commands.

First, if you **do not** specify an image tag after the repository name, Docker will assume you are referring to the image tagged as `latest`. If the repository doesn’t have an image tagged as `latest` the command will fail.

Second, the `latest` tag doesn’t have any magical powers. Just because an image is tagged as `latest` does not guarantee it is the most recent image in the repository!

Pulling images from an *unofficial repository* is essentially the same — you just need to prepend the repository name with a Docker Hub username or organization name. The following example shows how to pull the `v2` image from the `tu-demo` repository owned by a not-to-be-trusted person whose Docker Hub account name is `nigelpoulton`.

```
$ docker pull nigelpoulton/tu-demo:v2
//This will pull the image tagged as `v2`
//from the `tu-demo` repository within the `nigelpoulton` namespace
```

If you want to pull images from 3rd party registries (not Docker Hub) you just prepend the repository name with the DNS name of the registry. For example, the following command pulls the `3.1.5` image from the `google-containers/git-sync` repo on the Google Container Registry (`gcr.io`).


```
$ docker pull gcr.io/google-containers/git-sync:v3.1.5
v3.1.5: Pulling from google-containers/git-sync
597de8ba0c30: Pull complete
b263d8e943d1: Pull complete
a20ed723abc0: Pull complete
49535c7e3a51: Pull complete
4a20d0825f07: Pull complete
Digest: sha256:f38673f25b8...b5f8f63c4da7cc6
Status: Downloaded newer image for gcr.io/google-containers/git-sync:v3.1.5
gcr.io/google-containers/git-sync:v3.1.5
```

Notice how the pull experience is exactly the same from Docker Hub and other registries.

Images with multiple tags

One final word about image tags... A single image can have as many tags as you want. This is because tags are arbitrary alpha-numeric values that are stored as metadata alongside the image.

At first glance, the following output appears to show three images. However, on closer inspection it's actually two images – the image with the c610c6a38555 ID is tagged as `latest` as well as `v1`.

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nigelpoulton/tu-demo	latest	c610c6a38555	22 months ago	58.1MB
nigelpoulton/tu-demo	v1	c610c6a38555	22 months ago	58.1MB
nigelpoulton/tu-demo	v2	6ba12825d092	16 months ago	58.6MB

This is a perfect example of the warning issued earlier about the `latest` tag. In this example, the `latest` tag refers to the same image as the `v1` tag. This means it's pointing to the older of the two images! Moral of the story, `latest` is an arbitrary tag and is not guaranteed to point to the newest image in a repository!

Filtering the output of `docker images`

Docker provides the `--filter` flag to filter the list of images returned by `docker images`.

The following example will only return dangling images.

```
$ docker images --filter dangling=true
REPOSITORY    TAG        IMAGE ID      CREATED       SIZE
<none>        <none>     4fd34165afe0  7 days ago   14.5MB
```

A *dangling image* is one that is no longer tagged and appears in listings as `<none>: <none>`. A common way they occur is when building a new image with a tag that already exists. When this happens, Docker will build the new image, notice that an existing image already has the same tag, remove the tag from the existing image and give it to the new image.

Consider this example, you build a new application image based on `alpine:3.4` and tag it as `dodge:challenger`. Then you update the image to use `alpine:3.5` instead of `alpine:3.4`. When you build the new image, the operation will create a new image tagged as `dodge:challenger` and remove the tags from the old image. The old image will become a *dangling image*.

You can delete all dangling images on a system with the `docker image prune` command. If you add the `-a` flag, Docker will also remove all unused images (those not in use by any containers).

Docker currently supports the following filters:

- `dangling`: Accepts `true` or `false`, and returns only dangling images (`true`), or non-dangling images (`false`).
- `before`: Requires an image name or ID as argument, and returns all images created before it.
- `since`: Same as above, but returns images created after the specified image.
- `label`: Filters images based on the presence of a label or label and value. The `docker images` command does not display labels in its output.

For all other filtering you can use `reference`.

Here's an example using `reference` to display only images tagged as "latest". At the time of writing this works on some Docker installations and not others (possibly not working on systems that use `containerd` for image management).

```
$ docker images --filter=reference="*:latest"
REPOSITORY    TAG        IMAGE ID      CREATED       SIZE
busybox       latest     3596868f4ba8  7 days ago   3.72MB
alpine        latest     44dd6f223004  9 days ago   7.73MB
redis         latest     2334573cc576  2 weeks ago  111MB
```

You can also use the `--format` flag to format output using Go templates. For example, the following command will only return the size property of images on a Docker host.

```
$ docker images --format "{{.Size}}"
3.72MB
7.73MB
111MB
265MB
58.1MB
```

Use the following command to return all images, but only display repo, tag and size.

```
$ docker images --format "{{.Repository}}: {{.Tag}}: {{.Size}}"
busybox: latest: 3.72MB
alpine: latest: 7.73MB
redis: latest: 111MB
portainer/portainer-ce: latest: 265MB
nigelpoulton/tu-demo: latest: 58.1MB
<Snip>
```

If you need more powerful filtering, you can always use the tools provided by your OS and shell such as `grep` and `awk`. You may also find a Docker Desktop extension that's useful.

Searching Docker Hub from the CLI

The `docker search` command lets you search Docker Hub from the CLI. It has limited value as you can only pattern-match against strings in the “NAME” field. However, you can filter output based on any of the returned columns.

In its simplest form, it searches for all repos containing a certain string in the “NAME” field. For example, the following command searches for all repos with “nigelpoulton” in the “NAME” field.

```
$ docker search nigelpoulton
```

NAME	DESCRIPTION	STARS	AUTOMATED
nigelpoulton/pluralsight..	Web app used in...	22	[OK]
nigelpoulton/tu-demo		12	
nigelpoulton/k8sbook	Kubernetes Book web app	2	
nigelpoulton/workshop101	Kubernetes 101 Workshop	0	

```
<Snip>
```

The “NAME” field is the repository name. This includes the Docker ID, or organization name, for unofficial repositories. For example, the following command lists all repositories that include the string “alpine” in the name.

```
$ docker search alpine
NAME                DESCRIPTION          STARS   OFFICIAL   AUTOMATED
alpine              A minimal Docker..  9962    [OK]
rancher/alpine-git  Alpine Linux with..  1
grafana/alpine      Alpine Linux with..  4
<Snip>
```

Notice how some of the repositories returned are official and some are unofficial. You can use `--filter "is-official=true"` so that only official repos are displayed.

```
$ docker search alpine --filter "is-official=true"
NAME                DESCRIPTION          STARS   OFFICIAL   AUTOMATED
alpine              A minimal Docker..  9962    [OK]
```

One last thing about `docker search`. By default, Docker will only display 25 lines of results. However, you can use the `--limit` flag to increase that to a maximum of 100.

Images and layers

A Docker image is a collection of loosely-connected read-only layers where each layer comprises one or more files. Figure 6.3 shows an image with 5 layers.

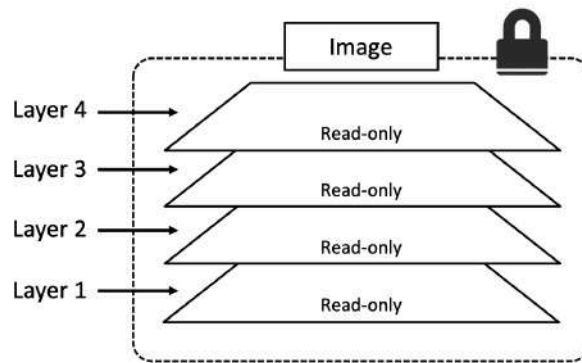


Figure 6.3

Docker takes care of stacking the layers and representing them as a single unified object. There are a few ways to see and inspect the layers that make up an image. In fact, we saw one earlier when pulling images. The following example looks closer at an image pull operation.

```
$ docker pull ubuntu:latest
latest: Pulling from library/ubuntu
952132ac251a: Pull complete
82659f8f1b76: Pull complete
c19118ca682d: Pull complete
8296858250fe: Pull complete
24e0251a0e2c: Pull complete
Digest: sha256:f4691c96e6bbaa99d...28ae95a60369c506dd6e6f6ab
Status: Downloaded newer image for ubuntu:latest
docker.io/ubuntu:latest
```

Each line in the output above that ends with “Pull complete” represents a layer in the image the was pulled. As we can see, this image has 5 layers and is shown in Figure 6.4 with layer IDs.

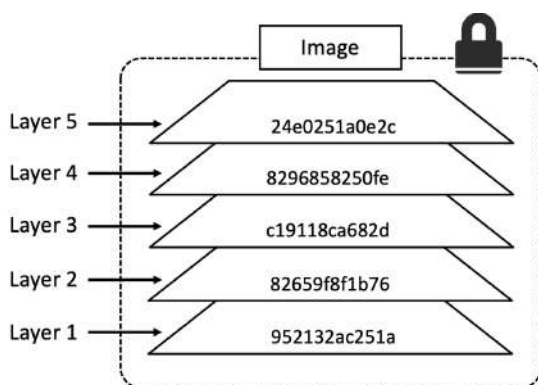


Figure 6.4

Another way to see the layers of an image is to inspect the image with the `docker inspect` command. The following example inspects the same `ubuntu:latest` image.

```
$ docker inspect ubuntu:latest
[
  {
    "Id": "sha256:bd3d4369ae.....fa2645f5699037d7d8c6b415a10",
    "RepoTags": [
      "ubuntu:latest"
    ]
  }
]
<Snip>
"RootFS": {
  "Type": "layers",
  "Layers": [
    "sha256:c8a75145fc...894129005e461a43875a094b93412",
    "sha256:c6f2b330b6...7214ed6aac305dd03f70b95cdc610",
    "sha256:055757a193...3a9565d78962c7f368d5ac5984998",
```

```

    "sha256:4837348061...12695f548406ea77feb5074e195e3",
    "sha256:0cad5e07ba...4bae4cfc66b376265e16c32a0aae9"
  ]
}
]

```

The trimmed output shows 5 layers again. Only this time they're shown using their SHA256 hashes.

The `docker inspect` command is a great way to see the details of an image.

The `docker history` command is another way of inspecting an image and seeing layer data. However, it shows the build history of an image and is **not** a strict list of layers in the final image. For example, some Dockerfile instructions ("ENV", "EXPOSE", "CMD", and "ENTRYPOINT") add metadata to the image and do not create a layer.

All Docker images start with a base layer, and as changes are made and new content is added, new layers are added on top.

Consider the following oversimplified example of building a simple Python application. You might have a corporate policy that all applications are based on the official Ubuntu 22:04 image. This would be your image's *base layer*. Adding the Python package will add a second layer on top of the base layer. If you later add source code files, these will be added as additional layers. The final image will have three layers as shown in Figure 6.5 (remember this is an over-simplified example for demonstration purposes).

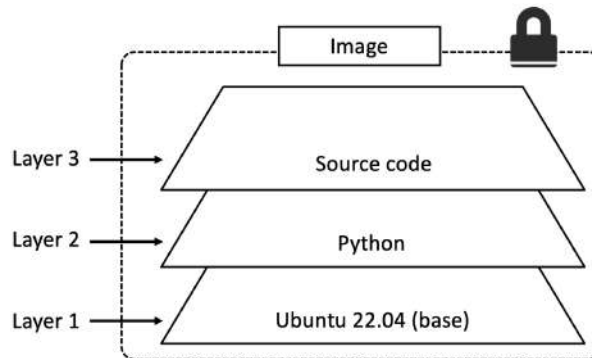


Figure 6.5

It's important to understand that as additional layers are added, the *image* is always the combination of all layers stacked in the order they were added. Take a simple example of two layers as shown in Figure 6.6. Each *layer* has 3 files, but the overall *image* has 6 files as it is the combination of both layers.

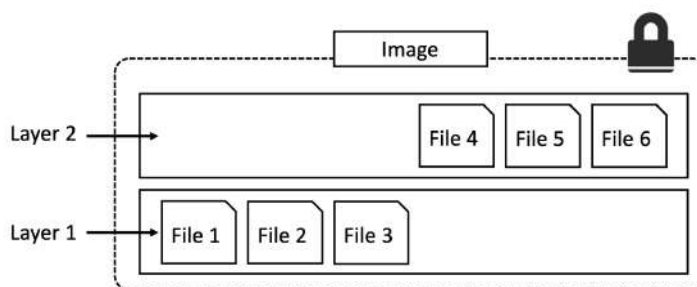


Figure 6.6

Note: We've shown the image layers in Figure 6.6 in a slightly different way to previous figures. This is just to make showing the files easier.

In the slightly more complex example of the three-layer image in Figure 6.7, the overall image only presents 6 files in the unified view. This is because `File 7` in the top layer is an updated version of `File 5` directly below (inline). In this situation, the file in the higher layer obscures the file directly below it. This allows updated versions of files to be added as new layers to the image.

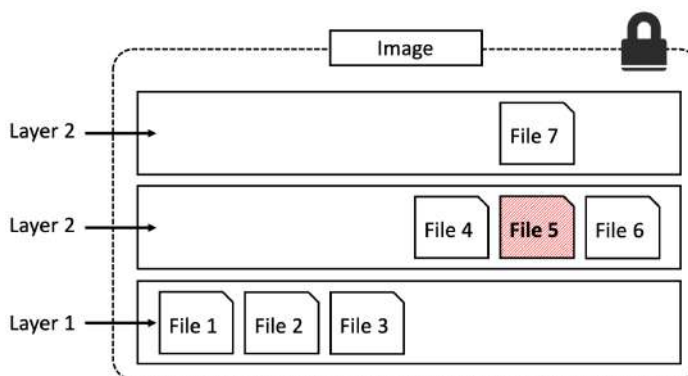


Figure 6.7

Docker employs a storage driver that is responsible for stacking layers and presenting them as a single unified filesystem/image. Examples of storage drivers on Linux include `overlay2`, `devicemapper`, `btrfs` and `zfs`. As their names suggest, each one is based on a Linux filesystem or block-device technology, and each has its own unique performance characteristics.

No matter which storage driver is used, the user experience is the same.

Figure 6.8 shows the same 3-layer image as it will appear to the system. I.e. all three layers stacked and merged, giving a single unified view.

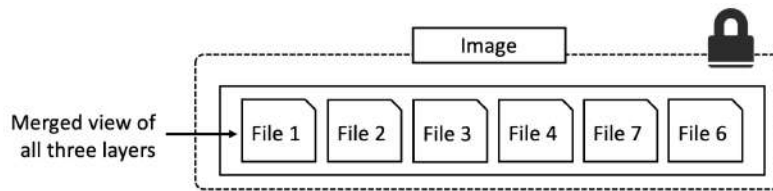


Figure 6.8

Sharing image layers

Multiple images can, and do, share layers. This leads to efficiencies in space and performance.

The following example shows the output of a `docker pull` command with the `-a` flag. This can be used to download all images in a repository. The command has limitations and may fail if the repository has images for multiple platforms and architectures such as Linux and Windows, or amd64 and arm64.

```
$ docker pull -a nigelpoulton/tu-demo
latest: Pulling from nigelpoulton/tu-demo
aad63a933944: Pull complete
f229563217f5: Pull complete
<Snip>
Digest: sha256:c9f8e18822...6cbb9a74cf

v1: Pulling from nigelpoulton/tu-demo
aad63a933944: Already exists
f229563217f5: Already exists
<Snip>
fc669453c5af: Pull complete
Digest: sha256:674cb03444...f8598e4d2a

v2: Pulling from nigelpoulton/tu-demo
Digest: sha256:c9f8e18822...6cbb9a74cf
Status: Downloaded newer image for nigelpoulton/tu-demo
docker.io/nigelpoulton/tu-demo

$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nigelpoulton/tu-demo	latest	d5e1e48cf932	2 weeks ago	104MB
nigelpoulton/tu-demo	v2	d5e1e48cf932	2 weeks ago	104MB
nigelpoulton/tu-demo	v1	6852022de69d	2 weeks ago	104MB

Notice the lines ending in `Already exists`.

These lines tell us that Docker is smart enough to recognize when it's being asked to pull an image layer that it already has a local copy of. In this example, Docker pulled the

image tagged as `latest` first. Then, when it pulled the `v1` and `v2` images, it noticed that it already had some of the layers that make up those images. This happens because the three images in this repository are almost identical, and therefore share many layers. In fact, the only difference between `v1` and `v2` is the top layer.

As mentioned previously, Docker on Linux supports many storage drivers. Each is free to implement image layering, layer sharing, and copy-on-write (CoW) behaviour in its own way. However, the end result and user experience is the same.

Pulling images by digest

So far, we've shown you how to pull images using their name (tag). This is by far the most common method, but it has a problem — tags are mutable! This means it's possible to accidentally tag an image with the wrong tag (name). Sometimes, it's even possible to tag an image with the same tag as an existing, but different, image. This can cause problems!

As an example, imagine you've got an image called `golftack:1.5` and it has a known bug. You pull the image, apply a fix, and push the updated image back to its repository using the **same tag**.

Take a moment to consider what happened there... You have an image called `golftack:1.5` that has a bug. That image is being used by containers in your production environment. You create a new version of the image that includes a fix. Then comes the mistake... you build and push the fixed image back to its repository with the **same tag as the vulnerable image!** This overwrites the original image and leaves you without a great way of knowing which of your production containers are using the vulnerable image and which are using the fixed image — they both have the same tag!

This is where *image digests* come to the rescue.

Docker supports content addressable storage model. As part of this model, all images get a cryptographic *content hash*. For the purposes of this discussion, we'll call this hash as the *digest*. As the digest is a hash of the contents of the image, it's impossible to change the contents of the image without creating a new unique digest. Put another way, you cannot change the content of an image and keep the old digest. This means digests are immutable and provide a solution to the problem we just mentioned.

Every time you pull an image, the `docker pull` command includes the image's digest as part of the information returned. You can also view the digests of images in your Docker host's local repository by adding the `--digests` flag to the `docker images` command. These are both shown in the following example.

```
$ docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
08409d417260: Pull complete
Digest: sha256:02bb6f42...44c9b11
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
```

```
$ docker images --digests alpine
```

REPOSITORY	TAG	DIGEST	IMAGE ID	CREATED	SIZE
alpine	latest	sha256:02bb6f42...44c9b11	44dd6f223004	9 days ago	7.73MB

The snipped output above shows the digest for the alpine image as -

```
sha256:02bb6f42...44c9b11
```

Now that we know the digest of the image, we can use it when pulling the image again. This will ensure that we get **exactly the image we expect!**

At the time of writing, there is no native Docker command that will retrieve the digest of an image from a remote registry such as Docker Hub. This means the only way to determine the digest of an image is to pull it by tag and then make a note of its digest. This may change in the future.

The following example deletes the `alpine:latest` image from your Docker host and then shows how to pull it again using its digest instead of its tag. The actual digest is truncated in the book so that it fits on one line. Substitute this for the real digest of the version you pulled on your own system.

```
$ docker rmi alpine:latest
Untagged: alpine:latest
Untagged: alpine@sha256:02bb6f428431fbc2809c5d1b41eab5a68350194fb508869a33cb1af4444c9b11
Deleted: sha256:44dd6f2230041eede4ee5e792728313e43921b3e46c1809399391535c0c0183b
Deleted: sha256:94dd7d531fa5695c0c033dcb69f213c2b4c3b5a3ae6e497252ba88da87169c3f

$ docker pull alpine@sha256:02bb6f42...44c9b11
docker.io/library/alpine@sha256:02bb6f42...44c9b11: Pulling from library/alpine
08409d417260: Pull complete
Digest: sha256:02bb6f428431...9a33cb1af4444c9b11
Status: Downloaded newer image for alpine@sha256:02bb6f428431...9a33cb1af4444c9b11
docker.io/library/alpine@sha256:02bb6f428431...9a33cb1af4444c9b11
```

A little bit more about image hashes (digests)

As previously mentioned, an image is a loose collection of independent layers.

In some ways, the *image* is just a manifest file that lists the layers and some metadata. The application and dependencies live in the *layers* and each layer is fully independent with no concept of being part of something bigger.

Each *image* is identified by a crypto ID that is a hash of the manifest file. Each *layer* is identified by a crypto ID that is a hash of the layer content.

This means that changing the contents of the image, or any of its layers, will cause the associated crypto hashes to change. As a result, images and layers are immutable and we can easily identify if changes have been made.

So far, things are pretty simple. But they're about to get a bit more complicated.

When we push and pull images, the layers are compressed to save network bandwidth and storage space in the registry. However, compressed content is different to uncompressed content. As a result, content hashes no longer match after push or pull operations.

This presents various problems. For example, Docker Hub verifies every pushed layer to make sure it wasn't tampered with en route. To do this, it runs a hash against the layer content and checks it against the hash that was sent. As the layer was compressed the hash verification will fail.

To get around this, each layer also gets something called a *distribution hash*. This is a hash of the compressed version of the layer and is included with every layer pushed and pulled to a registry. This is used to verify that the layer arrived without being tampered with.

Multi-architecture images

One of the best things about Docker is its simplicity. However, as technologies grow they get more complex. This happened for Docker when it started supporting different platforms and architectures such as Windows and Linux, on variations of ARM, x64, PowerPC, and s390x. All of a sudden, popular images had versions for different platforms and architectures and as users we had to add extra steps to make sure we were pulling the right version for our environments. This broke the smooth Docker experience.

Note: We're using the term "architecture" to refer to CPU architecture such as x64 and ARM. We use the term "platform" to refer to either the OS (Linux or Windows) or the combination of OS and architecture.

Multi-architecture images to the rescue!

Fortunately, there's a slick way of supporting multi-arch images. This means a single image, such as `golang:latest`, can have images for Linux on x64, Linux on PowerPC,

Windows x64, Linux on different versions of ARM, and more. To be clear, we're talking about a single image tag supporting multiple platforms and architectures. We'll see it in action in a second, but it means you can run a simple `docker pull golang:latest` from any platform or architecture and Docker will pull the correct image.

To make this happen, the Registry API supports two important constructs:

- **manifest lists**
- **manifests**

The **manifest list** is exactly what it sounds like: a list of architectures supported by a particular image tag. Each supported architecture then has its own *manifest* that lists the layers used to build it.

Figure 6.9 uses the official `golang` image as an example. On the left is the **manifest list** with entries for each architecture the image supports. The arrows show that each entry in the **manifest list** points to a **manifest** containing image config and layer data.

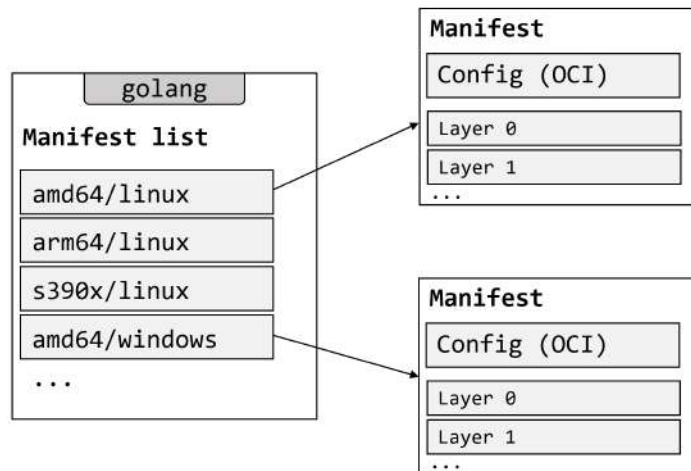


Figure 6.9

Let's look at the theory before seeing it in action.

Assume you're running Docker on a Raspberry Pi (Linux on ARM). When you pull an image, Docker makes the relevant calls to Docker Hub. If a **manifest list** exists for the image, it will be parsed to see if an entry exists for Linux on ARM. If it exists, the **manifest** for the Linux ARM image is retrieved and parsed for the crypto ID's of the layers. Each layer is then pulled from Docker Hub and assembled on the Docker host.

Let's see it in action.

The following examples start are from a Linux ARM system and a Windows x64 system. Both start a new container based the official golang image and run the `go version` command. The outputs show the version of Go as well as the platform and CPU architecture of the host. Notice how both commands are exactly the same and Docker takes care of getting the right image for the platform and architecture!

Linux on arm64 example:

```
$ docker run --rm golang go version
<Snip>
go version go1.20.4 linux/arm64
```

Windows on x64 example:

```
> docker run --rm golang go version
<Snip>
go version go1.20.4 windows/amd64
```

The Windows Golang image is currently over 2GB in size and may take a long time to download.

The `'docker manifest'` command lets you inspect the manifest list of any image on Docker Hub. The following example inspects the manifest list on Docker Hub for the `golang` image. You can see that Linux and Windows are supported on various CPU architectures. You can run the same command without the `grep` filter to see the full JSON manifest list.

```
$ docker manifest inspect golang | grep 'architecture\|os'
    "architecture": "amd64",
    "os": "linux"
    "architecture": "arm",
    "os": "linux",
    "architecture": "arm64",
    "os": "linux",
    "architecture": "386",
    "os": "linux"
    "architecture": "mips64le",
    "os": "linux"
    "architecture": "ppc64le",
    "os": "linux"
    "architecture": "s390x",
    "os": "linux"
    "architecture": "amd64",
    "os": "windows",
    "os.version": "10.0.20348.1726"
    "architecture": "amd64",
    "os": "windows",
    "os.version": "10.0.17763.4377"
```

All official images have manifest lists.

You can create your own builds for different platforms and architectures with `docker buildx` and then use `docker manifest create` to create your own manifest lists.

The following command builds an image for ARMv7 called `myimage:arm-v7` from the current directory. It's based on code in <https://github.com/nigelpoulton/psweb>.

```
$ docker buildx build --platform linux/arm/v7 -t myimage:arm-v7 .
[+] Building 43.5s (11/11) FINISHED
=> [internal] load build definition from Dockerfile          0.0s
=> => transferring dockerfile: 368B                          0.0s
<Snip>
=> => exporting manifest list sha256:2a621c3d06...84f9395d6 0.0s
=> => naming to docker.io/library/myimage:arm-v7            0.0s
=> => unpacking to docker.io/library/myimage:arm-v7          0.8s
```

The beauty of the command is that you don't have to run it from an ARMv7 Docker node. In fact, the example shown was ran on Linux on x64 hardware.

Deleting Images

When you no longer need an image on your Docker host, you can delete it with the `docker rmi` command. `rmi` is short for remove image.

Deleting an image will remove the image and all of its layers from your Docker host. This means it will no longer show up in `docker images` commands and all directories on the Docker host containing the layer data will be deleted. However, if an image layer is shared by another image, it won't be deleted until all images that reference it have been deleted.

Delete the images pulled in the previous steps with the `docker rmi` command. The following example deletes an image by its ID, this might be different on your system.

```
$ docker rmi 44dd6f223004
Untagged: alpine@sha256:02bb6f428431fbc2809c5d1...9a33cb1af4444c9b11
Deleted: sha256:44dd6f2230041eede4ee5e7...09399391535c0c0183b
Deleted: sha256:94dd7d531fa5695c0c033dc...97252ba88da87169c3f
```

You can list multiple images on the same command by separating them with whitespace like the following.

```
$ docker rmi f70734b6a266 a4d3716dbb72
```

You won't be able to delete an image if it's in use by a running container. You'll need to stop and delete any containers before deleting the image.

A handy shortcut for **deleting all images** on a Docker host is to run the `docker rmi` command and pass it a list of all image IDs on the system by calling `docker images` with the `-q` flag. This is shown next.

If you are following along on a Windows system, this will only work in a PowerShell terminal. It will not work on a CMD prompt.

```
$ docker rmi $(docker images -q) -f
```

To understand how this works, download a couple of images and then run `docker images -q`.

```
$ docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
08409d417260: Pull complete
Digest: sha256:02bb6f428431fbc2809c5...a33cb1af4444c9b11
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
```

```
$ docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
79d0ea7dc1a8: Pull complete
Digest: sha256:dfd64a3b4296d8c9b62aa3...ee20739e8eb54fbf
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
```

```
$ docker images -q
44dd6f223004
3f5ef9003cef
```

See how `docker images -q` returns a list containing just the image IDs of all local images. Passing this list to `docker rmi` will delete all images on the system as shown next.

```
$ docker rmi $(docker images -q) -f
Untagged: alpine:latest
Untagged: alpine@sha256:02bb6f428431fb...a33cb1af4444c9b11
Deleted: sha256:44dd6f2230041...09399391535c0c0183b
Deleted: sha256:94dd7d531fa56...97252ba88da87169c3f
Untagged: ubuntu:latest
Untagged: ubuntu@sha256:dfd64a3b4296d8...9ee20739e8eb54fbf
Deleted: sha256:3f5ef9003cefb...79cb530c29298550b92
Deleted: sha256:b49483f6a0e69...f3075564c10349774c3
```

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

Let's remind ourselves of the major commands for working with Docker images.

Images - The commands

- `docker pull` is the command to download images from remote registries. By default, images will be pulled from Docker Hub but you can pull from other registries. This command will pull the image tagged as `latest` from the `alpine` repository on Docker Hub: `docker pull alpine:latest`.
- `docker images` lists all of the images stored in your Docker host's local image cache. Add the `--digests` flag to see the SHA256 digests.
- `docker inspect` is a thing of beauty! It gives you all of the glorious details of an image — layer data and metadata.
- `docker manifest inspect` lets you to inspect the manifest list of any image stored on Docker Hub. This command will show the manifest list for the `redis` image: `docker manifest inspect redis`.
- `docker buildx` is a Docker CLI plugin that extends the Docker CLI to support multi-arch builds.
- `docker rmi` is the command to delete images. This command will delete the `alpine:latest` image — `docker rmi alpine:latest`. You cannot delete an image that is associated with a container in the running (Up) or stopped (Exited) states.

Chapter summary

In this chapter, we learned about container images. We learned that they contain everything needed to run an application as a container. This includes just enough OS, source code files, dependencies, and metadata. Images are used to start containers and are similar to VM templates or object-oriented programming classes. Under the hood they are made up of one or more read-only layers, that when stacked together, make up the overall image.

We used the `docker pull` command to pull some images into our local Docker host.

We covered image naming, official and unofficial repos, layering, sharing, and crypto IDs.

We looked at how Docker supports multi-architecture and multi-platform images, and we finished off by looking at some of the most common commands used to work with images.

In the next chapter we'll take a similar tour of containers — the runtime sibling of images.

7: Containers

Docker implements the Open Container Initiative (OCI) specifications. This means everything you learn in this chapter applies to other container runtimes and platforms that implement the OCI specifications.

We'll split this chapter into the usual three parts:

- The TLDR
- The deep dive
- The commands

Docker containers - The TLDR

A container is the runtime instance of an image. In the same way that you can start a virtual machine (VM) from a virtual machine template, you start one or more containers from a single image. The big difference between a VM and a container is that containers are smaller, faster, and more portable.

Figure 7.1 shows a single Docker image being used to start multiple Docker containers.

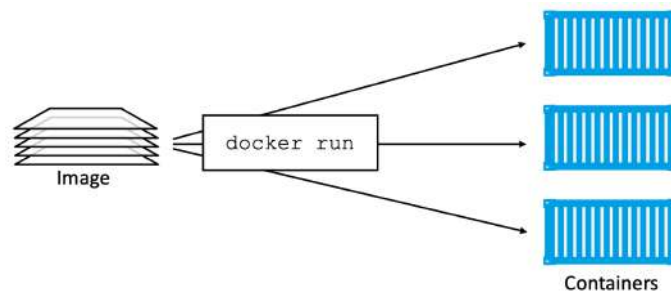


Figure 7.1

The simplest way to start a container is with the `docker run` command. The command can take a lot of arguments, but in its most basic form you tell it an image to use and an app to run: `docker run <image> <app>`. The following command will start a new container based on the Ubuntu Linux image and start a Bash shell.

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

The `-it` flags connect your current terminal window to the container's shell.

Containers run until the main app exits. In the previous example, the container will exit when the Bash shell exits.

A simple way to demonstrate this is to start a new container and tell it to run the sleep command for 10 seconds. The container will start, seize your terminal for 10 seconds, then exit. The following is a simple way to demonstrate this on a Linux host.

```
$ docker run -it alpine:latest sleep 10
```

You can manually stop a running container with `docker stop` and restart it with `docker start`. To get rid of a container forever, you have to explicitly delete it with `docker rm`.

That's the elevator pitch! Now let's get into the detail...

Docker containers - The deep dive

The first things we'll cover are the fundamental differences between a container and a VM. It's mainly theory at this point, but it's important stuff.

Containers vs VMs

Containers and VMs both need a host to run on. The host can be anything from your laptop, a bare metal server in your data center, or an instance in the public cloud.

Let's assume a requirement where your business has a single physical server that needs to run 4 business applications.

In the VM model, the physical server is powered on and the hypervisor boots. Once booted, the hypervisor claims all physical resources such as CPU, RAM, storage, and network cards. It then carves these hardware resources into virtual constructs that look smell and feel exactly like the real thing. It then packages them into a software construct called a virtual machine (VM). We take those VMs and install an operating system and application on each one.

Assuming the scenario of a single physical server needing to run 4 business applications — we'd create 4 VMs, install 4 operating systems, and then install the 4 applications. When it's all done it looks like Figure 7.2.

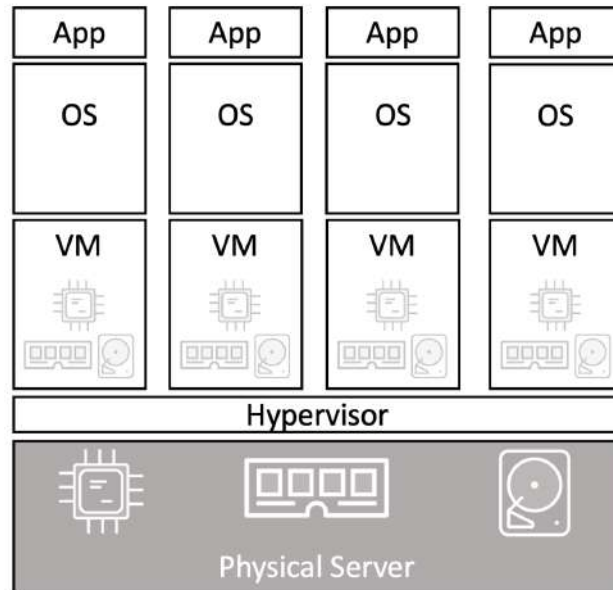


Figure 7.2

Things are a bit different in the container model.

The server is powered on and the OS boots. In this container model the host's OS claims all hardware resources. Next you install a container engine such as Docker. The container engine then carves-up the **OS resources** (*process tree, filesystem, network stack* etc) and packages them into virtual operating systems called *containers*. Each container looks smells and feels just like a real OS. Inside of each *container* we run an application.

If we assume the same scenario of a single physical server needing to run 4 business applications, we'd carve the OS into 4 containers and run a single application inside each. This is shown in Figure 7.3.

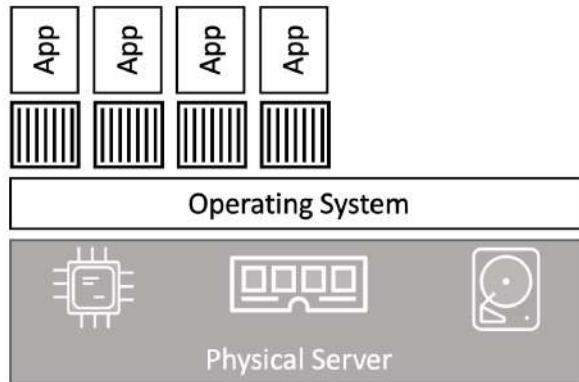


Figure 7.3

At a high level, hypervisors perform **hardware virtualization** — they carve up physical hardware resources into virtual versions called VMs. On the other hand, containers perform **OS virtualization** — they carve OS resources into virtual versions called containers.

The VM tax

Let's build on what we just covered and drill into one of the problems with the hypervisor model.

We started out with a single physical server and the requirement to run 4 business applications. In the VM model we installed a specialised OS called a hypervisor, in the container model we install any modern OS. So far, the models are almost identical. But this is where the similarities stop.

The VM model carves **low-level hardware resources** into VMs. Each VM is a software construct containing virtual CPUs, virtual RAM, virtual disks etc. As such, every VM needs its own OS to claim, initialize, and manage all of those virtual resources. Unfortunately, every OS comes with its own set of baggage and overheads. For example, every OS consumes CPU, RAM, and storage. Some need their own licenses, as well as people and infrastructure to patch and update them. Each OS also presents a sizable attack surface. We often refer to all of this as the **OS tax**, or **VM tax** — every OS is stealing resources you'd rather assign to applications.

There's only one OS kernel in the container model, and that's running on the shared host. And it's possible to hundreds of containers on a single host with a single shared OS. This means a one OS consuming CPU, RAM, and storage. It also means a single OS that needs licensing, a single OS that needs updating and patching, and a single OS presenting an attack surface. All in all, a single OS tax bill.

That might not seem a lot in our example of a single server running 4 business applications. But when you start talking about hundreds or thousands of apps, it's a game-changer.

Another thing to consider is application start times. Containers start a lot faster than VMs because they only have to start the application — the kernel is already up and running on the host. In the VM model, each VM needs to boot a full OS before it can start the app.

This all makes the container model leaner and more efficient than the VM model. You can pack more applications onto less resources, start them faster, and pay less in licensing and admin costs, as well as present less of an attack surface!

Early versions of containers and container platforms were considered less secure than VMs. However, that's been changing and most container engines and platforms now deploy containers with “*sensible defaults*” that attempt to lock things down without the security making things unusable. Lots of technologies exist that can make containers more secure than VMs, however, they're sometimes hard to configure. These include *SELinux*, *AppArmor*, *seccomp*, *capabilities*, and more.

With the theory out of the way, let's get our hands-on with containers.

Running containers

You'll need a working Docker host to follow along with the examples. I recommend Docker Desktop or Multipass from Canonical. Just search the web or ask your AI how to install them, they're super easy.

Checking that Docker is running

The first thing I always do when I log on to a Docker host is run a `docker version` to check that Docker is running. It's a good command because it checks the CLI and engine components.

```
$ docker version
Client: Docker Engine - Community
Version:      24.0.0
API version:  1.43
OS/Arch:      linux/arm64
Context:      default
<Snip>
Server: Docker Engine - Community
Engine:
Version:      24.0.0
API version:  1.43 (minimum version 1.12)
```

```
OS/Arch:          linux/arm64
<Snip>
```

As long as you get a response back in the Client and Server you should be good to go. If you get an error code in the Server section, there's a good chance that the Docker daemon (server) isn't running, or your user account doesn't have permission to access it.

On Linux you need to make sure your user account is a member of the local docker Unix group. If it isn't, you can add it with `usermod -aG docker <user>` and then you'll have to restart your shell for the changes to take effect. Alternatively, you can prefix all docker commands with `sudo`.

If your user account is already a member of the local docker group, the problem might be that the Docker daemon isn't running. To check the status of the Docker daemon, run one of the following commands depending on your Docker host's operating system.

Linux systems not using Systemd.

```
$ service docker status
docker start/running, process 29393
```

Linux systems using Systemd.

```
$ systemctl is-active docker
active
```

Starting a simple container

The simplest way to start a container is with the `docker run` command.

This command starts a simple container that will run a containerized version of Ubuntu Linux.

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

Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
79d0ea7dc1a8: Pull complete
Digest: sha256:dfd64a3b42...47492599ee20739e8eb54fbf
Status: Downloaded newer image for ubuntu:latest
root@e37f24dc7e0a:/#
```

Let's take a closer look at the command.

`docker run` tells Docker to run a new container. The `-it` flags make the container interactive and attach it to your terminal. The `ubuntu:latest` argument tells Docker which image to use to start the container. Finally, `/bin/bash` is the application to run in the container.

When you hit Return, the Docker client packages up the command and POSTs it to the API server running on the Docker daemon. The Docker daemon accepts the command and searches the host's local image repository to see if it already has a copy of the image. In our example it didn't, so it went to Docker Hub to see if it could find it there. It found it, pulled it locally, and stored it in the local cache.

Note: In a standard, out-of-the-box Linux installation, the Docker daemon implements the Docker API on a local Unix socket at `/var/run/docker.sock`. On Windows, it listens on a named pipe at `npipes:///./pipe/docker_engine`. It's also possible to configure the Docker daemon to listen on the network. The default non-TLS network port for Docker is 2375, the default TLS port is 2376.

Once the image is pulled, the daemon instructs `containerd` to start the container. `containerd` tasks `runc` with creating the container and starting the app.

If you're following along, your terminal is now attached to the container — look closely and you'll see that your shell prompt has changed. In the example it's `root@e37f24dc7e0a:/#` but yours will be different. The long number after the `@` is the first 12 characters of the container's unique ID.

Try executing some basic commands inside of the container. You might notice that some of them don't work. This is because the images are optimized to be lightweight and don't have all of the normal commands and packages installed. The following example shows a couple of commands — one succeeds and the other one fails.

```
root@50949b614477:/# ls -l
total 64
lrwxrwxrwx 1 root root 7 Apr 23 11:06 bin -> usr/bin
drwxr-xr-x 2 root root 4096 Apr 15 11:09 boot
drwxr-xr-x 5 root root 360 Apr 27 17:24 dev
drwxr-xr-x 1 root root 4096 Apr 27 17:24 etc
drwxr-xr-x 2 root root 4096 Apr 15 11:09 home
lrwxrwxrwx 1 root root 7 Apr 23 11:06 lib -> usr/lib
<Snip>

root@50949b614477:/# ping nigelpoulton.com
bash: ping: command not found
```

As you can see, the `ping` utility isn't included as part of the official Ubuntu image.

Container processes

When we started the Ubuntu container, we told it to run the Bash shell (`/bin/bash`). This makes the Bash shell the **one-and-only process running inside the container**. You can see this by running `ps -elf` from inside the container.

```
root@e37f24dc7e0a:/# ps -elf
F S UID    PID  PPID  NI  ADDR SZ  WCHAN  STIME TTY      TIME      CMD
4 S root     1     0    0   -  4558 wait  00:47 ?        00:00:00  /bin/bash
0 R root    11     1    0   -  8604 -    00:52 ?        00:00:00  ps -elf
```

The first process in the list, with PID 1, is the Bash shell we told the container to run. The second process is the `ps -elf` command we ran to produce the list. This is a short-lived process that exits as soon as the output is displayed. Long story short, this container is running a single process — `/bin/bash`.

Typing `exit` while logged in to the container will terminate the Bash process and the whole container will exit (terminate). This is because a container cannot exist without its designated main process. This is true of Linux and Windows containers — **killing the main process in the container will kill the container**.

Press `Ctrl-PQ` to exit the container without terminating its main process. Doing this will place you back in the shell of your Docker host and leave the container running in the background. You can use the `docker ps` command to view the list of running containers on your system.

```
$ docker ps
CNTNR ID      IMAGE          COMMAND        CREATED   STATUS    NAMES
e37..7e0a    ubuntu:latest  /bin/bash      6 mins      Up 6mins   sick_montalcini
```

It's important to understand that this container is still running and you can re-attach your terminal to it with the `docker exec` command.

```
$ docker exec -it e37f24dc7e0a bash
root@e37f24dc7e0a:/#
```

As you can see, the shell prompt has changed back to the container. If you run the `ps -elf` command again you will now see **two** Bash processes. This is because the `docker exec` command created a new Bash process and attached to that. This means typing `exit` in this shell will not terminate the container, because the original Bash process will continue running.

Type `exit` to leave the container and verify it's still running with a `docker ps`. It will still be running.

If you are following along with the examples, you should stop and delete the container with the following two commands (you will need to substitute the ID of your container). It may take a few seconds for the container to gracefully stop.

```
$ docker stop e37f24dc7e0a
e37f24dc7e0a

$ docker rm e37f24dc7e0a
e37f24dc7e0a
```

Container lifecycle

In this section, we'll look at the lifecycle of a container — from birth, through vacations and work, to eventual death.

We've already seen how to start containers with the `docker run` command. Let's start another one so we can walk it through its entire lifecycle.

```
$ docker run --name percy -it ubuntu:latest /bin/bash
root@9cb2d2fd1d65:/#
```

That's the container created, and we named it "percy" for persistent.

Now let's put it to work by writing some data to it.

The following procedure writes some text to a new file in the `/tmp` directory and verifies the operation succeeded. Be sure to run these commands from within the container you just started.

```
root@9cb2d2fd1d65:/# cd tmp

root@9cb2d2fd1d65:/tmp# ls -l
total 0

root@9cb2d2fd1d65:/tmp# echo "Sunderland is the greatest football team in the world" > newfile

root@9cb2d2fd1d65:/tmp# ls -l
total 4
-rw-r--r-- 1 root root 14 Apr 27 11:22 newfile

root@9cb2d2fd1d65:/tmp# cat newfile
Sunderland is the greatest football team in the world
```

Press `Ctrl-PQ` to exit the container without killing it.

Now use the `docker stop` command to stop the container and put it on *vacation*. It'll take a few seconds while it travels to its vacation destination ;-)

```
$ docker stop percy
percy
```

You can use the container's name or ID with the `docker stop` command. The format is `docker stop <container-id or container-name>`.

Now run a `docker ps` to list all running containers.

```
$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED    STATUS    PORTS    NAMES
```

The container isn't listed in the output because it's in the stopped state. Run the same command again with the `-a` flag to show all containers, including those that are stopped.

```
$ docker ps -a
CNTNR ID   IMAGE           COMMAND                  CREATED    STATUS    NAMES
9cb...65   ubuntu:latest   /bin/bash               4 mins     Exited (0)   percy
```

This time we see the container showing as `Exited (137)`. Stopping a container is like stopping a virtual machine — it's no longer running, but its entire configuration and contents still exist on the Docker host. This means it can be restarted at any time.

Let's use the `docker start` command to bring it back from vacation.

```
$ docker start percy
percy
```

```
$ docker ps
CONTAINER ID   IMAGE           COMMAND                  CREATED    STATUS    NAMES
9cb2d2fd1d65   ubuntu:latest   "/bin/bash"             4 mins     Up 7 secs   percy
```

The stopped container is now restarted. Time to verify that the file we created earlier still exists. Connect to the restarted container with the `docker exec` command.

```
$ docker exec -it percy bash
root@9cb2d2fd1d65:/#
```

Your shell prompt will change to show that you are now operating within the namespace of the container.

Verify the file is still there and contains the data you wrote to it.

```

root@9cb2d2fd1d65:/# cd tmp
root@9cb2d2fd1d65:/# ls -l
-rw-r--r-- 1 root root 14 Sep 13 04:22 newfile

root@9cb2d2fd1d65:/# cat newfile
Sunderland is the greatest football team in the world

```

As if by magic, the file you created is still there and the contents are exactly how you left it. This proves that stopping a container does not destroy the container or the data inside of it.

While this example illustrates the persistent nature of containers, it's important you understand two things:

1. The data created in this example is stored on the Docker hosts local filesystem. If the Docker host fails, the data will be lost.
2. Containers are designed to be immutable objects and it's not a good practice to write data to them.

For these reasons, Docker provides *volumes*. These exist outside of containers but can be mounted into them.

At this stage of your journey, this was an effective example of a container lifecycle, and you'd be hard pressed to draw a major difference between the lifecycle of a container and a VM.

Now let's kill the container and delete it from the system.

You can forcibly delete a *running* container with a single command. However, it might be best to stop it first, giving the application a chance to stop gracefully. More on this in a second.

The next example will stop the percy container, delete it, and verify the operation. If your terminal is still attached to the percy container, you'll need to press `Ctrl-PQ` to gracefully disconnect.

```

$ docker stop percy
percy

$ docker rm percy
percy

$ docker ps -a
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS        NAMES

```

The container is now deleted — literally wiped off the face of the planet. If it was a good container, it becomes a *WebAssembly app* in the next life. If it was a naughty container, it becomes a dumb terminal :-D

Summarizing the lifecycle of a container. You can stop, start, pause, and restart a container as many times as you want. It's not until you explicitly delete it that you run a chance of losing its data. Even then, if you're storing data outside the container in a *volume*, the data will persist even after the container has gone.

Let's quickly mention why we recommended a two-stage approach of stopping the container before deleting it.

Stopping containers gracefully

In the previous example, the container was running the `/bin/bash` app. When you kill a running container with `docker rm <container> -f`, the container is killed immediately without warning. You're literally giving the container, and the app it's running, no chance to complete any operations and gracefully exit.

However, the `docker stop` command is far more polite. It gives the process inside of the container ~10 seconds to complete any final tasks and gracefully shutdown. Once the command completes, you can delete the container with `docker rm`.

Behind the scenes the `docker stop` command sends a **SIGTERM** signal to the main application process inside the container (PID 1). This is a request to terminate and gives the process a chance to clean things up and gracefully shut itself down. If it's still running after 10 seconds it will be issued a **SIGKILL** which terminates it with force.

A `docker rm <container> -f` doesn't bother asking nicely with a **SIGTERM**, it goes straight to the **SIGKILL**.

Self-healing containers with restart policies

It's often a good idea to run containers with a *restart policy*. This is a very basic form of self-healing that allows the local Docker engine to automatically restart failed containers.

Restart policies are applied per-container. They can be configured imperatively on the command line as part of `docker run` commands, or declaratively in YAML files for use with higher-level tools such as Docker Swarm, Docker Compose, and Kubernetes.

At the time of writing, the following restart policies exist:

- `always`
- `unless-stopped`
- `on-failure`

The **always** policy is the simplest. It always restarts a failed container unless it's been explicitly stopped. An easy way to demonstrate this is to start a new interactive container with the `--restart always` policy and tell it to run a shell process. When the container starts you'll be automatically attached to its shell. Typing `exit` from the shell will kill the container's PID 1 process and kill the container. However, Docker will automatically restart it because it has the `--restart always` policy. If you issue a `docker ps` command, you'll see that the container's uptime is less than the time since it was created. Let's put it to the test.

```
$ docker run --name neversaydie -it --restart always alpine sh
/#
```

Wait a few seconds before typing the `exit` command.

Once you've exited the container and are back at your normal shell prompt, check the container's status.

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	NAME
0901afb84439	alpine	"sh"	35 seconds ago	Up 9 seconds	neversaydie

See how the container was created 35 seconds ago but has only been up for 9 seconds. This is because the `exit` command killed it and Docker restarted it. Be aware that Docker has restarted the same container and not created a new one. In fact, if you inspect it with `docker inspect` you can see the `restartCount` has been incremented.

An interesting feature of the `--restart always` policy is that if you stop a container with `docker stop` and then restart the Docker daemon, the container will be restarted. To be clear... you start a new container with the `--restart always` policy and then intentionally stop it with the `docker stop` command. At this point the container is in the `Stopped (Exited)` state. However, if you restart the Docker daemon, the container will be automatically restarted when the daemon comes back up. You need to be aware of this.

The main difference between the **always** and **unless-stopped** policies is that containers with the `--restart unless-stopped` policy will not be restarted when the daemon restarts if they were in the `Stopped (Exited)` state. That might be a confusing sentence, so let's walk through an example.

We'll create two new containers. One called "always" with the `--restart always` policy, and one called "unless-stopped" with the `--restart unless-stopped` policy. We'll stop them both with the `docker stop` command and then restart Docker. The "always" container will restart, but the "unless-stopped" container will not.

1. Create the two new containers

```
$ docker run -d --name always \
  --restart always \
  alpine sleep 1d

$ docker run -d --name unless-stopped \
  --restart unless-stopped \
  alpine sleep 1d

$ docker ps
CONTAINER ID   IMAGE     COMMAND                  STATUS              NAMES
3142bd91ecc4   alpine    "sleep 1d"              Up 2 secs           unless-stopped
4f1b431ac729   alpine    "sleep 1d"              Up 17 secs          always
```

We now have two containers running. One called “always” and one called “unless-stopped”.

2. Stop both containers

```
$ docker stop always unless-stopped

$ docker ps -a
CONTAINER ID   IMAGE     STATUS              NAMES
3142bd91ecc4   alpine    Exited (137) 3 seconds ago  unless-stopped
4f1b431ac729   alpine    Exited (137) 3 seconds ago  always
```

3. Restart Docker.

The process for restarting Docker is different on different Operating Systems. This example shows how to stop Docker on Linux hosts running systemd. If asked for a password, just re-run the command with `sudo` in front of it.

```
$ systemctl restart docker
```

4. Once Docker has restarted, you can check the status of the containers.

```
$ docker ps -a
CONTAINER   CREATED          STATUS              NAMES
314..cc4    2 minutes ago    Exited (137) 2 minutes ago  unless-stopped
4f1..729    2 minutes ago    Up 9 seconds         always
```

Notice that the “always” container (started with the `--restart always` policy) has been restarted, but the “unless-stopped” container (started with the `--restart unless-stopped` policy) has not.

The **on-failure** policy will restart a container if it exits with a non-zero exit code. It will also restart containers when the Docker daemon restarts, even ones that were in the stopped state.

If you are working with Docker Compose or Docker Stacks, you can apply the restart policy to a service object as follows. We’ll talk more about these technologies later in the book.

```
services:
  myservice:
    <Snip>
    restart_policy:
      condition: always | unless-stopped | on-failure
```

Web server example

So far, we've seen how to start a simple container and interact with it. We've also seen how to stop, restart and delete containers. Now let's take a look at a Linux-based web server example.

In this example, we'll start a new container from an image that contains a simple nodejs app running on port 8080.

Run the `docker stop` and `docker rm` commands to clean up any containers from previous examples.

Run the following command to start a new web server container.

```
$ docker run -d --name webserver -p 80:8080 \
  nigelpoulton/ddd-book:web0.1
```

Notice that your shell prompt hasn't changed. This is because the container was started in the background with the `-d` flag. Starting a container like this doesn't attach it to your terminal.

Let's take a look at some of the other arguments in the command.

We know `docker run` starts a new container. However, this time we give it the `-d` flag instead of `-it`. `-d` stands for **d**etached or **d**aemon mode and tells the container to run in the background. You can't use the `-d` and `-it` flags in the same command.

After that, the command names the container "webserver". The `-p` flag maps port 80 on the Docker host to port 8080 inside the container. This means that traffic hitting the Docker host on port 80 will be directed to port 8080 inside of the container. The image we're using for this container contains a web service that listens on port 8080. This means the container will come up running a web server listening on port 8080.

Finally, the command tells the container to base itself on the `nigelpoulton/ddd-book:web0.1` image. The image contains a node.js webserver and all dependencies. It is maintained approximately once per year, so will contain vulnerabilities!

Once the container is running, a `docker ps` command will show the container as running and the ports that are mapped. It's important to know that port mappings are expressed as `host-port:container-port`.


```
$ docker ps
```

CONTAINER ID	COMMAND	STATUS	PORTS	NAMES
b92d95e0b95b	"node ./app.js"	Up 2 mins	0.0.0.0:80->8080/tcp	webserver

Some of the columns have been removed from the output to help with readability.

Now that the container is running and ports are mapped, you can connect to it by pointing a web browser at the IP address or DNS name of your **Docker host** on port 80. If you're running Docker locally using Docker Desktop you can connect to `localhost:80` or `127.0.0.1:80`.

Figure 7.4 shows the web page that is being served up by the container.

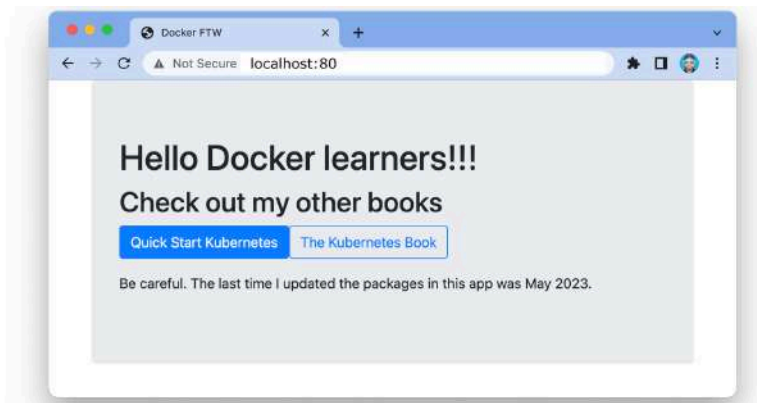


Figure 7.4

The same `docker stop`, `docker pause`, `docker start`, and `docker rm` commands can be used on the container.

Inspecting containers

In the previous web server example, you might have noticed that we didn't specify an app for the container when we issued the `docker run` command. Yet the container ran a web service. How did this happen?

When building a Docker image, you can embed an instruction that lists the default app for any containers that use the image. You can see this for any image by running a `docker inspect`.

```
$ docker inspect nigelpoulton/ddd-book:web0.1

[
  {
    "Id": "sha256:4b4292644137e5de...fc6d0835089b",
    "RepoTags": [
      "nigelpoulton/ddd-book:web0.1"
    ],
    <Snip>
    ],
    "Entrypoint": [
      "node",
      "./app.js"
    ],
    <Snip>
  ]
]
```

The output is snipped to make it easier to find the information we're interested in.

The entries after `Cmd` or `Entrypoint` show the app that the container will run unless you override it with a different one when you launch it with `docker run`.

It's common to build images with default commands like this as it makes starting containers easier. It also forces a default behavior and is a form of self documentation — i.e. you can *inspect* the image and know what app it's designed to run.

That's us done for the examples in this chapter. Let's see a quick way to tidy our system up.

Tidying up

Let's look at the simplest and quickest way to get rid of **every running container** on a Docker host. Be warned though, this procedure will forcibly destroy **all** containers without giving them a chance to clean up. **This should never be performed on production systems or systems running important containers.**

Run the following command from the shell of your Docker host to delete **all** containers. It will delete **all containers** without warning.

```
$ docker rmi $(docker ps -aq) -f
b92d95e0b95b
```

This example only had a single container running, so only one was deleted. However, the command works the same way as the `docker rmi $(docker images -q)` command we used in the previous chapter to delete all images on a single Docker host. We already know the `docker container rm` command deletes containers. Passing it `$(docker ps`

-aq) as an argument, effectively passes it the ID of every container on the system. The -f flag forces the operation so that even containers in the running state will be destroyed. Net result... all containers, running or stopped, will be destroyed and removed from the system.

Containers - The commands

- `docker run` is the command used to start new containers. In its simplest form, it accepts an *image* and a *command* as arguments. The image is used to create the container and the command is the application the container will run when it starts. This example will start an Ubuntu container in the foreground and tell it to run the Bash shell: `docker run -it ubuntu /bin/bash`.
- `Ctrl-PQ` will detach your shell from the terminal of a container and leave the container running in the background.
- `docker ps` lists all containers in the running state. If you add the -a flag you will also see containers in the stopped (Exited) state.
- `docker exec` runs a new process inside of a running container. It's useful for attaching the shell of your Docker host to a terminal inside a running container. This command will start a new Bash shell inside a running container and connect to it: `docker exec -it <container-name or container-id> bash`. For this to work, the image used to create the container must include the Bash shell.
- `docker stop` will stop a running container and put it in the Exited (0) state. It issues a SIGTERM to the process with PID 1 inside of the container. If the process hasn't cleaned up and stopped within 10 seconds it will send a SIGKILL to forcibly stop the container. `docker stop` accepts container IDs and container names as arguments.
- `docker start` will restart a stopped container. You can give it the name or ID of a container.
- `docker rm` will delete a stopped container. You can specify containers by name or ID. It is recommended that you stop a container with the `docker stop` command before deleting it with `docker rm`.
- `docker inspect` will show you detailed configuration and runtime information about a container. It accepts container names and container IDs as its main argument.

Chapter summary

In this chapter, we compared and contrasted the container and VM models. We looked at the *OS tax* problem inherent in the VM model and saw how the container model can bring huge advantages.

We saw how to use the `docker run` command to start a couple of simple containers, and we saw the difference between interactive containers in the foreground and daemon containers running in the background.

We know that killing the PID 1 process inside of a container will kill the container, and we've seen how to start, stop, and delete containers.

We finished the chapter using the `docker inspect` command to view detailed container metadata.

8: Containerizing an app

Docker is all about making it easy to take application source code and get it running in a container. This process is called *containerization*.

In this chapter, we'll walk through the process of containerizing some simple Linux apps. You'll need a Docker environment if you want to follow along. Any of the environments from the Getting Docker chapter will work.

We'll split this chapter into the usual three parts:

- The TLDR
- The deep dive
- The commands

Let's containerize an app!

Containerizing an app - The TLDR

Containers are all about making apps simple to **build**, **ship**, and **run**. The end-to-end process looks like this:

1. Start with your application code and dependencies
2. Create a *Dockerfile* that describes your app, dependencies, and how to run it
3. Build it into an image by passing the *Dockerfile* to the `docker build` command
4. Push the new image to a registry (optional)
5. Run a container from the image

Figure 8.1 shows the process in picture form.

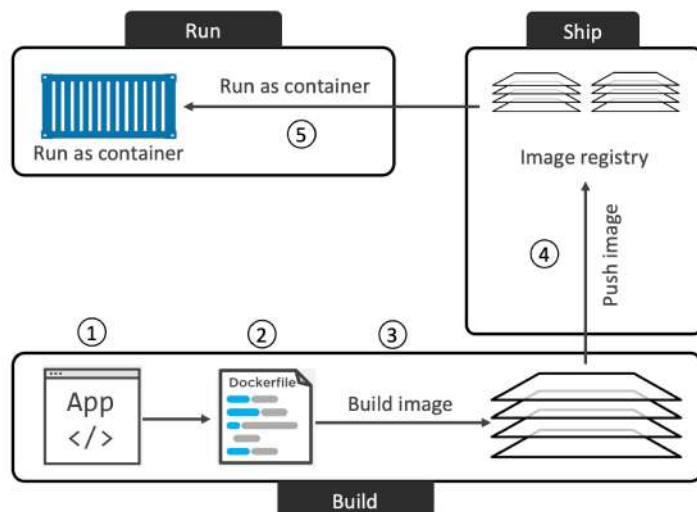


Figure 8.1 - Basic flow of containerizing an app

Containerizing an app - The deep dive

We'll break up this Deep Dive section as follows:

- Containerize a single-container app
- Moving to Production with multi-stage builds
- Multi-platform builds
- A few best practices

Containerize a single-container app

This section of the chapter walks through the process of containerizing a simple Node.js app.

We'll complete the following high-level steps:

- Clone the repo to get the app code
- Inspect the Dockerfile
- Containerize the app
- Run the app

- Test the app
- Look a bit closer

Getting the application code

The application used in this example is available from the book's GitHub repo at:

- <https://github.com/nigelpoulton/ddd-book>

Run the following command to clone the repo. You'll need `git` installed to complete this step.

```
$ git clone https://github.com/nigelpoulton/ddd-book.git

Cloning into 'ddd-book'...
remote: Enumerating objects: 47, done.
remote: Counting objects: 100% (47/47), done.
remote: Compressing objects: 100% (32/32), done.
remote: Total 47 (delta 11), reused 44 (delta 11), pack-reused 0
Receiving objects: 100% (47/47), 167.30 KiB | 1.66 MiB/s, done.
Resolving deltas: 100% (11/11), done.
```

The clone operation creates a new directory called `ddd-book` in your working directory. Change directory into `ddd-book/web-app` and list its contents.

```
$ cd ddd-book/web-app

$ ls -l
total 20
-rw-rw-r-- 1 ubuntu ubuntu 324 May 20 07:44 Dockerfile
-rw-rw-r-- 1 ubuntu ubuntu 377 May 20 07:44 README.md
-rw-rw-r-- 1 ubuntu ubuntu 341 May 20 07:44 app.js
-rw-rw-r-- 1 ubuntu ubuntu 404 May 20 07:44 package.json
drwxrwxr-x 2 ubuntu ubuntu 4096 May 20 07:44 views
```

This directory is called the *build context* and contains all of the application source code, as well as file containing a list of dependencies. It's also a common practice to keep the application's `Dockerfile` in the build context.

Now that we have the app code, let's look at its `Dockerfile`.

Inspecting the Dockerfile

A **Dockerfile** describes an application and tells Docker how to build it into an image.

Do not underestimate the impact of the Dockerfile as a form of documentation. It's a great document for bridging the gap between developers and operations. It also has the power to speed up on-boarding of new team members. This is because the file accurately describes the application and its dependencies in an easy-to-read format. You should treat it like you treat source code and keep it in a version control system.

Let's look at the contents of this application's Dockerfile.

```
$ cat Dockerfile
```

```
FROM alpine
LABEL maintainer="nigelpoulton@hotmail.com"
RUN apk add --update nodejs npm
COPY . /src
WORKDIR /src
RUN npm install
EXPOSE 8080
ENTRYPOINT ["node", "./app.js"]
```

At a high-level, the example Dockerfile says: Start with the `alpine` image, make a note that “nigelpoulton@hotmail.com” is the maintainer, install Node.js and NPM, copy everything in the build context to the `/src` directory in the image, set the working directory as `/src`, install dependencies, document the app's network port, and set `app.js` as the default application to run.

Let's look at it in a bit more detail.

Dockerfiles normally start with the `FROM` instruction. This pulls an image that will be used as the *base layer* for the image the Dockerfile will build – everything else will be added as new layers above this base layer. The app being defined in this Dockerfile is a Linux app, so it's important that the `FROM` instruction refers to a Linux-based image. If you're containerizing a Windows application, you'll need to specify an appropriate Windows base image.

At this point in the Dockerfile, the image has a single layer as shown in Figure 8.2.

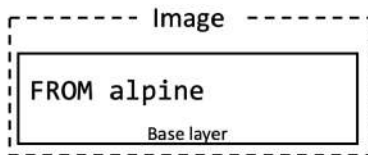


Figure 8.2

Next, the Dockerfile creates a LABEL that specifies “nigelpoulton@hotmail.com” as the maintainer of the image. Labels are optional key-value pairs and are a good way of adding custom metadata. It’s considered a best practice to list a maintainer so that other users have a point of contact to report problems etc.

The `RUN apk add --update nodejs nodejs-npm` instruction uses the apk package manager to install nodejs and nodejs-npm into the image. It does this by adding a new layer and installing the packages to this layer. At this point in the Dockerfile, the image looks like Figure 8.3.

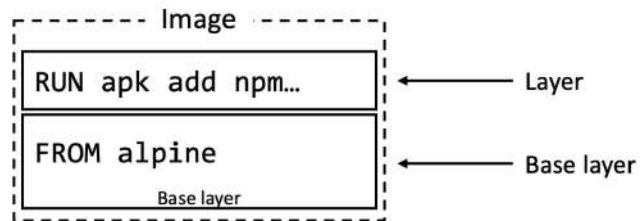


Figure 8.3

The `COPY . /src` instruction creates another new layer and copies in the application and dependency files from the *build context*. Now the image has three layers as shown in Figure 8.4.

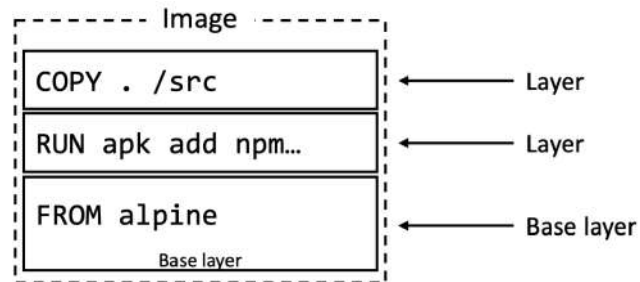


Figure 8.4

Next, the Dockerfile uses the `WORKDIR` instruction to set the working directory for the rest of the instructions. This creates metadata and does not create a new image layer.

The `RUN npm install` instruction runs within the context of the `WORKDIR` set in the previous instruction, and installs the dependencies listed in `package.json` into another new layer. At this point in the Dockerfile the image has four layers as shown in Figure 8.5.

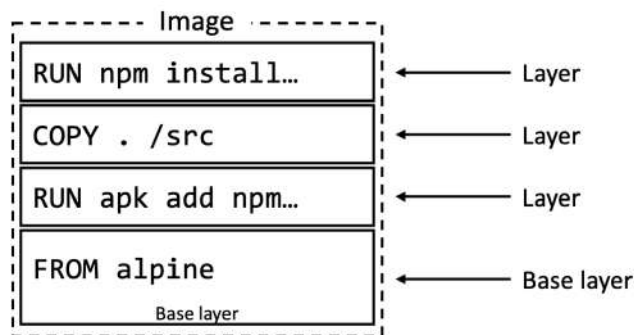


Figure 8.5

The application exposes a web service on port 8080, so the Dockerfile documents this with the `EXPOSE 8080` instruction. Finally, the `ENTRYPOINT` instruction sets the application to run when started as a container. Both of these are added as metadata and do not create new layers.

Containerize the app/build the image

Now that we understand the theory, let's see it in action.

The following command will build a new image called `ddd-book:ch8.1`. The period `(.)` at the end of the command tells Docker to use the working directory as the *build context*. Remember, the *build context* is where the app and all dependencies are stored.

Be sure to include the trailing period `(.)` and be sure to run the command from the web-app directory.

```
$ docker build -t ddd-book:ch8.1 .
```

```
[+] Building 16.2s (10/10) FINISHED
=> [internal] load build definition from Dockerfile           0.0s
=> => transferring dockerfile: 335B                          0.0s
=> => transferring context: 2B                                0.0s
=> [1/5] FROM docker.io/library/alpine                       0.1s
=> CACHED [2/5] RUN apk add --update nodejs npm curl         0.0s
=> [3/5] COPY . /src                                          0.0s
=> [4/5] WORKDIR /src                                         0.0s
=> [5/5] RUN npm install                                     10.4s
=> exporting to image                                         0.2s
=> => exporting layers                                         0.2s
=> => writing image sha256:f282569b8bd0f0...016cc1adafc91    0.0s
=> => naming to docker.io/library/ddd-book:ch8.1
```

Notice the five numbered steps reported in the build output. Those are the steps that create the five image layers.

Check that the image exists in your Docker host's local repository.

```
$ docker images
REPO          TAG          IMAGE ID          CREATED          SIZE
ddd-book      ch8.1        f282569b8bd0     4 minutes ago   95.4MB
```

Congratulations, the app is containerized!

You can use the `docker inspect ddd-book:ch8.1` command to verify the configuration of the image. It will list all of the settings that were configured from the Dockerfile. Look out for the list of image layers and the Entrypoint command.

```
$ docker inspect ddd-book:ch8.1
[
  {
    "Id": "sha256:f282569b8bd0...016cc1adafc91",
    "RepoTags": [
      "ddd-book:ch8.1"
    ],
    "WorkingDir": "/",
    "Entrypoint": [
      "node",
      "./app.js"
    ],
    "Labels": {
      "maintainer": "nigelpoulton@hotmail.com"
    },
    "Layers": [
      "sha256:94dd7d531fa5695c0c033dcb69f213c2b4c3b5a3ae6e497252ba88da87169c3f",
      "sha256:a990a785ba64395c8b9d05fbc32176d1fb3edd94f6fe128ed7415fd7e0bb4231",
      "sha256:efeb99f5a1b27e36bc6c46ea9eb2ba4aab942b47547df20ee8297d3184241b1d",
      "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef",
      "sha256:ccf07adfaecfba485ecd7274c092e7343c45e539fa4371c5325e664122c7c92b"
    ]
  }
]
```

Pushing images

Once you've created an image, it's a good idea to store it in a registry to keep it safe and make it available to others. Docker Hub is the most common public image registry and it's the default push location for `docker push` commands.

You'll need a Docker ID and if you want to push the image to Docker Hub. You'll also need to tag the image appropriately.

If you don't already have a Docker Hub ID, go to `hub.docker.com` and sign-up for one now, they're free.

Be sure to substitute my Docker ID with your own in the examples. So, any time you see **nigelpoulton**, swap it out for your Docker ID (Docker Hub username).

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub.
Username: nigelpoulton
Password:
WARNING! Your password will be stored unencrypted in /home/ubuntu/.docker/config.json.
Configure a credential helper to remove this warning.
```

Images need to be appropriately tagged before you can push them. This is because the tag includes the following important registry-related information:

- Registry DNS name
- Repository name
- Tag

Docker is opinionated about registries – it assumes you want to push to Docker Hub. You can push to other registries by adding the registry URL to the beginning of the image tag.

The previous `docker images` output shows the image is tagged as `ddd-book:ch8.1`. `docker push` will try and push this to a repository called `ddd-book` on Docker Hub. However, this repository doesn't exist and I wouldn't have access to it anyway, as all of my repositories exist in the `nigelpoulton` second-level namespace. This means I need to re-tag the image to include my Docker ID. Remember to substitute your own Docker ID.

The format of the command is `docker tag <current-tag> <new-tag>`. This adds an additional tag and does not overwrite the original.

```
$ docker tag ddd-book:ch8.1 nigelpoulton/ddd-book:ch8.1
```

Running another `docker images` shows the image now has two tags.

```
$ docker images
```

REPO	TAG	IMAGE ID	CREATED	SIZE
ddd-book	ch8.1	f282569b8bd0	13 mins ago	95.4MB
nigelpoulton/ddd-book	ch8.1	f282569b8bd0	13 mins ago	95.4MB

Now we can push it to Docker Hub. Be sure to substitute your Docker ID.

```
$ docker push nigelpoulton/ddd-book:ch8.1
The push refers to repository [docker.io/nigelpoulton/ddd-book]
ccf07adfaecf: Pushed
5f70bf18a086: Layer already exists
efeb99f5a1b2: Pushed
a990a785ba64: Pushed
94dd7d531fa5: Layer already exists
ch8.1: digest: sha256:80063789bce73a17...09ea29c5e6a91c28b4 size: 1365
```

Figure 8.6 shows how Docker worked out the push location.



Figure 8.6

Now that the image is pushed to a registry, you can access it from anywhere with an internet connection. You can also grant other people access to pull it and push changes.

Run the app

The containerized application is a web server that listens on port 8080. You can verify this in the `app.js` file in the build context you cloned from GitHub.

The following command will start a new container called `c1` based on the `ddd-book:ch8.1` image you just created. It maps port 80 on the Docker host, to port 8080 inside the container. This means you'll be able to point a web browser at the DNS name or IP address of the Docker host running the container and access the app.

Note: If your host is already running a service on port 80 you'll get a **port is already allocated** error. If this happens, specify a different port such as 5000 or 5001. For example, to map the app to port 5000 on the Docker host, use the `-p 5000:8080` flag.

```
$ docker run -d --name c1 \  
-p 80:8080 \  
ddd-book:ch8.1
```

The `-d` flag runs the container in the background, and the `-p 80:8080` flag maps port 80 on the host to port 8080 inside the running container.

Check that the container is running and verify the port mapping.

```
$ docker ps
```

ID	IMAGE	COMMAND	STATUS	PORTS	NAMES
49..	ddd-book:ch8.1	"node ./app.js"	UP 18 secs	0.0.0.0:80->8080/tcp	c1

The output above is snipped for readability but shows that the container is running. Note that port 80 is mapped on all host interfaces (`0.0.0.0:80`).

Test the app

Open a web browser and point it to the DNS name or IP address of the host that the container is running on. If you're using Docker Desktop or another technology that runs the container on your local machine, you can use `localhost` as the DNS name. Otherwise, use the IP or DNS of the Docker host.

You'll see the web page shown in Figure 8.7.



Figure 8.7

If the test doesn't work, try the following:

1. Make sure the container is up and running with the `docker ps` command. The container name is `c1` and you should see the port mapping as `0.0.0.0:80->8080/tcp`.

2. Check that firewall and other network security settings aren't blocking traffic to port 80 on the Docker host.
3. Retry the `docker run` command specifying a high numbered port on the Docker host such as `-p 5001:8080`.

Congratulations, the application is containerized and running as a container!

Looking a bit closer

Now that the application is containerized, let's take a closer look at how some of the machinery works.

The `docker build` command parses the Dockerfile one-line-at-a-time starting from the top.

Comment lines start with the `#` character.

All non-comment lines are **Instructions** and take the format `<INSTRUCTION> <arguments>`. Instruction names are not case sensitive but it's normal practice to write them in UPPERCASE to make reading the file easier.

Some instructions create new layers whereas others just add metadata.

Examples of instructions that create new layers are `FROM`, `RUN`, and `COPY`. Examples that create metadata include `EXPOSE`, `WORKDIR`, `ENV`, and `ENTRYPOINT`. The basic premise is this — if an instruction adds *content* such as files and programs, it will create a new layer. If it is adding instructions on how to build the image and run the container, it will create metadata.

You can view the instructions that were used to build the image with the `docker history` command.

```
$ docker history ddd-book:ch8.1
```

IMAGE	CREATED BY	SIZE
f282569b8bd0	ENTRYPOINT ["node" "./app.js"]	0B
<missing>	EXPOSE map[8080/tcp:{}]	0B
<missing>	RUN /bin/sh -c npm install	24.2MB
<missing>	WORKDIR /src	0B
<missing>	COPY . /src #	8.41kB
<missing>	RUN /bin/sh -c apk add --update nodejs npm	63.4MB
<missing>	LABEL maintainer=nigelpoulton@hotmail.com	0B
<missing>	/bin/sh -c #(nop) CMD ["/bin/sh"]	0B
<missing>	/bin/sh -c #(nop) ADD file:df7fccc3453b6ec1	7.73MB

Two things from the output are worth noting.

First. Each line corresponds to an instruction in the Dockerfile (starting from the bottom and working up). The `CREATED BY` column even lists the exact Dockerfile instruction that was executed.

Second. Only 4 of the lines displayed in the output create new layers (the ones with non-zero values in the `SIZE` column). These correspond to the `FROM`, `RUN`, and `COPY` instructions in the Dockerfile. The other instructions create metadata instead of layers.

Use the `docker inspect` command to see the list of image layers.

```
$ docker inspect ddd-book:ch8.1
```

```
<Snip>
},
"RootFS": {
  "Type": "layers",
  "Layers": [
    "sha256:94dd7d531fa5695c0c033dcb69f213c2b4c3b5a3ae6e497252ba88da87169c3f",
    "sha256:a990a785ba64395c8b9d05f32176d1fb3edd94f6fe128ed7415fd7e0bb4231",
    "sha256:efeb99f5a1b27e36bc6c46ea9eb2ba4aab942b47547df20ee8297d3184241b1d",
    "sha256:ccf07adfaecfba485ecd7274c092e7343c45e539fa4371c5325e664122c7c92b"
  ]
},
```

Figure 8.8 maps the Dockerfile instructions to image layers. The layer IDs will be different in your environment. The Dockerfile instructions with arrows from them create layers, the others don't.

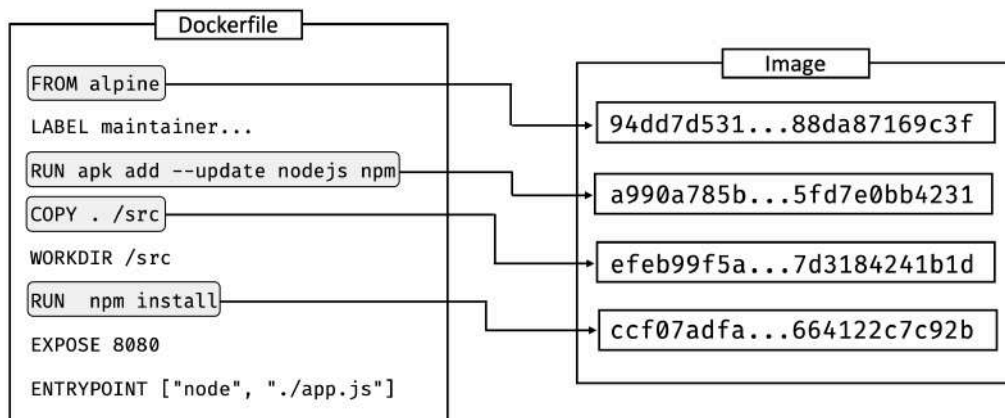


Figure 8.8

Note: There may be a bug in the builder used by Docker that causes the `WORKDIR` instruction to create a layer. This may cause your environment to show more layers than expected.

It's considered a good practice to use images from official repositories as the base layer for new images. This is because their content is vetted and they are quick to release new versions when vulnerabilities are fixed.

Moving to production with Multi-stage Builds

When it comes to Docker images, *big is bad!* For example:

- Big means slow
- Big means more potential vulnerabilities
- Big means a bigger attack surface

For these reasons, container images should only contain the stuff **needed** to run your app in production.

In the past, keeping images small was hard work. However, multi-stage builds make it easy. Here's the high-level...

Multi-stage builds have multiple `FROM` instructions in a single Dockerfile, and each `FROM` instruction is a new **build stage**. You can do the heavy-lifting work of building your app in a large image with all the compilers and other build tools required. You can then copy the final production app into a tiny image used for production. You can also perform build steps in parallel for faster builds.

A high-level flow is shown in Figure 8.9. Stage 1 build an image with all the build and compilation tools you need. Stage 2 copies in your app code and builds it. Stage 3 creates a small production-ready image with just the bits required to run the app.

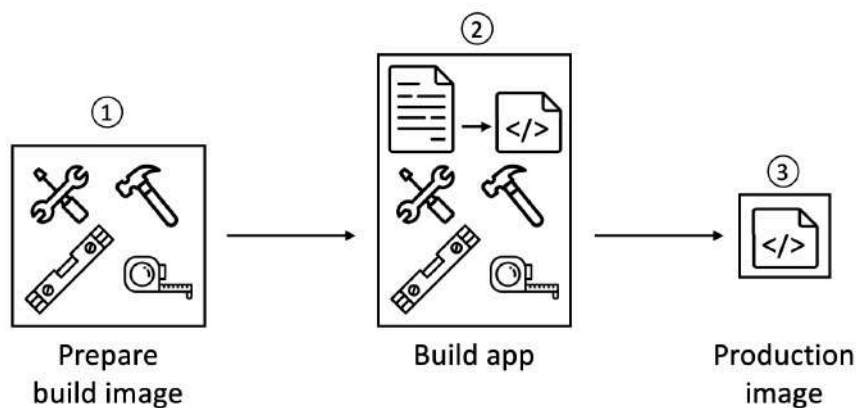


Figure 8.9

Let's look at an example!

All the code for the example is in the `multi-stage` folder of the book's GitHub repo. The example is a simple Go app with a client and server and is borrowed from the *Docker samples buildme* repo on GitHub. Don't worry if you're not a Go programmer, we'll only focus on the Dockerfile. The only thing you need to know is that it builds the *client* and *server* apps into executable files that do not need the Go language or any other tools or runtime in order to execute.

The Dockerfile is shown below:

```
FROM golang:1.20-alpine AS base
WORKDIR /src
COPY go.mod go.sum .
RUN go mod download
COPY . .

FROM base AS build-client
RUN go build -o /bin/client ./cmd/client

FROM base AS build-server
RUN go build -o /bin/server ./cmd/server

FROM scratch AS prod
COPY --from=build-client /bin/client /bin/
COPY --from=build-server /bin/server /bin/
ENTRYPOINT [ "/bin/server" ]
```

The first thing to note is that the Dockerfile has **four** FROM instructions. Each of these is a distinct **build stage** and Docker numbers them starting from 0. However, each stage has been given a friendly name.

- Stage 0 is called `base`
- Stage 1 is called `build-client`
- Stage 2 is called `build-server`
- Stage 3 is called `prod`

Each stage outputs an image that can be used by other stages. These *intermediate images* are cleaned up when the final build completes.

The goal of the base stage is to create a reusable build image with all the tools needed to build the application in the later stages. The image created by this stage will only be used to build the app and not used for production.

It pulls the `golang:1.20-alpine` image which is over 250MB when uncompressed on the host. It sets the working directory to `/src` and copies in the `go.mod` and `go.sum` files.

These list the application dependencies and hashes. Next, it installs the dependencies and copies the app code into the image. This stage will add three new layers containing a lot of build stuff but not very much app code. When this build stage is complete it will output a large image that can be used by later stages.

The `build-client` stage doesn't pull a new image. Instead, it uses the `FROM base AS build-client` instruction to use the intermediate image created by the base stage. It then uses a `RUN` instruction to build the client app into a binary executable. The goal of this stage is to create an image with the compiled client binary that can be referenced by later build stages.

The `build-server` stage does the same for the server component of the app and outputs an image that can be referenced by later stages as `build-server`.

The `build-client` and `build-server` stages can run in parallel, speeding up your build process.

The `prod` stage pulls the minimal scratch image. It then uses the `COPY --from` instruction to copy the compiled client app from the `build-client` stage and the compiled server app from the `build-server` stage. It outputs the final image which is just the client and server apps in a tiny scratch image.

Let's see it in action.

Change into the `multi-stage` directory of the repo and verify the Dockerfile exists.

```
$ ls -l

total 28
-rw-rw-r-- 1 ubuntu ubuntu 368 May 21 10:09 Dockerfile
-rw-rw-r-- 1 ubuntu ubuntu 433 May 21 10:09 Dockerfile-final
-rw-rw-r-- 1 ubuntu ubuntu 305 May 21 10:09 README.md
drwxrwxr-x 4 ubuntu ubuntu 4096 May 21 10:09 cmd
-rw-rw-r-- 1 ubuntu ubuntu 1013 May 21 10:09 go.mod
-rw-rw-r-- 1 ubuntu ubuntu 5631 May 21 10:09 go.sum
```

Perform the build. Watch and see the `build-client` and `build-server` stages execute in parallel. This makes large builds faster.

```
$ docker build -t multi:stage .
```

```
[+] Building 18.6s (14/14) FINISHED
=> [internal] load build definition from Dockerfile          0.0s
=> => transferring dockerfile: 407B                        0.0s
<Snip>
=> [build-client 1/1] RUN go build -o /bin/client ./cmd/client 15.8s
=> [build-server 1/1] RUN go build -o /bin/server ./cmd/server 14.8s
<Snip>
```

Run a `docker images` to see the new image.

```
$ docker images
```

REPO	TAG	IMAGE ID	CREATED	SIZE
multi	stage	638e639de548	3 minutes ago	15MB

The final production image is only 15MB. This is a lot smaller than the 250MB base image that was pulled to create the build. This is because the final stage of the multi-stage build used the tiny scratch image and only added the compiled client and server binaries.

The following `docker history` command shows the final production image with just two layers – one copying in the client binary and the other copying in the server binary. None of the previous build stages are included in this final production image.

```
$ docker history multi:stage
```

IMAGE	CREATED	CREATED BY	SIZE
638e639de548	6 minutes ago	ENTRYPOINT ["/bin/server"]	0B
<missing>	6 minutes ago	COPY /bin/server /bin/ # buildkit	7.46MB
<missing>	6 minutes ago	COPY /bin/client /bin/ # buildkit	7.58MB

Multi-stage builds and build targets

It's also possible to build multiple images from a single Dockerfile.

In our example, we might want to create separate images for the client and server binaries. We can do this by splitting the final prod stage in the Dockerfile into two stages as follows. This is in the repo as `Dockerfile-final`.

```

FROM golang:1.20-alpine AS base
WORKDIR /src
COPY go.mod go.sum .
RUN go mod download
COPY . .

FROM base AS build-client
RUN go build -o /bin/client ./cmd/client

FROM base AS build-server
RUN go build -o /bin/server ./cmd/server

FROM scratch AS prod-client
COPY --from=build-client /bin/client /bin/
ENTRYPOINT [ "/bin/client" ]

FROM scratch AS prod-server
COPY --from=build-server /bin/server /bin/
ENTRYPOINT [ "/bin/server" ]

```

The only change is the last two build stages that used to be a single stage called `prod`.

We can reference these stage names in two `docker build` commands as follows. The commands `-f` flag to reference the Dockerfile called `Dockerfile-final` that has the two separate `prod` stages.

```
$ docker build -t multi:client --target prod-client -f Dockerfile-final .
<Snip>
```

```
$ docker build -t multi:server --target prod-server -f Dockerfile-final .
<Snip>
```

Check the builds and images sizes.

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
multi	client	0d318210282f	23 minutes ago	7.58MB
multi	server	f1dbe58b5dbe	39 minutes ago	7.46MB
multi	stage	638e639de548	23 minutes ago	15MB

All three images are present. The `client` and `server` images are each about half the size of the stage image. This is because the stage image contains the client and server binaries.

Multi-platform builds

The `docker build` command lets you build images for multiple different platforms with a single command. As a quick example, I build all of the images for this book on my M1

Mac with an ARM chip. However, I use multi-platform builds to build images for other platforms such as AMD64. This way, you can use the images in the book whether you're on ARM or AMD (x64).

You'll need to use the `docker buildx` command to perform multi-platform builds. Fortunately it ships with Docker Desktop and a lot of modern Docker engine installations.

The following steps will configure `docker buildx` and walk you through a multi-platform build. They have been tested on Docker Desktop on an M1 Mac.

Check you have Buildx installed.

```
$ docker buildx version
github.com/docker/buildx v0.10.4 c513d34
```

Create a builder called `docker` that uses the `docker-container` endpoint.

```
$ docker buildx create --driver=docker-container --name=container
```

Run the following command from the `web-fe` directory of the book's GitHub repo. The command builds images for the following three platforms and exports them directly to Docker Hub:

- `linux/amd64`
- `linux/arm64`
- `linux/arm/v7`

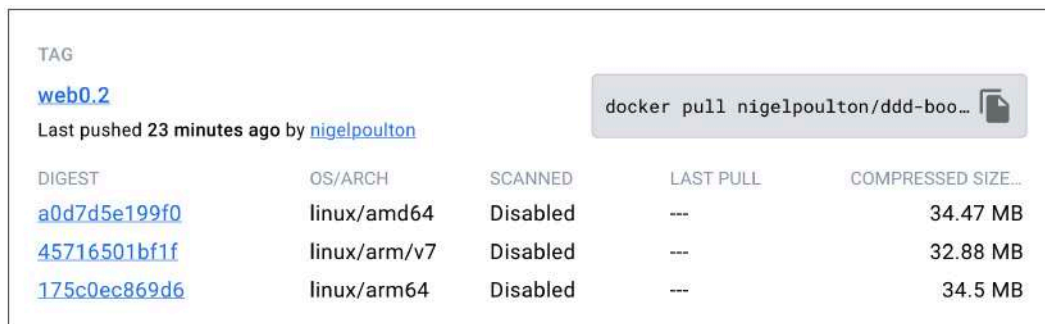
Be sure to substitute your Docker ID as the command pushes directly to Docker Hub and will fail if you try and push to my repositories.

```
$ docker buildx build --builder=container \
  --platform=linux/amd64,linux/arm64,linux/arm/v7 \
  -t nigelpoulton/ddd-book:ch8.1 --push .
```

```
[+] Building 79.3s (24/24) FINISHED
<Snip>
=> CACHED [linux/amd64 2/5] RUN apk add --update nodejs npm curl 0.0s
=> CACHED [linux/arm64 2/5] RUN apk add --update nodejs npm curl 0.0s
=> CACHED [linux/arm/v7 2/5] RUN apk add --update nodejs npm curl 0.0s
=> [linux/amd64 3/5] COPY . /src 0.0s
=> [linux/arm/v7 3/5] COPY . /src 0.0s
=> [linux/arm64 3/5] COPY . /src 0.0s
<Snip>
=> => pushing layers 31.5s
=> => pushing manifest for docker.io/nigelpoulton/ddd-book:web0.2@sha256:8fc61... 3.6s
=> [auth] nigelpoulton/ddd-book:pull,push token for registry-1.docker.io 0.0s
```

The output is snipped, but notice the two important things. All instructions from the Dockerfile are executed three times – once for each of the three target platforms. The last three lines show the image layers being pushed to Docker Hub.

Figure 8.10 shows how the three images for the three architectures appear on Docker Hub.




TAG				
web0.2				
Last pushed 23 minutes ago by nigelpoulton				
<div>docker pull nigelpoulton/ddd-boo... </div>				
DIGEST	OS/ARCH	SCANNED	LAST PULL	COMPRESSED SIZE...
a0d7d5e199f0	linux/amd64	Disabled	---	34.47 MB
45716501bf1f	linux/arm/v7	Disabled	---	32.88 MB
175c0ec869d6	linux/arm64	Disabled	---	34.5 MB

Figure 8.10 - Multi-platform image

A few best practices

Let's list a few best practices before closing out the chapter. This list is not intended to be exhaustive.

Leverage the build cache

The builder used by Docker uses a cache to speed-up the build process. The best way to see the impact of the cache is to build a new image on a clean Docker host, then repeat the same build immediately after. The first build will pull images and take time building layers. The second build will complete almost instantaneously. This is because the layers and other artefacts from the first build are cached and leveraged by later builds.

As we know, the `docker build` process iterates through a Dockerfile one-line-at-a-time starting from the top. For each instruction, Docker looks to see if it already has an image layer for that instruction in its cache. If it does, this is a *cache hit* and it uses that layer. If it doesn't, this is a *cache miss* and it builds a new layer from the instruction. Getting *cache hits* can significantly speed up the build process.

Let's look a little closer.

We'll use the following Dockerfile as an example:


```
FROM alpine
RUN apk add --update nodejs nodejs-npm
COPY . /src
WORKDIR /src
RUN npm install
EXPOSE 8080
ENTRYPOINT ["node", "./app.js"]
```

The first instruction tells Docker to use the `alpine:latest` image as its *base image*. If this image already exists on the host, the builder will move on to the next instruction. If the image doesn't exist, it gets pulled from Docker Hub.

The next instruction (`RUN apk . . .`) runs a command to update package lists and install `nodejs` and `nodejs-npm`. Before performing the instruction, Docker checks the build cache for a layer that was built from the same base image using the same instruction it's being asked to execute. In this case, it's looking for a layer that was built directly on top of `alpine:latest` by executing the `RUN apk add --update nodejs nodejs-npm` instruction.

If it finds a layer, it links to that layer and continues the build with the cache intact. If it does **not** find a layer, it invalidates the cache and builds the layer. This operation of *invalidating the cache* invalidates it for the remainder of the build. This means all subsequent Dockerfile instructions are completed in full without attempting to reference the build cache.

Let's assume that Docker already had a layer for this instruction so we had a cache hit. And let's assume the ID of that layer was AAA.

The next instruction copies some code into the image (`COPY . /src`). The previous instruction resulted in a cache hit, meaning Docker can check if it has a cached layer that was built from the AAA layer with the `COPY . /src` command. If it does, it links to the layer and proceeds to the next instruction. If it doesn't, it builds the layer and invalidates the cache for the rest of the build.

This process continues for the rest of the Dockerfile.

It's important to understand a few more things.

Firstly, as soon as any instruction results in a cache-miss (no existing layer was found for that instruction), the cache is invalidated and no longer checked for the rest of the build. This has an important impact on how you write your Dockerfiles. For example, you should try and write them in a way that places instructions that are likely to invalidate the cache towards the end of the Dockerfile. This means that a cache-miss will not occur until later stages of the build — allowing the build to benefit as much as possible from the cache.

You can force the build process to ignore the entire cache by passing the `--no-cache` flag to the `docker build` command.

It's also important to understand that the COPY and ADD instructions include steps to ensure the content being copied into the image hasn't changed since the last build. For example, it's possible that the COPY `. /src` **instruction** in the Dockerfile hasn't changed since the previous build, **but...** the contents of the directory being copied into the image **have** changed!

To protect against this, Docker performs a checksum against each file being copied. If the checksums don't match, the cache is invalidated and a new layer is built.

Squash the image

Squashing an image isn't really a best practice as it has pros and cons.

At a high level, squashing an image follows the normal build process but adds an additional step that squashes everything into a single layer. It can reduce the size of images but doesn't allow layer sharing with other images.

Just add the `--squash` flag to the `docker build` command if you want to create a squashed image.

Figure 8.11 shows some of the inefficiencies that come with squashed images. Both images are exactly the same except for the fact that one is squashed and the other is not. The non-squashed image shares layers with other images on the host (saving disk space) but the squashed image does not. The squashed image will also need to send every byte to Docker Hub on a `docker push` command, whereas the non-squashed image only needs to send unique layers.

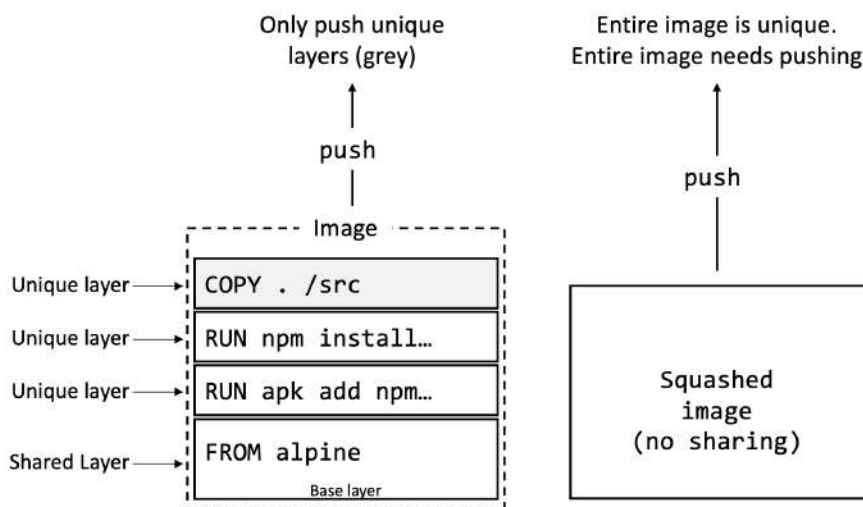


Figure 8.11 - Squashed images vs non-squashed images

Use no-install-recommends

If you're building Linux images and using the apt package manager, you should use the `no-install-recommends` flag with `apt-get install` commands. This makes sure that apt only installs main dependencies (packages in the `Depends` field) and not recommended or suggested packages. This can greatly reduce the number of unwanted packages that are downloaded into your images.

Containerizing an app - The commands

- `docker build` is the command that reads a Dockerfile and containerizes an application. The `-t` flag tags the image, and the `-f` flag lets you specify the name and location of the Dockerfile. With the `-f` flag, you can use a Dockerfile with an arbitrary name and in an arbitrary location. The *build context* is where your application files exist and can be a directory on your local Docker host or a remote Git repo.
- The Dockerfile `FROM` instruction specifies the base image for the new image you're building. It's usually the first instruction in a Dockerfile and a best-practice is to use images from official repos on this line. `FROM` is also used to distinguish a new build stage in multi-stage builds.
- The Dockerfile `RUN` instruction lets you to run commands inside the image during a build. It's commonly used to update packages and install dependencies. Each `RUN` instruction adds a new layer to the overall image.
- The Dockerfile `COPY` instruction adds files into the image as a new layer. It's common to use it to copy your application code into an image.
- The Dockerfile `EXPOSE` instruction documents the network port an application uses.
- The Dockerfile `ENTRYPOINT` instruction sets the default application to run when the image is started as a container.
- Some other Dockerfile instructions include `LABEL`, `ENV`, `ONBUILD`, `HEALTHCHECK`, `CMD` and more.

Chapter summary

This chapter taught you how to containerize an application.

We pulled some application code from a remote Git repo. The repo also included a file called `Dockerfile` containing instructions telling Docker how to build the application

into an image. We learned the basics of how Dockerfiles work and used them to build new images.

We saw how multi-stage builds are a great way to build smaller safer images for production environments.

We also learned that the Dockerfile is a great tool for documenting an app. As such, it can speed-up the on-boarding of new developers and bridge the gap between developers and operations staff. With this in mind, treat it like code and check it in and out of a source control system.

Although the examples cited were Linux-based apps, the process for containerizing Windows apps is the same: Start with your app code, create a Dockerfile describing the app, build the image with `docker build`.

9: Multi-container apps with Compose

In this chapter, we'll look at how to deploy multi-container applications using Docker Compose. We usually shorten it to just *Compose*.

We'll split this chapter into the usual three parts:

- The TLDR
- The deep dive
- The commands

Deploying apps with Compose - The TLDR

Modern cloud-native apps are made of multiple smaller services that interact to form a useful app. We call this the *microservices* pattern.

A microservices app might have the following seven independent services that work together to form a useful application:

- Web front-end
- Ordering
- Catalog
- Back-end datastore
- Logging
- Authentication
- Authorization

Deploying and managing lots of small microservices like these can be hard. This is where *Compose* comes in to play.

Instead of gluing microservices together with scripts and long docker commands, Compose lets you describe everything in a declarative configuration file. You can use this file to deploy it and manage it.

Once the app is *deployed*, you can *manage* its entire lifecycle with a simple set of commands. You can even store and manage the configuration file in a version control system.

That's the basics. Let's dig deeper.

Deploying apps with Compose - The Deep Dive

We'll divide the Deep Dive section as follows:

- Compose background
- Installing Compose
- Compose files
- Deploying apps with Compose
- Managing apps with Compose

Compose background

When Docker was new, a company called Orchard built a tool called *Fig* that made it really easy to manage multi-container apps. It was a Python tool that sat on top of Docker and let you define entire multi-container microservices apps in a single YAML file. You could even use *Fig* to deploy and manage the entire lifecycle of the app with the `fig` command-line tool.

Behind the scenes, *Fig* would read the YAML file and call the appropriate Docker commands to deploy and manage it.

In fact, it was so good that Docker, Inc. acquired Orchard and re-branded *Fig* as *Docker Compose*. The command-line tool was renamed from `fig` to `docker-compose` and the more recently it was folded into the `docker` CLI with its own sub-command. You can now run simple `docker compose` commands on the CLI.

There is also a Compose Specification¹² aimed at creating an open standard for defining multi-container microservices apps. The specification is community-led and kept separate from the Docker implementation. This helps maintain better governance and clearer lines of demarcation. However, we should expect Docker to implement the full spec in the Docker engine.

The spec itself is a great document to learn the details.

Time to see Compose in action.

Installing Compose

Compose now ships with the Docker engine and you no longer need to install it as a separate program.

Test it works with the following command. Be sure to use the `docker compose` command and not `docker-compose`.

¹²<https://github.com/compose-spec/compose-spec>

```
$ docker compose version
Docker Compose version v2.17.3
```

Compose files

Compose uses YAML files to define microservices applications.

The default name for a Compose YAML file is `compose.yaml`. However, it also accepts `compose.yml` and you can use the `-f` flag to specify custom filenames.

The following example shows a very simple Compose file that defines a small Flask app with two microservices (`web-fe` and `redis`). The app is a simple web server that counts the number of visits to a web page and stores the value in a Redis cache. We'll call the app `multi-container` and use it as the example application for the rest of the chapter.

The file is in the `multi-container` folder of the book's GitHub repository.

```
services:
  web-fe:
    build: .
    command: python app.py
    ports:
      - target: 8080
        published: 5001
    networks:
      - counter-net
    volumes:
      - type: volume
        source: counter-vol
        target: /app
  redis:
    image: "redis:alpine"
    networks:
      counter-net:

networks:
  counter-net:

volumes:
  counter-vol:
```

We'll skip through the basics of the file before taking a closer look.

The first thing to note is that the file has 3 top-level keys:

- `services`
- `networks`

- volumes

Other top-level keys exist, such as `secrets` and `configs`, but we're not looking at those.

The top-level `services` key is where we define application microservices. This example defines two: a web front-end called `web-fe`, and an in-memory cache called `redis`. Compose will deploy each of these microservices to its own container.

The top-level `networks` key tells Docker to create new networks. By default, modern versions of Compose create *overlay* networks that span multiple hosts. However, you can use the `driver` property to specify different network types.

The following YAML can be used in your Compose file to create a new *overlay* network called `over-net` that allows standalone containers to connect to it (attachable).

```
networks:
  over-net:
    driver: overlay
    attachable: true
```

The top-level `volumes` key is where you tell Docker to create new volumes.

Our specific Compose file

The example Compose file defines two services, a network called `counter-net`, and a volume called `counter-vol`. It's shown again here:

```
services:
  web-fe:
    build: .
    command: python app.py
    ports:
      - target: 8080
        published: 5001
    networks:
      - counter-net
    volumes:
      - type: volume
        source: counter-vol
        target: /app
  redis:
    image: "redis:alpine"
    networks:
      counter-net:
networks:
  counter-net:
volumes:
  counter-vol:
```

Most of the detail is in the `services` section, so let's take a closer look at that.

The `services` section has two second-level keys:

- `web-fe`
- `redis`

Each of these defines a microservice. It's important to know that Compose will deploy each of these as its own container and will use the name of the keys in the container names. In our example, we've defined two keys: `web-fe` and `redis`. This means Compose will deploy two containers, one will have `web-fe` in its name and the other will have `redis`.

Within the definition of the `web-fe` service, we give Docker the following instructions:

- `build:` `.` This tells Docker to build a new image using the `Dockerfile` in the current directory (`.`). The newly built image will be used in a later step to create the container for this service.
- `command:` `python app.py` This tells Docker to run a Python app called `app.py` in every container for this service. The `app.py` file must exist in the image, and the image must have Python installed. The `Dockerfile` takes care of both of these requirements.
- `ports:` The example in our Compose file tells Docker to map port `8080` inside the container (`target`) to port `5001` on the host (`published`). This means traffic hitting the Docker host on port `5001` will be directed to port `8080` on the container. The app inside the container listens on port `8080`.
- `networks:` Tells Docker which network to attach the service's containers to. The network should already exist or be defined in the `networks` top-level key. If it's an overlay network, it will need to have the `attachable` flag so that standalone containers can be attached to it (Compose deploys standalone containers instead of Docker Services).
- `volumes:` Tells Docker to mount the `counter-vol` volume (`source:`) to `/app` (`target:`) inside the container. The `counter-vol` volume needs to already exist or be defined in the `volumes` top-level key in the file.

In summary, Compose will instruct Docker to deploy a single standalone container for the `web-fe` microservice. It will be based on an image built from a `Dockerfile` in the same directory as the Compose file. This image will be started as a container and run `app.py` as its main app. It will attach to the `counter-net` network, expose itself on port `5001` on the host, and mount a volume to `/app`.

Note: We don't actually need the command: `python app.py` option in the Compose file as it's already defined in the Dockerfile. We show it here so you know how it works. You can also use Compose to override instructions set in Dockerfiles.

The definition of the `redis` service is simpler:

- `image: redis:alpine` This tells Docker to start a standalone container called `redis` based on the `redis:alpine` image. This image will be pulled from Docker Hub.
- `networks:` The `redis` container will also be attached to the `counter-net` network.

As both services will be deployed onto the same `counter-net` network, they'll be able to resolve each other by name. This is important as the application is configured to connect to the Redis service by name.

Now that we understand how the Compose file works, let's deploy it!

Deploying apps with Compose

In this section, we'll deploy the app defined in the Compose file from the previous section. To do this, you'll need a local copy of the book's GitHub repo and run all commands from the `multi-container` folder.

If you haven't already done so, clone the Git repo locally. You'll need Git installed to do this.

```
$ git clone https://github.com/nigelpoulton/ddd-book.git
```

```
Cloning into 'ddd-book'...
remote: Enumerating objects: 67, done.
remote: Counting objects: 100% (67/67), done.
remote: Compressing objects: 100% (47/47), done.
remote: Total 67 (delta 17), reused 63 (delta 16), pack-reused 0
Receiving objects: 100% (67/67), 173.61 KiB | 1.83 MiB/s, done.
Resolving deltas: 100% (17/17), done.
```

Cloning the repo will create a new directory called `ddd-book` containing all of the files used in the book.

Change directory to the `ddd-book/multi-container` directory and use this as your build context for the remainder of the chapter. Compose will also use the name of the directory (`multi-container`) as the project name. We'll see this later, but Compose will prepend all resource names with `multi-container_`.

```
$ cd ddd-book/multi-container/
$ ls -l
total 20
-rw-rw-r-- 1 ubuntu ubuntu 288 May 21 15:53 Dockerfile
-rw-rw-r-- 1 ubuntu ubuntu 332 May 21 15:53 README.md
drwxrwxr-x 4 ubuntu ubuntu 4096 May 21 15:53 app
-rw-rw-r-- 1 ubuntu ubuntu 355 May 21 15:53 compose.yaml
-rw-rw-r-- 1 ubuntu ubuntu 18 May 21 15:53 requirements.txt
```

Let's quickly describe each file:

- `compose.yaml` is the Docker Compose file that describes the app and how Compose should build and deploy it
- `app` is a folder and contains the application code and views
- `Dockerfile` describes how to build the image for the web-fe service
- `requirements.txt` lists the application dependencies

Feel free to inspect the contents of each file.

The `app/app.py` file is the core of the application, however, `compose.yaml` is the glue that sticks all the microservices together.

Let's use Compose to bring the app up. You must run all of the following commands from within the `multi-container` directory of the repo you just cloned from GitHub.

```
$ docker compose up &
```

```
[+] Running 7/7
- redis 6 layers [|||||] 0B/0B Pulled 5.2s
- 08409d417260 Already exists 0.0s
- 35afda5186ef Pull complete 0.5s
- ebab1fe9c8cc Pull complete 1.5s
- e438114652e6 Pull complete 3.1s
- 80fd0bfc19ad Pull complete 3.1s
- ca04d454c47d Pull complete 1.1s
[+] Building 10.3s (9/9) FINISHED
<Snip>
[+] Running 4/4
- Network multi-container_counter-net Created 0.1s
- Volume "multi-container_counter-vol" Created 0.0s
- Container multi-container-redis-1 Started 0.6s
- Container multi-container-web-fe-1 Started 0.5s
```

It'll take a few seconds for the app to come up and the output can be verbose. You may also have to hit the Return key when the deployment completes.

We'll step through what happened in a second, but first let's talk about the `docker compose` command.

`docker compose up` is the most common way to bring up a Compose app. It builds or pulls all required images, creates all required networks and volumes, and starts all required containers.

We didn't specify the name or location of the Compose file as it's called `compose.yaml` and in the local directory. However, if it has a different name or location we would use the `-f` flag. The following example will deploy an application from a Compose file called `prod-equus-bass.yaml`

```
$ docker compose -f prod-equus-bass.yaml up &
```

Normally you'll use the `--detach` flag to bring the app up in the background as shown next. However, we brought it up in the foreground and used the `&` to give us the terminal window back. This forces Compose to output all messages to the terminal window which we'll use later.

Now that the app is built and running, we can use normal `docker` commands to view the images, containers, networks, and volumes that Compose created — remember, Compose is building and working with normal Docker constructs behind the scenes.

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
multi-container-web-fe	latest	ade6252c30cc	7 minutes ago	76.3MB
redis	alpine	b64252cb5430	9 days ago	30.7MB

The `multi-container-web-fe:latest` image was created by the `build: .` instruction in the `compose.yaml` file. This instruction caused Docker to build a new image using the Dockerfile in the same directory. It contains the `web-fe` microservice and was built from the `python:alpine` image. See the contents of the `Dockerfile` for more information:

<code>FROM python:alpine</code>	<code><< Base image</code>
<code>COPY . /app</code>	<code><< Copy app into image</code>
<code>WORKDIR /app</code>	<code><< Set working directory</code>
<code>RUN pip install -r requirements.txt</code>	<code><< Install requirements</code>
<code>ENTRYPOINT ["python", "app.py"]</code>	<code><< Set the default app</code>

I've added comments to the end of each line to help explain.

Notice how Compose has named the newly-built image as a combination of the project name (`multi-container`) and the resource name as specified in the Compose file (`web-fe`). The project name is the name of the directory with the Compose file in it. All resources created by Compose will follow this naming convention.

The `redis:alpine` image was pulled from Docker Hub by the image: `"redis:alpine"` instruction in the `.Services.redis` section of the Compose file.

The following container listing shows two running containers. These following the same naming convention and each one has a numeric suffix that indicates the instance number — this is because Compose allows for scaling up and down.

```
$ docker ps
```

ID	COMMAND	STATUS	PORTS	NAMES
61..	"python app/app.py"	Up 5 mins	0.0.0.0:5001->8080/tcp..	multi-container-web-fe-1
80..	"docker-entrypoint.."	Up 5 mins	6379/tcp	multi-container-redis-1

The `multi-container-web-fe-1` container is running the application's web front end. This is running the `app.py` code and is mapped to port 5001 on all interfaces on the Docker host. We'll connect to this in a later step.

The following network and volume listings show the `multi-container_counter-net` network and `multi-container_counter-vol` volume.

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
46100cae7441	multi-container_counter-net	bridge	local

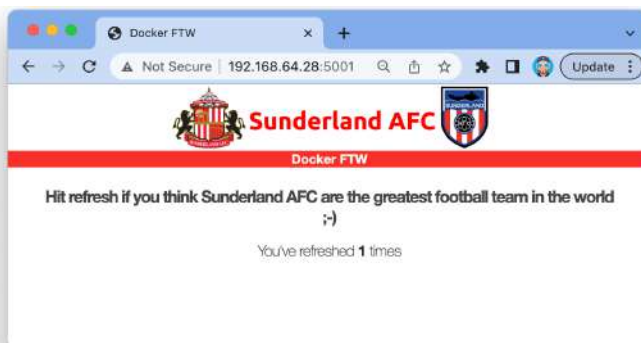
<Snip>

```
$ docker volume ls
```

DRIVER	VOLUME NAME
local	multi-container_counter-vol

<Snip>

With the application successfully deployed, you can point a web browser at your Docker host on port 5001 and see the application in all its glory.



The image shows I connected to a Docker host with an IP address of 192.168.64.28 on port 5001. If you're on Docker Desktop or another local environment you might be able to connect on localhost:5001.

Hitting your browser's refresh button will cause the counter on the web page to increment. Have a look at the app (app/app.py) to see how the counter data is stored in the Redis back-end.

As we brought the application up in the foreground, we can see the HTTP 200 response codes being logged in the terminal window for each refresh. These indicate successful requests, and you'll see one for each time you load the web page.

```
multi-container-web-fe-1 | 192.168.64.1 - - [21/May/2023 15:38:40] "GET / HTTP/1.1" 200 -
multi-container-web-fe-1 | 192.168.64.1 - - [21/May/2023 15:38:41] "GET / HTTP/1.1" 200 -
```

Congratulations. You've successfully deployed a multi-container application using Docker Compose!

Managing apps with Compose

In this section, you'll see how to stop, restart, delete, and get the status of Compose applications. You'll also see how the volume can be used to directly inject updates to the app.

As the application is already up, let's see how to bring it down. To do this, replace the up sub-command with down.

```
$ docker compose down
[+] Running 3/3
- Container multi-container-web-fe-1   Removed          0.3s
- Container multi-container-redis-1    Removed          0.2s
- Network multi-container_counter-net  Removed          0.3
```

As the app was started in the foreground we'll get verbose output to the terminal. This can give you good insight into how things work and I recommend you analyse the output. We don't go through it here in the book as it can change between different versions of Compose.

It's clear to see that the two containers (microservices) and the network have been removed.

However, volumes are **not** deleted by default. This is because volumes are intended to be long-term persistent data stores and their lifecycles are entirely decoupled from application lifecycles. Running a `docker volume ls` will prove the volume is still present on the system. If you'd written any data to it, the data would still exist. Adding

the `--volumes` flag to the `docker compose down` command will delete all associated volumes.

Any images that were built or pulled as part of the `docker-compose up` operation will also remain on the system. This means future deployments of the app will be faster. Adding the `--rm` flag to the `docker compose down` command will delete all images built or pulled when starting the app.

Let's look at a few other `docker compose` sub-commands.

Use the following command to bring the app up again, but this time in the background.

```
$ docker compose up --detach
<Snip>
```

See how the app started much faster this time — the `counter-vol` volume already exists, and all images already exist on the Docker host.

Show the current state of the app with the `docker compose ps` command.

```
$ docker compose ps
```

NAME	COMMAND	SERVICE	STATUS	PORTS
multi-container-redis-1	"docker-entrypoint.."	redis	Up 28 sec	6379/tcp
multi-container-web-fe-1	"python app/app.py"	web-fe	Up 28 sec	0.0.0.0:5001->8080

You can see both containers, the commands they are running, their current state, and the network ports they're listening on.

Use `docker compose top` to list the processes running inside of each service (container).

```
$ docker compose top
```

```
multi-container-redis-1
UID    PID    PPID    ... CMD
lxd    22312  22292  redis-server *:6379
```

```
multi-container-web-fe-1
UID    PID    PPID    ... CMD
root    22346  22326  0 python app/app.py python app.py
root    22415  22346  0 /usr/local/bin/python app/app.py python app.py
```

The PID numbers returned are the PID numbers as seen from the Docker host (not from within the containers).

Use the `docker compose stop` command to stop the app without deleting its resources. Then show the status of the app with `docker compose ps`.


```
$ docker compose stop
[+] Running 2/2
 - Container multi-container-redis-1   Stopped           0.4s
 - Container multi-container-web-fe-1   Stopped           0.5

$ docker compose ps
NAME          COMMAND          SERVICE    STATUS    PORTS
```

Previous versions of Compose used to list the containers in the stopped state. Verify that the containers for the two Compose microservices still exist on the system and are in the stopped state.

```
$ docker ps -a
CONTAINER ID   COMMAND          STATUS          NAMES
f1442d484ccd   "python app/app.py"   Exited (0)...   multi-container-web-fe-1
541efbd7185d   "docker-entrypoint"   Exited (0)...   multi-container-redis-1
```

You can delete a stopped Compose app with `docker compose rm`. This will delete the containers and networks but not the volumes or images. Nor will it delete the application source code in your project's build context directory (`app.py`, `Dockerfile`, `requirements.txt`, and `compose.yaml`).

With the app in the stopped state, restart it with the `docker compose restart` command.

```
$ docker compose restart
[+] Running 2/2
 - Container multi-container-redis-1   Started           0.4s
 - Container multi-container-web-fe-1   Started           0.5s
```

Verify the operation app is back up.

```
$ docker compose ls
NAME          STATUS          CONFIG FILES
multi-container   running(2)      /home/ubuntu/ddd-book/multi-container/compose.yaml
```

Run the following command to **stop and delete** the app with a single command. It will also delete any volumes and images used to start the app.

```
$ docker-compose down --volumes --rmi all
Stopping multi-container-web-fe-1 ... done
Stopping multi-container-redis-1 ... done
Removing multi-container-web-fe-1 ... done
Removing multi-container-redis-1 ... done
Removing network multi-container_counter-net
Removing volume multi-container_counter-vol
Removing image multi-container_web-fe
Removing image redis:alpine
```

Using volumes to insert data

Let's deploy the app one last time and see a little more about how the volume works.

```
$ docker compose up --detach
<Snip>
```

If you look in the Compose file, you'll see it defines a volume called `counter-vol` and mounts it in to the `web-fe` container at `/app`.

```
volumes:
  counter-vol:
services:
  web-fe:
    volumes:
      - type: volume
        source: counter-vol
        target: /app
```

The first time we deployed the app, Compose checked to see if a volume called `counter-vol` already existed. It didn't, so Compose created it. You can see it with the `docker volume ls` command, and you can get more detailed information with `docker volume inspect multi-container_counter-vol`.

```
$ docker volume ls
RIVER      VOLUME NAME
local      multi-container_counter-vol

$ docker volume inspect multi-container_counter-vol
[
  {
    "CreatedAt": "2023-05-21T19:49:25+01:00",
    "Driver": "local",
    "Labels": {
      "com.docker.compose.project": "multi-container",
```

```

    "com.docker.compose.version": "2.17.3",
    "com.docker.compose.volume": "counter-vol"
  },
  "Mountpoint": "/var/lib/docker/volumes/multi-container_counter-vol/_data",
  "Name": "multi-container_counter-vol",
  "Options": null,
  "Scope": "local"
}
]

```

It's also worth knowing that Compose builds networks and volumes **before** deploying services. This makes sense, as networks and volumes are lower-level infrastructure objects that are consumed by services (containers).

If we take another look at the service definition for `web-fe`, we'll see that it's mounting the `counter-app` volume into the container at `/app`. We can also see from the Dockerfile that `/app` is where the app is installed and executed from. This means the app code is running from a Docker volume. See Figure 9.2.

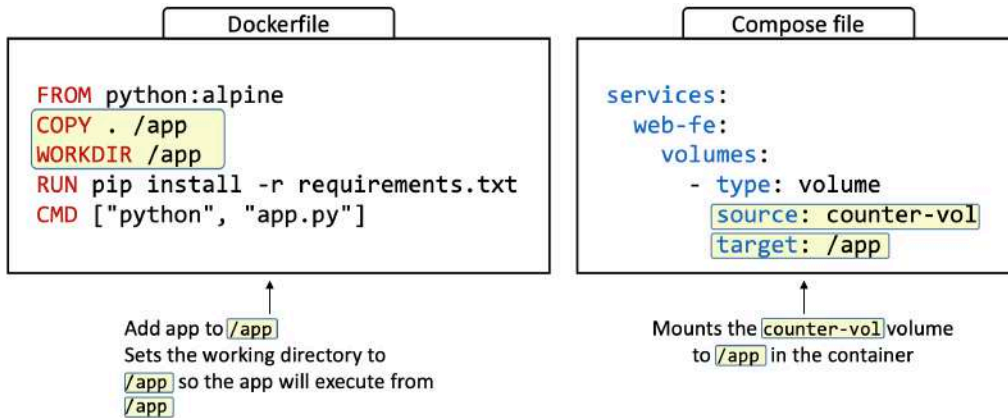


Figure 9.2

This means we can make changes to files in the volume, from the outside of the container, and have them reflected immediately in the app. Let's see how that works.

The next steps will walk you through the following process.

- Update the contents of `app/templates/index.html` in the project's build context
- Copy the updated `index.html` to the container's volume (this resides on the Docker host's filesystem)
- Refresh the web page and see the updates

Note: This won't work if you are using Docker Desktop on Mac or Windows. This is because Docker Desktop runs Docker inside of a lightweight VM on these platforms and all volumes exist inside the VM.

Use your favourite text editor to edit the `index.html`. Be sure to run the command from the `multi-container` directory.

```
$ vim app/templates/index.html
```

Change text on line 16 to the following and **save your changes**.

```
<h2>Sunderland til I die</h2>
```

Now that you've updated the app, you need to copy it into the volume on the Docker host. Each Docker volume exists at a location within the Docker host's filesystem. Use the following `docker inspect` command to find where the volume is exposed on the Docker host.

```
$ docker inspect multi-container_counter-vol | grep Mount
```

```
"Mountpoint": "/var/lib/docker/volumes/multi-container_counter-vol/_data",
```

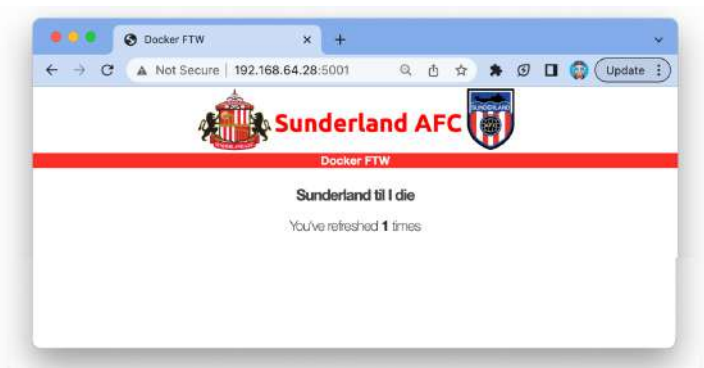
Copy the updated `index.html` file to the appropriate subdirectory below the directory returned by the previous command (remember that this will not work on Docker Desktop). As soon as you do this, the updated file will appear in the container.

You may have to prefix the command with `sudo` and you should run it on one line and **not** split it over multiple lines with the `\`. It's only split over multiple lines in the book to avoid line wrapping.

```
$ cp ./counter-app/app.py \
  var/lib/docker/volumes/multi-container_counter-vol/_data/app/templates/index.html
```

The updated app file is now on the container. Connect to the app to see your change. You can do this by pointing your web browser to the IP of your Docker host on port 5001.

Figure 9.3 shows the updated app.



You wouldn't do update operations like this in production, but it demonstrates how volumes work.

Congratulations. You've deployed and managed a multi-container microservices app using Docker Compose.

Before reminding ourselves of the commands we learned, it's important to understand that this was a very simple example. Docker Compose is capable of deploying and managing far more complex applications.

Deploying apps with Compose - The commands

- `docker compose up` is the command to deploy a Compose app. It creates all images, containers, networks and volumes needed by the app. It expects the Compose file to be called `compose.yaml` but you can specify a custom filename with the `-f` flag. It's common to start the app in the background with the `--detach` flag.
- `docker compose stop` will stop all containers in a Compose app without deleting them from the system. They can be easily restarted with `docker compose restart`.
- `docker compose rm` will delete a stopped Compose app. It will delete containers and networks, but it won't delete volumes and images by default.
- `docker compose restart` will restart a Compose app that has been stopped with `docker compose stop`. If you make changes to your Compose app while it's stopped, these changes will **not** appear in the restarted app. You need to re-deploy the app to get the changes.
- `docker compose ps` lists each container in the Compose app. It shows current state, the command running inside each container, and network ports.

- `docker compose down` will stop and delete a running Compose app. It deletes containers and networks, but not volumes and images.

Chapter Summary

In this chapter, we learned how to deploy and manage multi-container applications using Docker Compose.

Compose is now an integrated part of the Docker engine and has its own `docker` sub-command. It lets you define multi-container apps in declarative configuration file and deploy them with a single command.

Compose files can be YAML or JSON, and they define all of the containers, networks, volumes, and secrets an application requires. You then feed the file to the `docker compose` command line and Compose uses Docker to deploy it.

Once the Compose app is deployed, you can manage its entire lifecycle using the many `docker compose` sub-commands.

You also saw how volumes have a separate lifecycle to the rest of the app, as well as how you can use them to inject changes directly into running containers.

Docker Compose is popular with developers, and the Compose file is an excellent source of application documentation — it defines all the services that make up the app, the images they use, ports they expose, networks and volumes they use, and much more. As such, it can help bridge the gap between dev and ops. You should also treat Compose files the same way you treat code. This means, among other things, storing them in source control repos.

10: Docker Swarm

Now that we know how to install Docker, pull images, and work with containers, the next thing we need is a way to do it all at scale. That's where Docker Swarm comes into play.

As usual, we'll split this chapter into three parts:

- The TLDR
- The deep dive
- The commands

Docker Swarm - The TLDR

Docker Swarm is two things:

1. An enterprise-grade secure cluster of Docker hosts
2. An orchestrator of microservices apps

On the clustering front, Swarm groups one or more Docker nodes and lets you manage them as a cluster. Out-of-the-box, you get an encrypted distributed cluster store, encrypted networks, mutual TLS, secure cluster join tokens, and a PKI that makes managing and rotating certificates a breeze. You can even non-disruptively add and remove nodes. It's a beautiful thing.

On the orchestration front, Swarm allows you to deploy and manage complex microservices apps with ease. You can define your apps in declarative files and deploy them to the swarm with native Docker commands. You can even perform rolling updates, rollbacks, and scaling operations. Again, all with simple commands.

Docker Swarm is similar Kubernetes — they both orchestrate containerized applications. Kubernetes has a lot more momentum and a more active community and ecosystem. However, Swarm is a lot easier to use and is a popular choice for many small-to-medium businesses and application deployments. Learning Swarm is a stepping-stone to learning Kubernetes.

Docker Swarm - The Deep Dive

We'll split the deep dive part of this chapter as follows:

- Swarm primer
- Build a secure Swarm cluster
- Deploying Swarm services
- Troubleshooting
- Backing up and recovering a Swarm

Swarm primer

On the clustering front, a *swarm* consists of one or more Docker *nodes*. These *nodes* can be physical servers, VMs, Raspberry Pi's, or cloud instances. The only requirement is that they all have Docker installed and can communicate over reliable networks.

Terminology: When referring to *Docker Swarm* we'll write *Swarm* with an uppercase "S". When referring to a *swarm* (*cluster of nodes*) we'll use a lowercase "s".

Nodes are configured as *managers* or *workers*. *Managers* look after the control plane, meaning things like the state of the cluster and dispatching tasks to *workers*. *Workers* accept tasks from *managers* and execute them.

The configuration and state of the *swarm* is held in a distributed database replicated on all managers. It's kept in-memory and is extremely up-to-date. However, the best thing is that it requires zero configuration — it's installed as part of the swarm and just takes care of itself.

TLS is so tightly integrated that it's impossible to build a swarm without it. In today's security conscious world, things like this deserve all the plaudits they get. *Swarm* uses TLS to encrypt communications, authenticate nodes, and authorize roles. Automatic key rotation is also thrown in as the icing on the cake. And the best part... it all happens so smoothly that you don't even know it's there.

On the orchestration front, the atomic unit of scheduling on a swarm is the *service*. This is a high-level construct that wraps some advanced features around containers. These features include scaling, rolling updates, and simple rollbacks. It's useful to think of a *service* as an enhanced container.

A high-level view of a swarm is shown in Figure 10.1.

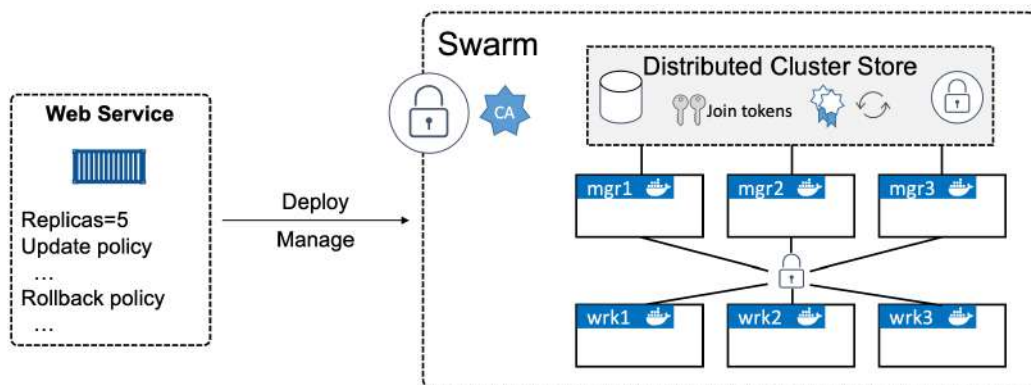


Figure 10.1 High-level swarm

That's enough of a primer. Let's get our hands dirty with some examples.

Build a secure swarm cluster

In this section, we'll build a secure swarm cluster with three *manager nodes* and three *worker nodes* as shown in Figure 10.2.

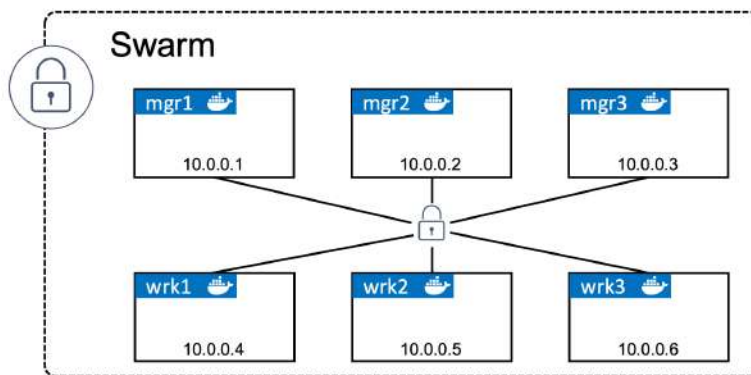


Figure 10.2

Pre-reqs

If you plan on following along, I recommend using Multipass to create multiple Docker VMs on your laptop or local machine. Multipass is free and easy to use, and all VMs will be able to communicate. Just install Multipass and then use the following commands to create VMs and log on to them:

- Create new Docker VMs with: `multipass launch docker --name <name>`
- List Multipass VMs and their IPs with: `multipass ls`
- Log on to a Multipass VM with: `multipass shell <name>`
- Log out of a Multipass VM with: `exit`

I've created 6 VMs and named them according to Figure 10.2.

If you can't use Multipass, I recommend creating multiple nodes on Play with Docker at <https://labs.play-with-docker.com>. It's free to use and you get a 4-hour playground.

However, any Docker environment should work. The only requirements are that each node has Docker installed and can communicate over a reliable network. It's also beneficial if name resolution is configured — it makes it easier to identify nodes in command outputs and helps when troubleshooting.

If you plan to follow along on Docker Desktop, be warned that it only supports a single Docker node. This is OK but isn't the best for some of the later examples.

If you think you hit networking issues, be sure the following ports are open between all swarm nodes:

- 2377/tcp: for secure client-to-swarm communication
- 7946/tcp and udp: for control plane gossip
- 4789/udp: for VXLAN-based overlay networks

Initializing a new swarm

The process of building a swarm is called *initializing a swarm*, and the high-level process is this: Initialize the first manager > Join additional managers > Join workers > Done.

Docker nodes that are not part of a swarm are said to be in *single-engine mode*. Once they're added to a swarm they're automatically switched into *swarm mode*.

Running `docker swarm init` on a Docker host in *single-engine mode* will switch that node into *swarm mode*, create a new *swarm*, and make the node the first *manager* of the swarm.

Additional nodes can then be joined as either workers or managers, and the join process automatically flips them into *swarm mode*.

The following steps will initialize a new swarm from **mgr1**. It will then join **wrk1**, **wrk2**, and **wrk3** as worker nodes — automatically putting them into *swarm mode* as part of the process. Finally, it will add **mgr2** and **mgr3** as additional managers and switch them into *swarm mode*. At the end of the procedure, all 6 nodes will be in *swarm mode* and operating as part of the same swarm.

This example will use the names and IP addresses of the nodes shown in Figure 10.2. Yours may be different.

1. Log on to **mgr1** and initialize a new swarm. This command uses the IP address from Figure 10.2. You should use an appropriate private IP on your Docker host. If you're using Multipass this will usually be the VM's 192 address.

```
$ docker swarm init \
  --advertise-addr 10.0.0.1:2377 \
  --listen-addr 10.0.0.1:2377
```

Swarm initialized: current node (d21lyz...c79qzkx) is now a manager.
<Snip>

The command can be broken down as follows:

- `docker swarm init`: This tells Docker to initialize a new swarm and make this node the first manager. It also puts the node into *swarm mode*.
- `--advertise-addr`: This is the swarm API endpoint that will be advertised to other managers and workers. It will usually be one of the node's IP addresses but can be an external load-balancer address. It's an optional flag unless you need to specify a load-balancer or specific IP on a node with multiple IPs.
- `--listen-addr`: This is the IP address that the node will accept swarm traffic on. If you don't set it, it defaults to the same value as `--advertise-addr`. If `--advertise-addr` is a load-balancer, you **must** use `--listen-addr` to specify a local IP or interface for swarm traffic.

For production environments I recommend you be specific and always use both flags. It's not so important for lab environments like ours.

The default port that Swarm mode operates on is **2377**. This is customizable, but it's convention to use `2377/tcp` for secured (HTTPS) client-to-swarm connections.

2. List the nodes in the swarm.

```
$ docker node ls
ID                HOSTNAME        STATUS    AVAILABILITY    MANAGER STATUS
d21...qzkx * mgr1          Ready     Active           Leader
```

mgr1 is currently the only node in the swarm and is listed as the *Leader*. We'll come back to this in a second.

3. From **mgr1** run the `docker swarm join-token` command to extract the commands and tokens required to add new workers and managers to the swarm.

```
$ docker swarm join-token worker
To add a manager to this swarm, run the following command:
docker swarm join \
  --token SWMTKN-1-0uahebax...c87tu8dx2c \
  10.0.0.1:2377

$ docker swarm join-token manager
To add a manager to this swarm, run the following command:
docker swarm join \
  --token SWMTKN-1-0uahebax...ue4hv6ps3p \
  10.0.0.1:2377
```

Notice that the commands to join workers and managers are identical apart from the join tokens (SWMTKN...). This means that whether a node joins as a worker or a manager depends entirely on which token you use when joining it. **You should keep your join tokens in a safe place as they're the only thing required to join a node to a swarm!**

4. Log on to **wrk1** and join it to the swarm using the `docker swarm join` command with the worker join token.

```
$ docker swarm join \
  --token SWMTKN-1-0uahebax...c87tu8dx2c \
  10.0.0.1:2377 \
  --advertise-addr 10.0.0.4:2377 \
  --listen-addr 10.0.0.4:2377
```

This node joined a swarm as a worker.

The `--advertise-addr`, and `--listen-addr` flags are optional. I've added them as I consider it best practice to be as specific as possible when it comes to network configuration in production environments. You probably don't need them just for a lab.

5. Repeat the previous step on **wrk2** and **wrk3** so that they join the swarm as workers. If you're specifying the `--advertise-addr` and `--listen-addr` flags, make sure you use **wrk2** and **wrk3's** respective IP addresses.
6. Log on to **mgr2** and join it to the swarm as a manager using the `docker swarm join` command with the manager join token.

```
$ docker swarm join \
  --token SWMTKN-1-0uahebax...ue4hv6ps3p \
  10.0.0.1:2377 \
  --advertise-addr 10.0.0.2:2377 \
  --listen-addr 10.0.0.2:2377
```

This node joined a swarm as a manager.

7. Repeat the previous step on **mgr3**, remembering to use **mgr3's** IP address for the `advertise-addr` and `--listen-addr` flags.
8. List the nodes in the swarm by running `docker node ls` from any of the manager nodes.

```
$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
0g4rl...babl8 *	mgr2	Ready	Active	Reachable
2xlti...l0nyp	mgr3	Ready	Active	Reachable
8yv0b...wmr67	wrk1	Ready	Active	
9mzwf...e4m4n	wrk3	Ready	Active	
d2lly...9qzkx	mgr1	Ready	Active	Leader
e62gf...l5wt6	wrk2	Ready	Active	

Congratulations. You've created a 6-node swarm with 3 managers and 3 workers. As part of the process, the Docker Engine on each node was automatically put into *swarm mode* and the *swarm* was automatically secured with TLS.

If you look in the `MANAGER STATUS` column, you'll see the three manager nodes are showing as either "Reachable" or "Leader". We'll learn more about leaders shortly. Nodes with nothing in the `MANAGER STATUS` column are *workers*. Also note the asterisk (*) after the ID on the line showing **mgr2**. This tells you which node you're executing commands from. The previous command was issued from **mgr2**.

Note: It's a pain to specify the `--advertise-addr` and `--listen-addr` flags every time you join a node to the swarm. However, it can be a much bigger pain if you get the network configuration of your swarm wrong. Also, manually adding nodes to a swarm is unlikely to be a daily task, so it's worth the extra up-front effort to use the flags. It's your choice though. In lab environments or nodes with only a single IP you probably don't need to use them.

Now that you have a *swarm* up and running, let's take a look at manager high availability (HA).

Swarm manager high availability (HA)

So far, we've added three manager nodes to a swarm. Why three? And how do they work together?

Swarm *managers* have native support for high availability (HA). This means one or more can fail and the survivors will keep the swarm running.

Technically speaking, swarm implements *active/passive multi-manager HA*. This means only one manager is *active* at any given moment. This active manager is called the “*leader*”, and the leader is the only one that will ever issue updates to the *swarm*. So, it's only ever the leader that changes the config, or issues tasks to workers. If a follower manager (passive) receives commands for the swarm, it proxies them across to the leader.

This process is shown in Figure 10.3. Step 1 is the command coming into a *manager* from a remote Docker client. Step 2 is the non-leader manager receiving the command and proxying it to the leader. Step 3 is the leader executing the command on the swarm.

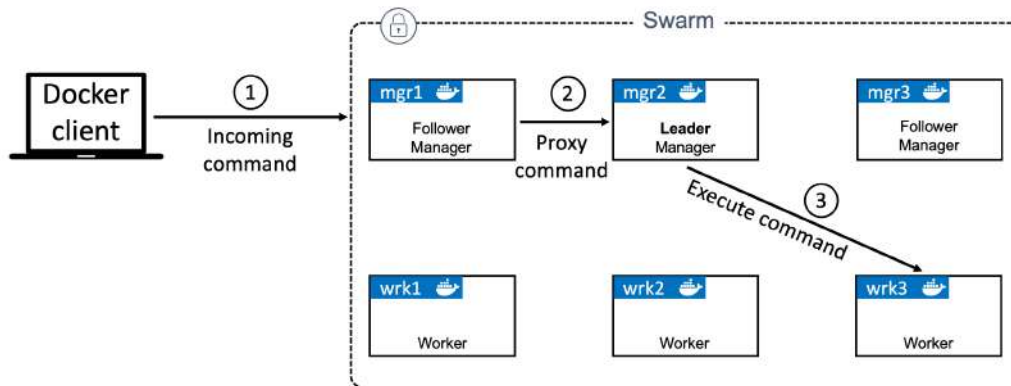


Figure 10.3

Leaders and *followers* is *Raft* terminology. This is because swarm uses an implementation of the Raft consensus algorithm¹³ to maintain a consistent cluster state across multiple highly-available managers.

On the topic of HA, the following two best practices apply:

1. Deploy an odd number of managers
2. Don't deploy too many managers (3 or 5 is recommended)
3. Spread managers across availability zones

¹³<https://raft.github.io/>

Having an odd number of managers reduces the chances of *split-brain conditions*. For example, if we had 4 managers and the network partitioned, we could be left with two managers on each side of the partition. This is known as a split brain — each side knows there used to be 4 but can now only see 2. But crucially, neither side has any way of knowing if the other two are still alive and whether it holds a majority (quorum). Apps on a swarm cluster continue to operate during split-brain conditions, however, we're not able to alter the configuration or add and manage application workloads.

However, if we have 3 or 5 managers and the same network partition occurs, it is impossible to have an equal number of managers on both sides of the partition. This means that one knows it has a majority (quorum) and full cluster management services remain available. The example on the right side of Figure 10.4 shows a partitioned cluster where the left side of the split knows it has a majority of managers.

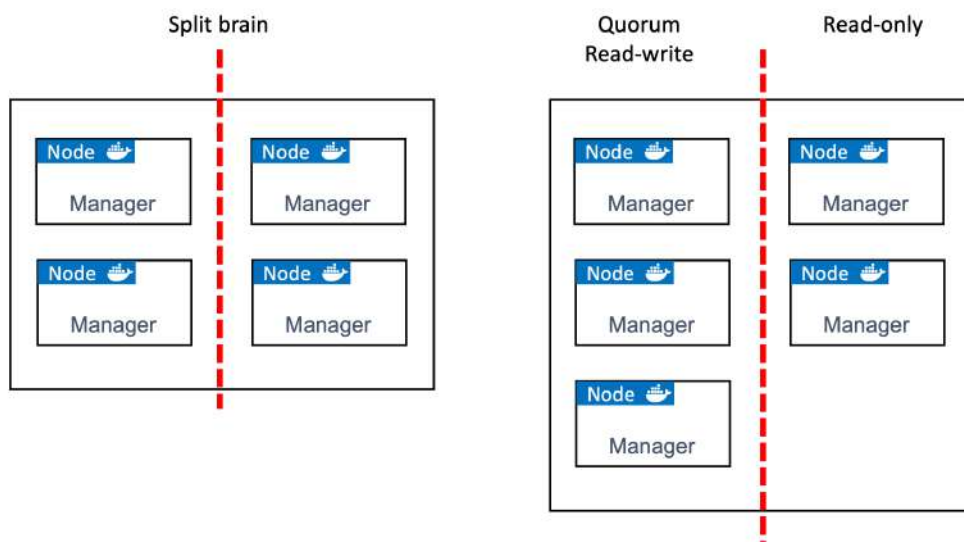


Figure 10.4

As with all consensus algorithms, more participants means more time required to achieve consensus. It's like deciding where to eat — it's always quicker and easier for 3 people to make a decision than it is for 33! With this in mind, it's a best practice to have either 3 or 5 managers for HA. 7 might work, but it's generally accepted that 3 or 5 is optimal.

A final word of caution regarding manager HA. While it's a good practice to spread your managers across availability zones, you need to make sure the networks connecting them are reliable, as network partitions can be difficult to troubleshoot and resolve. This means, at the time of writing, having managers on different cloud platforms for multi-cloud HA probably isn't a great idea.

Built-in Swarm security

Swarm clusters have a ton of built-in security that's configured out-of-the-box with sensible defaults — CA settings, join tokens, mutual TLS, encrypted cluster store, encrypted networks, cryptographic node ID's and more.

Locking a Swarm

Despite all of this built-in security, restarting an older manager or restoring an old backup has the potential to compromise the cluster. Old managers re-joining might be able to decrypt and gain access to the Raft log time-series database. They may also pollute, or wipe-out, the current swarm configuration.

To prevent situations like these, Docker allows you to lock a swarm with the Autolock feature. This forces restarted managers to present a key before being admitted back into the cluster.

It's possible to lock a swarm as part of the initialization process by passing the `--autolock` flag to the `docker swarm init` command. However, we've already built a swarm, so we'll lock ours using the `docker swarm update` command.

Run the following command from a swarm manager.

```
$ docker swarm update --autolock=true
Swarm updated.
```

To unlock a swarm manager after it restarts, run the `'docker swarm unlock'` command and provide the following key:

```
SWMKEY-1-XDeU3XC75Ku7rvGXixJ0V7evhDJGvIAvq0D8VuEAEaw
```

Please remember to store this key in a password manager, since without it you will not be able to restart the manager.

Be sure to keep the unlock key in a secure place. You can always check your current swarm unlock key with the `docker swarm unlock-key` command.

Restart one of your manager nodes to see if it automatically re-joins the cluster. You may need to prefix the command with `sudo`.

```
$ service docker restart
```

Try and list the nodes in the swarm.

```
$ docker node ls
```

Error response from daemon: Swarm is encrypted and needs to be unlocked before it can be used.

Although the Docker service has restarted on the manager, it hasn't been allowed to re-join the swarm. You can prove this even further by running a `docker node ls` command on another manager node. The restarted manager will show as down and unreachable.

Run the `docker swarm unlock` command to unlock the swarm for the restarted manager. You'll need to run this command on the restarted manager, and you'll need to provide the unlock key.

```
$ docker swarm unlock
```

Please enter unlock key: <enter your key>

The node will be allowed to re-join the swarm and will show as ready and reachable if you run another `docker node ls`.

Locking your swarm and protecting the unlock key is recommended for production environments.

Dedicated manager nodes

By default, manager and worker nodes can execute user applications. In production environments it's common to configure swarms so that only workers execute user applications. This allows managers to focus solely on control-plane duties.

Run the following three commands from any manager to prevent all three managers from running application containers.

```
$ docker node update --availability drain mgr1
```

```
$ docker node update --availability drain mgr2
```

```
$ docker node update --availability drain mgr3
```

You'll see this in action in later steps when we deploy services with multiple replicas.

Now that we've got our *swarm* built and we understand the infrastructure concepts of *leaders* and *manager HA*, let's move on to the application side of things.

Deploying Swarm services

Everything we do in this section of the chapter gets improved on by *Docker Stacks* in a later chapter. However, it's important that you learn the concepts here so that you're prepared for later.

Services let us specify most of the familiar container options such as *name*, *port mappings*, *attaching to networks*, and *images*. But they add important cloud-native features, including *desired state* and *reconciliation*. For example, swarm services let us define the desired state for an application and let the swarm take care of deploying it and managing it.

Let's look at a quick example. Assume you have an app with a web front-end. You have an image for the web server and testing has shown that you need 5 instances to handle normal daily traffic. You translate this requirement into a single *service* declaring the image to use, and that the service should always have 5 running replicas. You issue that to the swarm as your desired state and the swarm takes care of ensuring there are always 5 instances of the web server running.

We'll see some of the other things that can be declared as part of a service in a minute, but before we do that, let's see one way to create what we just described.

You can create services in one of two ways:

1. Imperatively on the command line with `docker service create`
2. Declaratively with a stack file

We'll look at stack files in a later chapter. For now we'll focus on the imperative method.

```
$ docker service create --name web-fe \
  -p 8080:8080 \
  --replicas 5 \
  nigelpoulton/ddd-book:web0.1

z70vearqmruwk0u2vc5o7ql0p
overall progress: 5 out of 5 tasks
1/5: running [=====>]
2/5: running [=====>]
3/5: running [=====>]
4/5: running [=====>]
5/5: running [=====>]
verify: Service converged
```

Let's review the command and output.

The `docker service create` command tells Docker to deploy a new service. We used the `--name` flag to name it **web-fe**. We told Docker to map port 8080 on every swarm node to 8080 inside of each service replica (container). Next, we used the `--replicas` flag to tell Docker there should always be 5 replicas of this service. Finally, we told Docker which image to base the replicas on — it's important to understand that all service replicas use the same image and config!

Terminology: Services deploy containers, and we often call these containers *replicas*. For example, a service that deploys three replicas will deploy three identical containers.

The command was sent to a manager node and the *leader manager* instantiated 5 replicas across the *swarm*. Managers on this swarm aren't allowed to run application containers, meaning all 5 replicas are deployed to worker nodes. Each worker that received a work task pulled the image and started a replica listening on port 8080. The swarm leader also ensured a copy of the *desired state* was stored on the cluster and replicated to every manager.

But this isn't the end. All *services* are constantly monitored by the swarm — the swarm runs a background *reconciliation loop* that constantly compares the *observed state* of the service with the *desired state*. If the two states match, the world is a happy place and no further action is needed. If they don't match, swarm takes actions to bring *observed state* into line with *desired state*.

As an example, if a *worker* hosting one of the 5 replicas fails, the *observed state* of the service will drop from 5 replicas to 4 and will no longer match the *desired state* of 5. As a result, the swarm will start a new replica to bring the *observed state* back in line with *desired state*. We call this *reconciliation* or *self-healing* and it's a key tenet of cloud-native applications.

Viewing and inspecting services

You can use the `docker service ls` command to see a list of all services running on a swarm.

```
$ docker service ls
```

ID	NAME	MODE	REPLICAS	IMAGE	PORTS
z7o...uw	web-fe	replicated	5/5	nigelpoulton/ddd...	*:8080->8080/tcp

The output shows a single service and some basic config and state info. Among other things, we can see the name of the service and that 5 out of the 5 desired replicas are running. If you run this command soon after deploying the service it might not show all replicas as running. This is usually while the workers pull the image.

You can use the `docker service ps` command to see a list of service replicas and the state of each.

```
$ docker service ps web-fe
```

ID	NAME	IMAGE	NODE	DESIRED	CURRENT
817...f6z	web-fe.1	nigelpoulton/...	wrk1	Running	Running 2 mins
a1d...mzn	web-fe.2	nigelpoulton/...	wrk1	Running	Running 2 mins
cc0...ar0	web-fe.3	nigelpoulton/...	wrk2	Running	Running 2 mins
6f0...azu	web-fe.4	nigelpoulton/...	wrk3	Running	Running 2 mins
dyl...p3e	web-fe.5	nigelpoulton/...	wrk3	Running	Running 2 mins

The format of the command is `docker service ps <service-name or service-id>`. The output displays each replica on its own line, shows the node it's running on, and shows desired state and the current observed state.

For detailed information about a service, use the `docker service inspect` command.

```
$ docker service inspect --pretty web-fe
ID:                z70vearqmruwk0u2vc5o7ql0p
Name:              web-fe
Service Mode:      Replicated
  Replicas:         5
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:             nigelpoulton/ddd-book:web0.1@sha256:8d6280c0042...1b9e4336730e5
  Init:              false
Resources:
Endpoint Mode:      vip
Ports:
  PublishedPort = 8080
  Protocol = tcp
  TargetPort = 8080
  PublishMode = ingress
```

The example uses the `--pretty` flag to limit the output to the most interesting items printed in an easy-to-read format. Leaving off the `--pretty` flag will give you more info. I highly recommend you read through the output of `docker inspect` commands as they're a great source of information and a great way to learn what's going on under the hood.

We'll come back to some of these outputs later.

Replicated vs global services

The default replication mode of a service is `replicated`. This deploys a desired number of replicas and distributes them as evenly as possible across the cluster.

The other mode is `global`. This runs a single replica on every node in the swarm.

To deploy a *global service* you need to pass the `--mode global` flag to the `docker service create` command. Global services do not accept the `--replicas` flag as they always run one replica per available node. However, they do respect a node's *availability*.

For example, if you've drained manager nodes so they don't run application containers, global services will not schedule replicas to them.

Scaling a service

Another powerful feature of *services* is the ability to easily scale them up and down.

Let's assume business is booming and we're seeing double the amount of traffic hitting the web front-end. Fortunately, we can easily scale the service up with the `docker service scale` command.

```
$ docker service scale web-fe=10
web-fe scaled to 10
overall progress: 10 out of 10 tasks
1/10: running [=====>]
2/10: running [=====>]
3/10: running [=====>]
4/10: running [=====>]
5/10: running [=====>]
6/10: running [=====>]
7/10: running [=====>]
8/10: running [=====>]
9/10: running [=====>]
10/10: running [=====>]
```

This command scales the number of service replicas from 5 to 10. In the background it updates the service's *desired state* from 5 to 10. Run another `docker service ls` to verify the operation was successful.

```
$ docker service ls
```

ID	NAME	MODE	REPLICAS	IMAGE	PORTS
z7o...uw	web-fe	replicated	10/10	nigelpoulton/ddd...	*:8080->8080/tcp

Running a `docker service ps` command will show that the service replicas are balanced evenly across all available nodes.

```
$ docker service ps web-fe
```

ID	NAME	IMAGE	NODE	DESIRED	CURRENT
nwf...tpn	web-fe.1	nigelpoulton/...	wrk1	Running	Running 7 mins
yb0...e3e	web-fe.2	nigelpoulton/...	wrk3	Running	Running 7 mins
mos...gf6	web-fe.3	nigelpoulton/...	wrk2	Running	Running 7 mins
utn...6ak	web-fe.4	nigelpoulton/...	wrk3	Running	Running 7 mins
2ge...fyy	web-fe.5	nigelpoulton/...	wrk2	Running	Running 7 mins
64y...m49	web-fe.6	igelpoulton/...	wrk3	Running	Running about a min
ild...51s	web-fe.7	nigelpoulton/...	wrk1	Running	Running about a min
vah...rjf	web-fe.8	nigelpoulton/...	wrk1	Running	Running about a min
xe7...fvu	web-fe.9	nigelpoulton/...	wrk2	Running	Running 45 seconds ago
l7k...jkh	web-fe.10	nigelpoulton/...	wrk1	Running	Running 46 seconds ago

Behind the scenes, Swarm runs a scheduling algorithm called *spread* that attempts to balance replicas as evenly as possible across available nodes. At the time of writing, this amounts to running an equal number of replicas on each node without taking into consideration things like CPU load etc.

Run another `docker service scale` command to bring the number down from 10 to 5.

```
$ docker service scale web-fe=5
web-fe scaled to 5
overall progress: 5 out of 5 tasks
1/5: running [=====>]
2/5: running [=====>]
3/5: running [=====>]
4/5: running [=====>]
5/5: running [=====>]
verify: Service converged
```

Now that you know how to scale services, let's see how to remove them.

Removing services

Removing services is simple — may be too simple.

Run the following `docker service rm` command to delete the `web-fe` service.

```
$ docker service rm web-fe
web-fe
```

Confirm it's gone with the `docker service ls` command.

```
$ docker service ls
ID          NAME          MODE          REPLICAS    IMAGE          PORTS
```

Be careful using this command as it deletes all service replicas without asking for confirmation.

Let's look at how to push rolling updates.

Rolling updates

Pushing updates to applications is a fact of life, and for the longest time it was a painful process. I've personally lost more than enough weekends to major application updates and I've no intention of doing it again.

Well... thanks to Docker *services*, pushing updates to well-designed microservices apps is easy.

Terminology: We use terms like *rollouts*, *updates*, and *rolling updates* to mean the same thing – updating a live application.

To see a rollout, we're going to deploy a new service. But before we do that, we're going to create a new overlay network for the service. This isn't necessary, but I want you to see how it is done and how to attach the service to it.

```
$ docker network create -d overlay uber-net
43wfp6pzea470et4d57udn9ws
```

Run a `docker network ls` to verify that the network created properly and is visible on the Docker host.

```
$ docker network ls
NETWORK ID          NAME          DRIVER          SCOPE
43wfp6pzea47        uber-net      overlay         swarm
<Snip>
```

The `uber-net` network was successfully created with the `swarm` scope and is *currently* only visible on manager nodes in the swarm. It will be dynamically extended to worker nodes when they run workloads that use it.

An overlay network is a layer 2 network that can span all swarm nodes. All containers on the same overlay network will be able to communicate, even if they're deployed to different nodes. This works even if all swarm nodes are on different underlying networks.

Figure 10.5 shows four swarm nodes connected to two different underlay networks connected by a layer 3 router. The overlay network spans all 4 nodes and creates a single flat layer 2 network that abstracts all the underlying networking.

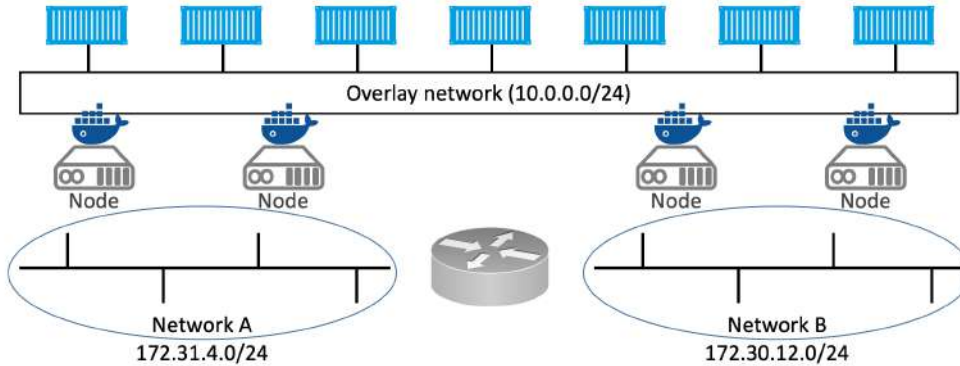


Figure 10.5

Let's create a new service and attach it to the uber-net network.

```
$ docker service create --name uber-svc \
  --network uber-net \
  -p 8080:8080 --replicas 12 \
  nigelpoulton/ddd-book:web0.1

dhbtgvqrg2q4sg07ttfuhg8nz
overall progress: 12 out of 12 tasks
1/12: running  [=====>]
2/12: running  [=====>]
3/12: running  [=====>]
<Snip>
12/12: running [=====>]
verify: Service converged
```

Let's see what we deployed.

The first thing we did was name the service `uber-svc`. We then used the `--network` flag to tell it to attach all replicas to the `uber-net` network. We then exposed port 8080 across the entire swarm and mapped it to port 8080 inside each of the 12 replicas we asked it to run. Finally, we told it to base all replicas on the `nigelpoulton/ddd-book:web0.1` image.

This mode of publishing a port on every node in the swarm — even nodes not running service replicas — is called *ingress mode* and is the default. The alternative mode is *host mode* which only publishes the service on swarm nodes running replicas.

Run a `docker service ls` and a `docker service ps` to verify the state of the new service.

```
$ docker service ls
```

ID	NAME	REPLICAS	IMAGE
dhbtgvqrg2q4	uber-svc	12/12	nigelpoulton/ddd-book:web0.1


```
$ docker service ps uber-svc
```

ID	NAME	IMAGE	NODE	DESIRED	CURRENT	STATE
0v...7e5	uber-svc.1	nigelpoulton/ddd...	wrk3	Running	Running	1 min
bh...wa0	uber-svc.2	nigelpoulton/ddd...	wrk2	Running	Running	1 min
23...u97	uber-svc.3	nigelpoulton/ddd...	wrk2	Running	Running	1 min
82...5y1	uber-svc.4	nigelpoulton/ddd...	wrk2	Running	Running	1 min
c3...gny	uber-svc.5	nigelpoulton/ddd...	wrk3	Running	Running	1 min
e6...3u0	uber-svc.6	nigelpoulton/ddd...	wrk1	Running	Running	1 min
78...r7z	uber-svc.7	nigelpoulton/ddd...	wrk1	Running	Running	1 min
2m...kdz	uber-svc.8	nigelpoulton/ddd...	wrk3	Running	Running	1 min
b9...k7w	uber-svc.9	nigelpoulton/ddd...	wrk3	Running	Running	1 min
ag...v16	uber-svc.10	nigelpoulton/ddd...	wrk2	Running	Running	1 min
e6...dfk	uber-svc.11	nigelpoulton/ddd...	wrk1	Running	Running	1 min
e2...k1j	uber-svc.12	nigelpoulton/ddd...	wrk1	Running	Running	1 min

Open a web browser and point it to the IP address of any swarm node on port 8080 to see the service running.



Figure 10.6

Feel free to point your web browser to other nodes in the swarm. You'll be able to reach the web service from any node because the service is published across the entire swarm.

Let's now assume there's another book you need to add to the site. Let's also assume a new image has been created for it and added to the same Docker Hub repository, but this one is tagged as `web0.2` instead of `web0.1`.

Let's also assume that you've been tasked with pushing the updated image to the swarm in a staged manner — 2 replicas at a time with a 20 second delay between each. You can use the following `docker service update` command to accomplish this.

```
$ docker service update \
  --image nigelpoulton/ddd-book:web0.2 \
  --update-parallelism 2 \
  --update-delay 20s \
  uber-svc

uber-svc
overall progress: 2 out of 12 tasks
1/12: running  [=====>]
2/12: running  [=====>]
3/12: ready    [=====> ]
4/12: ready    [=====> ]
5/12:
6/12:
<Snip>
11/12:
12/12:
```

Let's review the command. `docker service update` lets us make updates to running services by updating the service's desired state. This example specifies a new version of the image (`web0.2` instead of `web0.1`). It also specifies the `--update-parallelism` and `--update-delay` flags to make sure that the new image is pushed out 2 replicas at a time with a 20 second cool-off period after each. Finally, it instructs the swarm to make the changes to the `uber-svc` service.

If you run a `docker service ps uber-svc` while the update is in progress, some of the replicas will be on the new version and some on the old. If you give the operation enough time to complete, all replicas will eventually reach the new desired state of using the `web0.2` image.

```
$ docker service ps uber-svc
```

ID	NAME	IMAGE	NODE	DESIRED	CURRENT STATE
7z...nys	uber-svc.1	nigel...web0.2	mgr2	Running	Running 13 secs
0v...7e5	_uber-svc.1	nigel...web0.1	wrk3	Shutdown	Shutdown 13 secs
bh...wa0	uber-svc.2	nigel...web0.1	wrk2	Running	Running 1 min
e3...gr2	uber-svc.3	nigel...web0.2	wrk2	Running	Running 13 secs
23...u97	_uber-svc.3	nigel...web0.1	wrk2	Shutdown	Shutdown 13 secs
82...5y1	uber-svc.4	nigel...web0.1	wrk2	Running	Running 1 min
c3...gny	uber-svc.5	nigel...web0.1	wrk3	Running	Running 1 min
e6...3u0	uber-svc.6	nigel...web0.1	wrk1	Running	Running 1 min
78...r7z	uber-svc.7	nigel...web0.1	wrk1	Running	Running 1 min
2m...kdz	uber-svc.8	nigel...web0.1	wrk3	Running	Running 1 min
b9...k7w	uber-svc.9	nigel...web0.1	wrk3	Running	Running 1 min
ag...v16	uber-svc.10	nigel...web0.1	wrk2	Running	Running 1 min

```
e6...dfk  uber-svc.11  nigel...web0.1  wrk1  Running  Running 1 min
e2...k1j  uber-svc.12  nigel...web0.1  wrk1  Running  Running 1 min
```

You can observe the update in real-time by opening a web browser to any swarm node on port 8080 and hitting refresh a few times. Some of the requests will be serviced by replicas running the old version and some will be serviced by replicas running the new version. After enough time, all requests will be serviced by replicas running the updated version.

Congratulations. You’ve just completed a zero-downtime rolling update to a live containerized application.

If you run a `docker service inspect --pretty` command against the service, you’ll see the update parallelism and update delay settings have been merged into the service’s definition. This means future updates will automatically use these settings unless you override them as part of the `docker service update` command.

```
$ docker service inspect --pretty uber-svc
ID:          mub0dgtc8szm80ez5bs8wlt19
Name:        uber-svc
Service Mode: Replicated
  Replicas:   12
<Snip>
UpdateConfig:
  Parallelism: 2          <-----
  Delay:       20s        <-----
<Snip>
ContainerSpec:
  Image:       nigelpoulton/ddd-book:web0.2@sha256:8fc6161f981b...4c2d16062678d
Resources:
Networks:     uber-net
Ports:
  PublishedPort = 8080
  Protocol      = tcp
  TargetPort    = 8080
  PublishMode   = ingress
```

You should also note a couple of things about the service’s network config. All nodes in the swarm that are running a replica for the service will have the `uber-net` overlay network that we created earlier. We can verify this by running `docker network ls` on any node running a replica.

You should also note the `Networks` portion of the `docker service inspect` output. This shows the `uber-net` network as well as the swarm-wide (`PublishMode: ingress`) port mapping.

Troubleshooting

Swarm Service logs can be viewed with the `docker service logs` command. However, not all logging drivers support it.

By default, Docker nodes configure services to use the `json-file` log driver, but other drivers exist, including:

- `awslogs`
- `gelf`
- `gcplogs`
- `journald` (only works on Linux hosts running `systemd`)
- `splunk`
- `syslog`

`json-file` and `journald` are the easiest to configure, and both work with the `docker service logs` command. The format of the command is `docker service logs <service-name>`.

If you're using 3rd-party logging drivers, you should view those logs using the logging platform's native tools.

The following snippet from a `daemon.json` configuration file shows a Docker host configured to use `syslog`. The default location for `daemon.json` is `/etc/docker/daemon.json` but the file may not exist unless you manually create it to configure custom settings.

```
{
  "log-driver": "syslog"
}
```

You can force individual services to use a different driver by passing the `--log-driver` and `--log-opts` flags to the `docker service create` command. These will override anything set in `daemon.json`.

Service logs expect applications to run as PID 1 and send logs to `STDOUT` and errors to `STDERR`. The logging driver forwards these “logs” to the locations configured via the logging driver.

The following `docker service logs` command shows the logs for all replicas in a service called `svc1` that experienced a couple of failures starting a replica.

```
$ docker service logs svc1
svc1.1.zhc3cjeti9d4@wrk2 | [emerg] 1#1: host not found...
svc1.1.zhc3cjeti9d4@wrk2 | nginx: [emerg] host not found..
svc1.1.6m1nmbzmwh2d@wrk2 | [emerg] 1#1: host not found...
svc1.1.6m1nmbzmwh2d@wrk2 | nginx: [emerg] host not found..
svc1.1.1tmya243m5um@mgr1 | 10.255.0.2 "GET / HTTP/1.1" 302
```

The output is trimmed to fit the page, but you can see that logs from all three service replicas are shown (the two that failed and the one that’s running). Each line starts with the name of the replica, which includes the service name, replica number, replica ID, and name of host that it’s scheduled on. Following that is the log output.

It’s hard to tell because it’s trimmed to fit the book, but it looks like the first two replicas failed because they were trying to connect to another service that was still starting.

You can follow logs (`--follow`), tail them (`--tail`), and get extra details (`--details`).

Backing up and recovering a Swarm

Backing up a swarm is the process of backup the control plane and can be used to recover swarm in the event of a catastrophic failure or corruption. Recovering a swarm from a backup is extremely rare. However, business critical environments should always be prepared for worst-case scenarios.

You might be asking why backups are necessary if the control plane is replicated and highly-available (HA). To answer this question, consider the scenario where a malicious actor deletes all of the Secrets on a swarm. HA cannot help in this scenario as the delete operation is automatically replicated to all manager nodes. In this scenario, the highly-available replicated cluster store is working against you — quickly propagating the delete operation. Your only recovery options are to either recreate the deleted objects from copies in a vault or source code repo, or attempt a recovery from a recent backup.

Managing your swarm and applications declaratively is a great way to prevent the need to recover from a backup. For example, storing configuration objects outside of the swarm in a version control repository will give you the option to redeploy things like networks, services, secrets, and other objects.

Anyway, let’s see how to **backup a swarm**.

Backing up a Swarm

Swarm configuration and state is stored in `/var/lib/docker/swarm` on every manager node. This Raft log keys, overlay networks, Secrets, Configs, Services, and more. A *swarm backup* is a copy of all the files in this directory.

As the contents of the directory are replicated to all managers, it’s possible to perform backups from multiple managers. However, as you have to stop the Docker daemon as

part of the procedure, it's a good idea to perform the backup from a non-leader manager. This is because stopping Docker on the leader will initiate a leader election. You should also perform the backup at a quiet time for the business, as stopping managers increases the risk of the swarm losing quorum if another manager fails during the backup.

Create the following network before starting the backup. We'll check for this in a later step after performing the recovery.

```
$ docker network create -d overlay unimatrix01
```

The procedure we're about to follow is a high-risk procedure and for demonstration purposes only. You'll need to tweak it for your production environment. You may also have to prefix commands with `sudo`.

1. Stop Docker on a non-leader manager.

If you have any containers or service replicas running on the node, this action may stop them. However, if you've been following along, your manager nodes won't be running any application containers.

If you locked your swarm, be sure to have a copy of the swarm unlock key.

```
$ service docker stop
```

2. Backup the Swarm config.

This example uses the Linux `tar` utility to perform the file copy that will be the backup. Feel free to use a different tool.

```
$ tar -czvf swarm.bkp /var/lib/docker/swarm/
tar: Removing leading `/' from member names
/var/lib/docker/swarm/
/var/lib/docker/swarm/docker-state.json
/var/lib/docker/swarm/state.json
<Snip>
```

3. Verify the backup file exists.

```
$ ls -l
-rw-r--r-- 1 root  root  450727 May 22 12:34 swarm.bkp
```

In the real world, you should store and rotate this backup in line with any corporate backup retention policies.

At this point, the swarm is backed up and you can restart Docker on the node.

4. Restart Docker.

```
$ service docker restart
```

5. Unlock the Swarm to admit the restarted manager. You will only have to perform this step if your Swarm is locked. If you can't remember your Swarm unlock key, run a `docker swarm unlock-key` command on a different manager.

```
$ docker swarm unlock
```

Please enter unlock key:

Recovering a Swarm

Restoring a Swarm from backup is only for situations where the swarm is corrupted, or otherwise lost, and you cannot recover objects from copies of config files.

You'll need the `swarm.bkp` and a copy of your swarm's unlock key (if your swarm is locked).

The following requirements must be met for a recovery operation to work:

1. You can only restore to a node running the same version of Docker the backup was performed on
2. You can only restore to a node with the same IP address as the node the backup was performed on

Perform the operation on the manager you performed the backup from. You may need to prefix commands with `sudo`.

1. Stop Docker on the manager.

```
$ service docker stop
```

2. Delete the Swarm config.

```
$ rm -r /var/lib/docker/swarm
```

At this point, the manager is down and ready for the restore operation.

3. Restore the Swarm configuration from the backup file and verify the files recover properly.

In this example, we'll restore from a zipped tar file called `swarm.bkp`. Restoring to the root directory is required, as backup includes the full path to the original files. This may be different in your environment.


```
$ tar -zxvf swarm.bkp -C /
```

```
$ ls /var/lib/docker/swarm
certificates  docker-state.json  raft  state.json  worker
```

4. Start Docker.

```
$ service docker start
```

5. Unlock your Swarm with your Swarm unlock key.

```
$ docker swarm unlock
Please enter unlock key: <your key>
```

6. Initialize a new swarm with the configuration from the backup. Be sure to use the appropriate IP address for the node you're performing the restore operation on.

```
$ docker swarm init --force-new-cluster \
  --advertise-addr 10.0.0.1:2377 \
  --listen-addr 10.0.0.1:2377
```

```
Swarm initialized: current node (jhsg...3l9h) is now a manager.
```

7. Check that the unimatrix01 network was recovered as part of the operation.

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
z21s5v82by8q	unimatrix01	overlay	swarm

Congratulations. The Swarm is recovered.

8. Add new managers and workers and take fresh backups.

Remember to test this procedure regularly and thoroughly. You do not want it to fail when you need it most!

Docker Swarm - The Commands

- `docker swarm init` is the command to create a new swarm. The node that you run the command on becomes the first manager and is switched to run in *swarm mode*.
- `docker swarm join-token` reveals the commands and tokens needed to join workers and managers to a swarm. To reveal the command to join a new manager, use the `docker swarm join-token manager` command. To get the command to join a worker, use the `docker swarm join-token worker` command. Be sure to keep your join tokens secure!

- `docker node ls` lists all nodes in the swarm, including which are managers and which is the leader.
- `docker service create` is the command to create a new service.
- `docker service ls` lists running services and gives basic info on the state of the service and any replicas it's running.
- `docker service ps <service>` gives more detailed information about individual service replicas.
- `docker service inspect` gives very detailed information on a service. It accepts the `--pretty` flag to return only the most important information.
- `docker service scale` lets you scale the number of replicas in a service up and down.
- `docker service update` lets you update many of the properties of a running service.
- `docker service logs` lets you view the logs of a service.
- `docker service rm` is the command to delete a service from the swarm. Use it with caution as it deletes all service replicas without asking for confirmation.

Chapter summary

Docker Swarm is Docker's native technology for managing clusters of Docker nodes and orchestrating microservices applications. It is similar to Kubernetes but easier to use.

At its core, Swarm has a secure clustering component, and an orchestration component.

The clustering component is enterprise-grade and offers a wealth of security and HA features that are automatically configured and extremely simple to modify.

The orchestration component allows you to deploy and manage cloud-native microservices applications in a simple declarative manner.

We'll dig deeper into deploying cloud-native microservices apps Chapter 14.

11: Docker Networking

It's always the network!

Any time there's an infrastructure problem, we always blame the network. Part of the reason is that networks are at the center of everything — **no network, no app!**

In the early days of Docker, networking was hard. These days, it's *almost* a pleasure ;-)

In this chapter, we'll look at the fundamentals of Docker networking. Things like the Container Network Model (CNM) and `libnetwork`. We'll also get our hands dirty building and testing networks.

As usual, we'll split the chapter into three parts:

- The TLDR
- The deep dive
- The commands

Docker Networking - The TLDR

Docker runs applications inside of containers, and applications need to communicate with other application. Some of these other applications are containers and some aren't. This means Docker needs strong networking capabilities.

Fortunately, Docker has solutions for container-to-container networks, as well as connecting to existing networks and VLANs. The latter is important for containerized apps that interact with external services such as VM's and physical servers.

Docker networking is based on an open-source pluggable architecture called the Container Network Model (CNM). `libnetwork` is the reference implementation of the CNM, and it provides all of Docker's core networking capabilities. Drivers plug-in to `libnetwork` to provide specific network topologies.

To create a smooth out-of-the-box experience, Docker ships with a set of native drivers for the most common networking requirements. These include single-host bridge networks, multi-host overlays, and options for plugging into existing VLANs. Ecosystem partners can extend things further by providing their own drivers.

Last but not least, `libnetwork` provides native service discovery and basic container load balancing.

That's this big picture. Let's get into the detail.

Docker Networking - The Deep Dive

We'll organize this section of the chapter as follows:

- The theory
- Single-host bridge networks
- Multi-host overlay networks
- Connecting to existing networks
- Service Discovery
- Ingress load balancing

The theory

At the highest level, Docker networking comprises three major components:

- The Container Network Model (CNM)
- Libnetwork
- Drivers

The CNM is the design specification and outlines the fundamental building blocks of a Docker network.

Libnetwork is a real-world implementation of the CNM. It's open-sourced as part of the Moby project and used by Docker and other projects.

Drivers extend the model by implementing specific network topologies such as VXLAN overlay networks.

Figure 11.1 shows how they fit together at a very high level.

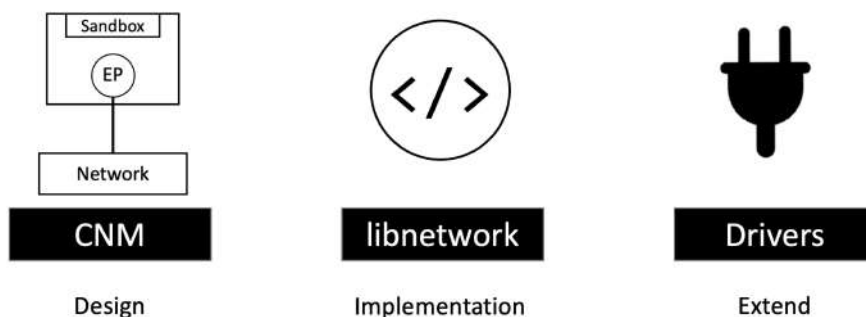


Figure 11.1

Let's look a bit closer at each.

The Container Network Model (CNM)

Everything starts with a design.

The design guide for Docker networking is the CNM. It outlines the fundamental building blocks of a Docker network, and you can read the full spec here:

- <https://github.com/docker/libnetwork/blob/master/docs/design.md>

I recommend reading the entire spec, but at a high level, it defines three building blocks:

- Sandboxes
- Endpoints
- Networks

A **sandbox** is an isolated network stack in a container. It includes Ethernet interfaces, ports, routing tables, and DNS config.

Endpoints are virtual network interfaces (E.g. veth). Like normal network interfaces, these are responsible for making connections. For example, *endpoints* to connect *sandboxes* to *networks*.

Networks are a software implementation of a switch (802.1d bridge). As such, they group together and isolate a collection of endpoints that need to communicate.

Figure 11.2 shows the three components and how they connect.

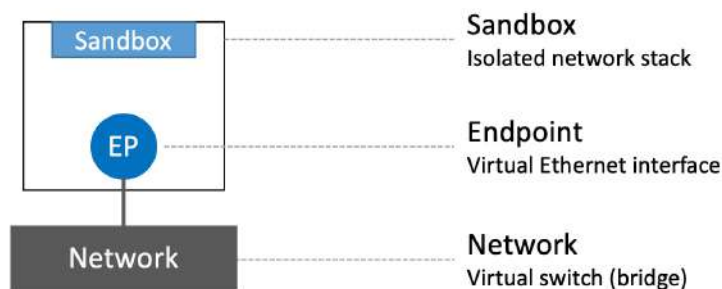


Figure 11.2 The Container Network Model (CNM)

The atomic unit of scheduling on Docker is the container, and as the name suggests, the Container Network Model is all about providing networking for containers. Figure 11.3 shows how CNM components relate to containers — sandboxes are placed inside of containers to provide network connectivity.

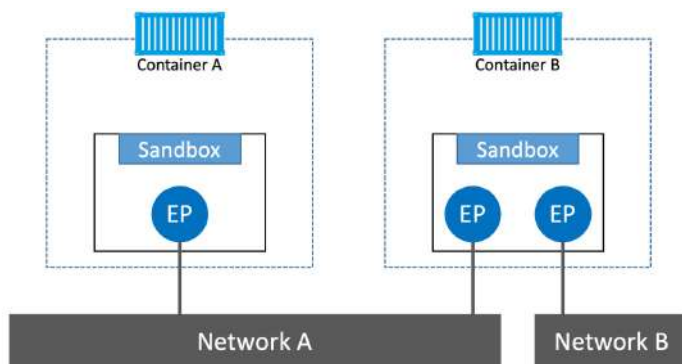


Figure 11.3

Container A has a single interface (endpoint) and is connected to Network A. Container B has two interfaces (endpoints) and is connected to Network A **and** Network B. The two containers can communicate because they are both connected to Network A. However, the two *endpoints* in Container B cannot communicate with each other without the assistance of a layer 3 router.

It's also important to understand that *endpoints* behave like regular network adapters, meaning they can only be connected to a single network. Therefore, a container needing to connect to multiple networks will need multiple endpoints.

Figure 11.4 extends the diagram again, this time adding a Docker host. Although Container A and Container B are running on the same host, their network stacks are completely isolated at the OS-level via the sandboxes and can only communicate via a network.

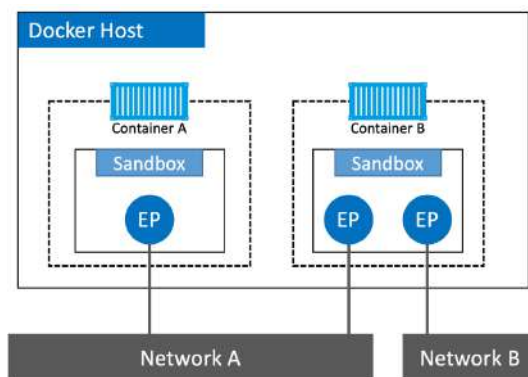


Figure 11.4

Libnetwork

The CNM is the design doc and `libnetwork` is the canonical implementation. It's open-source, cross-platform (Linux and Windows), lives in the Moby project, and used by Docker.

In the early days of Docker, all the networking code existed inside the daemon. This was a nightmare — the daemon became bloated, and it didn't follow the Unix principle of building modular tools that can work on their own, but also be easily composed into other projects. As a result, the network code got ripped out and refactored into an external library called `libnetwork` based on the principles of the CNM. Today, all of the core Docker networking code lives in `libnetwork`.

As well as implementing the core components of the CNM, `libnetwork` also implements native *service discovery*, *ingress-based container load balancing*, and the network control plane and management plane.

Drivers

If `libnetwork` implements the control plane and management plane, then drivers implement the data plane. For example, connectivity and isolation is all handled by drivers. So is the creation of networks. The relationship is shown in Figure 11.5.

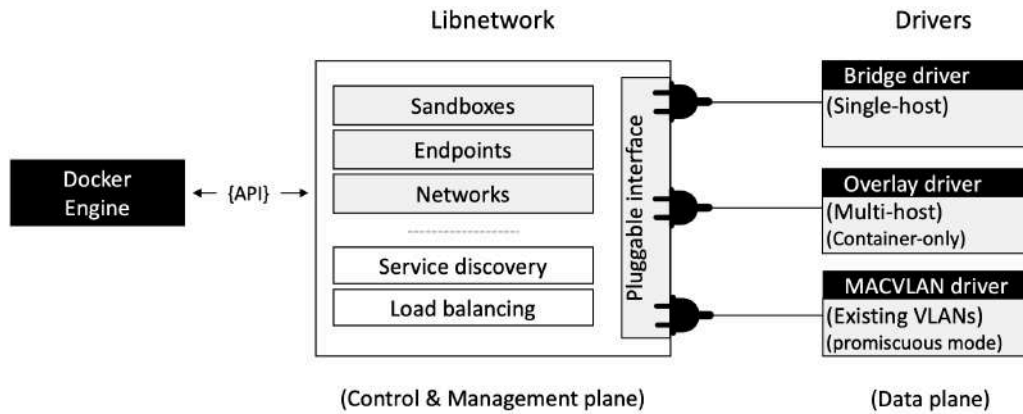


Figure 11.5

Docker ships with several built-in drivers, known as native drivers or *local drivers*. These include bridge, overlay, and macvlan, and they build the most common network topologies. 3rd-parties can also write network drivers to implement other network topologies and more advanced configurations.

Every network is owned by a driver, and the driver is responsible for the creation and management of all resources on the network. For example, an overlay network called “prod-fe-cuda” will be owned and managed by the overlay driver. This means the overlay driver is invoked for the creation, management, and deletion of all resources on that network.

In order to meet the demands of complex highly-fluid environments, libnetwork allows multiple network drivers to be active at the same time. This means your Docker environment can sport a wide range of heterogeneous networks.

Let’s look a bit closer at single-host bridge networks, multi-host overlay network, and connecting to existing networks...

Single-host bridge networks

The simplest type of Docker network is the single-host bridge network.

The name tells us two things:

- **Single-host** tells us it only spans a single Docker host and can only connect containers that are on the same host.
- **Bridge** tells us that it’s an implementation of an 802.1d bridge (layer 2 switch).

Docker on Linux creates single-host bridge networks with the built-in bridge driver, whereas Docker on Windows creates them using the built-in nat driver. For all intents and purposes, they work the same.

Figure 11.6 shows two Docker hosts with identical local bridge networks called “mynet”. Even though the networks are identical, they are independent and isolated. This means the containers in the picture cannot communicate because they are on different networks.

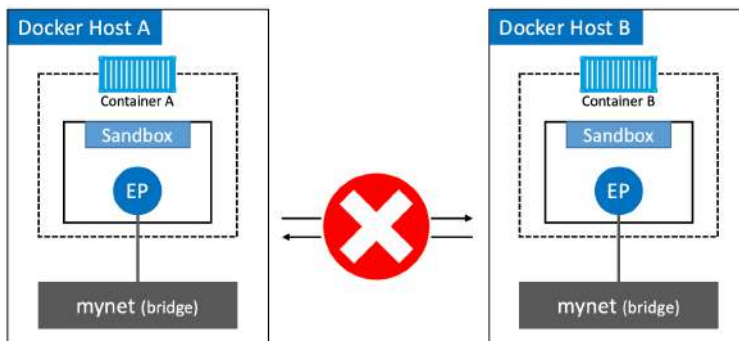


Figure 11.6

Every Docker host gets a default single-host bridge network. On Linux it’s called “bridge” and on Windows it’s called “nat” (it’s just a coincidence that they have the same name as the drivers used to create them). By default, all new containers will be attached to these networks unless you override it on the command line with the `--network` flag.

The following commands show the output of a `docker network ls` command on newly installed Linux and Windows Docker hosts. The output is trimmed so that it only shows the default network on each host. Notice how the name of the network is the same as the driver that was used to create it — this is a coincidence and not a requirement.

```
//Linux
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
333e184cd343        bridge             bridge              local

//Windows
> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
095d4090fa32        nat                nat                 local
```

As always, the `docker inspect` command is a treasure trove of great information. I highly recommend reading through its output if you’re interested in low-level detail.

```
$ docker inspect bridge
[
  {
    "Name": "bridge",      << Will be nat on Windows
    "Id": "333e184...d9e55",
    "Scope": "local",
    "Driver": "bridge",    << Will be nat on Windows
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "172.17.0.0/16"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    <Snip>
  }
]
```

Docker networks built with the `bridge` driver on Linux hosts are based on the battle-hardened *linux bridge* technology that has existed in the Linux kernel for 20 years. This means they're high performance and extremely stable. It also means you can inspect them using standard Linux utilities. For example.

```
$ ip link show docker0
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc...
    link/ether 02:42:af:f9:eb:4f brd ff:ff:ff:ff:ff:ff
```

The default “bridge” network, on all Linux-based Docker hosts, maps to an underlying *Linux bridge* in the kernel called “**docker0**”. We can see this from the output of `docker inspect`.

```
$ docker inspect bridge | grep bridge.name
"com.docker.network.bridge.name": "docker0",
```

Figure 11.7 shows containers connecting to the “bridge” network. The “bridge” network maps to the “docker0” Linux bridge in the host’s kernel, which maps to an Ethernet interface on the host via port mappings.

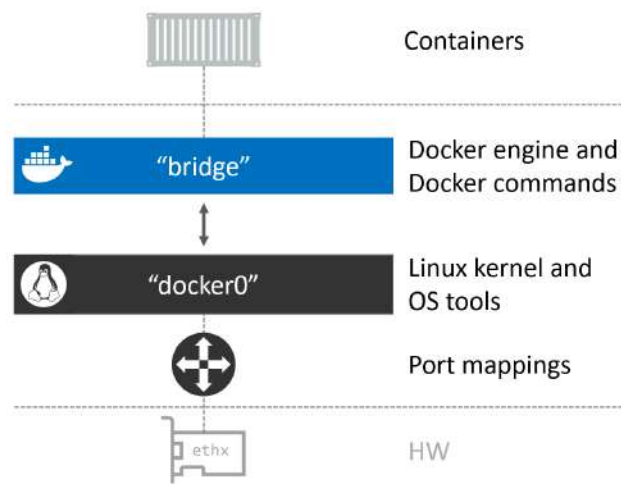


Figure 11.7

Let's use the `docker network create` command to create a new single-host bridge network called "localnet".

```
$ docker network create -d bridge localnet
```

The new network is created and will appear in the output of any future `docker network ls` commands. You'll also have a new *Linux bridge* created in the kernel.

Let's use the Linux `brctl` tool to look at the Linux bridges currently on our system. You may have to install the `brctl` binary using `apt-get install bridge-utils`, or the equivalent for your Linux distro.

```
$ brctl show
bridge name      bridge id        STP enabled    interfaces
docker0          8000.0242aff9eb4f  no
br-20c2e8ae4bbb  8000.02429636237c  no
```

The output shows two bridges. The first line is the "docker0" bridge that we already know about. The second bridge (br-20c2e8ae4bbb) relates to the new `localnet` bridge network just created. Neither of them have spanning tree enabled, and neither have any devices connected (`interfaces` column).

At this point, the bridge configuration on the host looks like Figure 11.8.

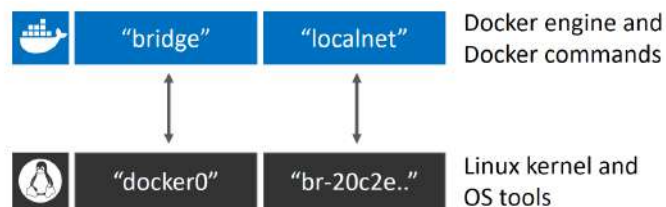


Figure 11.8

Let's create a new container and attach it to the new `localnet` bridge network.

```
$ docker run -d --name c1 \
  --network localnet \
  alpine sleep 1d
```

This container will be attached to the `localnet` network. Confirm this with a `docker inspect`. If your output isn't formatted properly, you can try piping it through `jq`. You'll obviously need `jq` installed on your system.

```
$ docker inspect localnet --format '{{json .Containers}}'
{
  "4edcbd...842c3aa": {
    "Name": "c1",
    "EndpointID": "43a13b...3219b8c13",
    "MacAddress": "02:42:ac:14:00:02",
    "IPv4Address": "172.20.0.2/16",
    "IPv6Address": ""
  }
},
```

The output shows that the new `"c1"` container is on the `localnet` bridge/nat network.

If you run another `brctl show` you'll see `c1`'s interface attached to the `br-20c2e8ae4bbb` bridge.

```
$ brctl show
bridge name      bridge id        STP enabled      interfaces
br-20c2e8ae4bbb  8000.02429636237c  no               vethe792ac0
docker0          8000.0242aff9eb4f  no
```

This is shown in Figure 11.9.

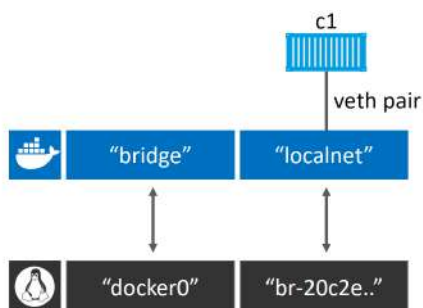


Figure 11.9

If we add another new container to the same network, it will be able to ping the “c1” container by name. This is because all containers automatically register with the embedded Docker DNS service, allowing them to resolve the names of all other containers on the same network.

Beware: The **default** bridge network, the one called “bridge”, doesn’t support name resolution via the Docker DNS service. All other *user-defined* bridge networks do. The following demo will work because the container is on the user-defined `localnet` network.

Let’s test it.

1. Create a new interactive container called “c2” and put it on the same `localnet` network as “c1”.

```
$ docker run -it --name c2 \
  --network localnet \
  alpine sh
```

Your terminal will switch into the “c2” container.

2. From within the “c2” container, ping the “c1” container by name.

```
> ping c1
Ping: c1 [172.26.137.130] with 32 bytes of data:
Reply from 172.26.137.130: bytes=32 time=1ms TTL=128
Reply from 172.26.137.130: bytes=32 time=1ms TTL=128
Control-C
```

It works! This is because the c2 container is running a local DNS resolver that forwards requests to the internal Docker DNS server. This DNS server maintains mappings for all containers started with the `--name` or `--net-alias` flag.

Try running some network-related commands while you're still logged on to the container. It's a great way of learning more about how Docker networking works. You might have to manually install your favourite networking tools to do this.

So far, we've said that containers on bridge networks can only communicate with other containers on the same network. However, you can get around this using *port mappings*.

Port mappings let you map a container to a port on the Docker host. Any traffic hitting the Docker host on the configured port will be re-directed to the container. The high-level flow is shown in Figure 11.10

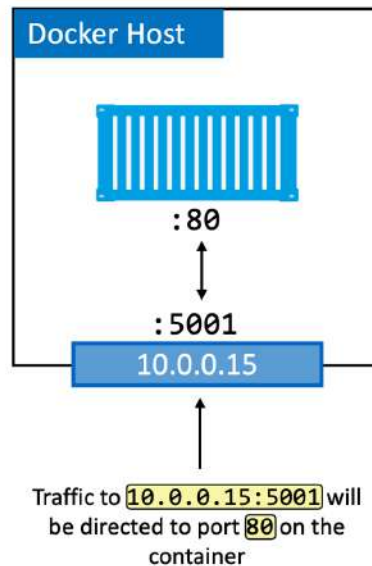


Figure 11.10

In the diagram, the application running in the container is operating on port 80. This is mapped to port 5001 on the host's 10.0.0.15 interface. The result is all traffic hitting the host on 10.0.0.15:5001 being redirected to the container on port 80.

Let's walk through an example of mapping port 80 on a container running a web server, to port 5001 on the Docker host. The example will use NGINX on Linux.

1. Run a new NGINX web server container and map port 80 to 5001 on the Docker host.

```
$ docker run -d --name web \
  --network localnet \
  --publish 5001:80 \
  nginx
```

2. Verify the port mapping.

```
$ docker port web
80/tcp -> 0.0.0.0:5001
80/tcp -> [::]:5001
```

This shows the port mapping exists on all interfaces on the Docker host.

3. Test the configuration by pointing a web browser to port 5001 on the Docker host. To complete this step, you'll need to know the IP or DNS name of your Docker host. If you're using Docker Desktop, you'll be able to use `localhost:5001` or `127.0.0.1:5001`.

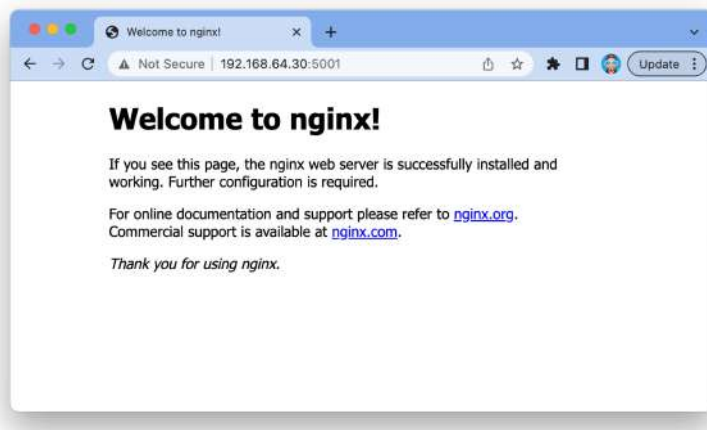


Figure 11.11

Any external system can now access the NGINX container (running on the `localnet` bridge network) by hitting the Docker host on port 5001.

Mapping ports like this works, but it's clunky and doesn't scale. For example, only a one container can bind to any particular port on the host. In our example, no other containers will be able to bind to port 5001. This is one of the reason's that single-host bridge networks are only useful for local development and very small applications.

Multi-host overlay networks

The next chapter is dedicated to multi-host overlay networks. So we'll keep this section short.

Overlay networks are multi-host. This means a single network can span every node in a swarm, allowing containers on different hosts to communicate. They're great for container-to-container communication and they scale well.

Docker provides a native driver for overlay networks. This makes creating them as simple as adding the `-d overlay` flag to the `docker network create` command.

We'll dive into lots of examples in the next chapter.

Connecting to existing networks

The ability to connect containerized apps to external systems and physical networks is important. A common example is partially containerized apps — the containerized parts need a way to communicate with the parts still running on existing physical networks and VLANs.

The built-in MACVLAN driver (transparent on Windows) was created with this in mind. It gives each container its own IP and MAC address on the external physical network, making them look just like a physical server or VM. This is shown in Figure 11.12.

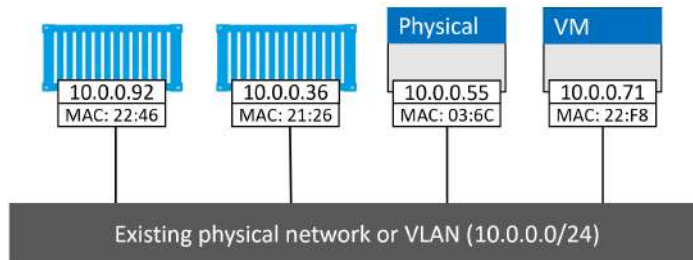


Figure 11.12

On the positive side, MACVLAN performance is good as it doesn't require port mappings or additional bridges. However, it requires the host NIC to be in **promiscuous mode**, which isn't allowed on many corporate networks and public cloud platforms. So... MACVLAN is great for your data center networks *if* your network team allows *promiscuous mode*, but it probably won't work on your public cloud.

Let's dig a bit deeper with the help of some pictures and a hypothetical example. This example will work if your host NIC is in promiscuous mode on a network that allows it. It also requires an existing VLAN 100 on the network. You can adapt it if the VLAN config on your physical network is different.

Assume we have an existing physical network with two VLANs:

- VLAN 100: 10.0.0.0/24
- VLAN 200: 192.168.3.0/24



Figure 11.13

Next, we add a Docker host and connect it to the network.

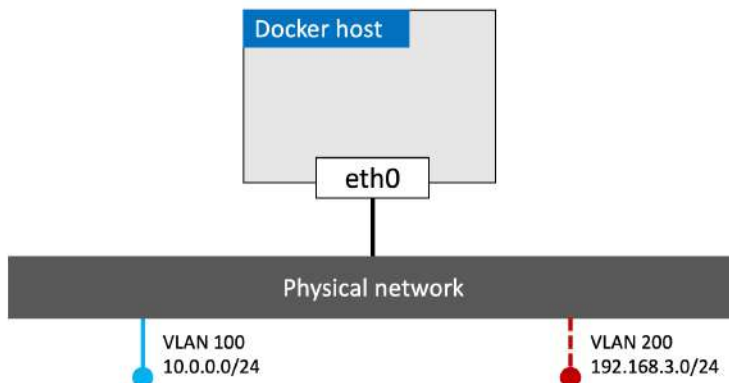


Figure 11.14

Then comes a requirement for a container running on that host to be on VLAN 100. To do this, we create a new Docker network with the `macvlan` driver. However, the `macvlan` driver needs us to tell it a few things about the network we're plumbing it into. Things like:

- Subnet info
- Gateway
- Range of IP's it can assign to containers
- Which interface or sub-interface on the host to use

The following command will create a new MACVLAN network called "macvlan100" that will connect containers to VLAN 100. You may have to change the parent interface name from `eth0` to match the parent interface name on your system, such as `enp0s1`. For example, changing `-o parent=eth0.100` to `-o parent=enp0s1.100`.

```
$ docker network create -d macvlan \  
  --subnet=10.0.0.0/24 \  
  --ip-range=10.0.0.0/25 \  
  --gateway=10.0.0.1 \  
  -o parent=eth0.100 \  
  macvlan100
```

This will create the “macvlan100” network and the eth0.100 sub-interface. The config now looks like this.

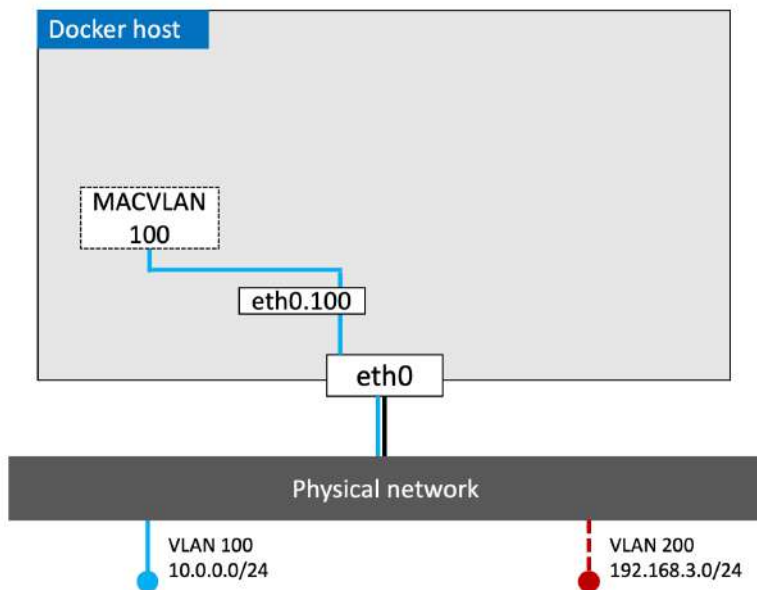


Figure 11.15

MACVLAN uses standard Linux *sub-interfaces*, and you tag them with the ID of the VLAN they will connect to. In this example, we’re connecting to VLAN 100, so we tag the sub-interface with `.100` (`-o parent=eth0.100`).

We also used the `--ip-range` flag to tell the MACVLAN network which sub-set of IP addresses it can assign to containers. It’s vital that this range of addresses is reserved for Docker and not in use by other nodes or DHCP servers as the MACVLAN driver has no management plane feature to check for overlapping IP ranges.

The `macvlan100` network is ready for containers, so let’s deploy one with the following command.

```
$ docker run -d --name mactainer1 \
  --network macvlan100 \
  alpine sleep 1d
```

The config now looks like Figure 11.16. But remember, the underlying network (VLAN 100) does not see any of the MACVLAN magic, it only sees the container with its MAC and IP addresses. This means the “mactainer1” container will be able to ping and communicate with any other systems on VLAN 100. Pretty sweet!

Note: If you can’t get this to work, it might be because the host NIC is not in promiscuous mode. Remember that public cloud platforms don’t usually allow promiscuous mode.

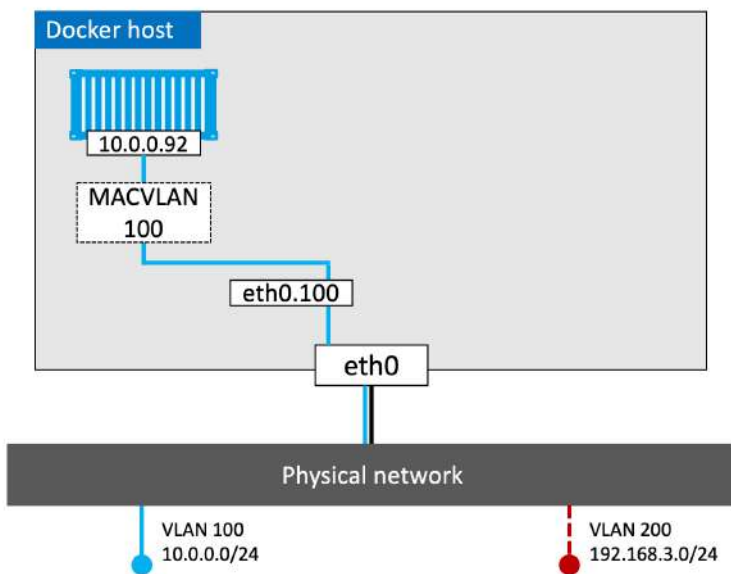


Figure 11.16

At this point, we’ve got a MACVLAN network and used it to connect a new container to an existing VLAN. However, it doesn’t stop there. The Docker MACVLAN driver is built on top of the tried-and-tested Linux kernel driver with the same name. As such, it supports VLAN trunking. This means we can create multiple MACVLAN networks and connect containers on the same Docker host to them as shown in Figure 11.17.

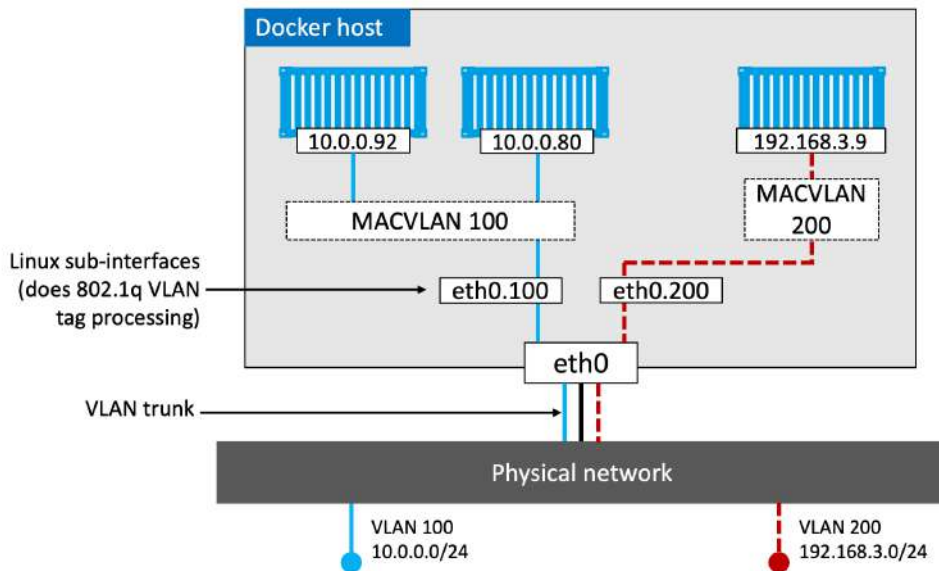


Figure 11.17

Container and Service logs for troubleshooting

A quick note on troubleshooting connectivity issues before moving on to Service Discovery.

If you think you're experiencing connectivity issues between containers, it's worth checking the Docker daemon logs as well as container logs.

On Windows systems, daemon logs are stored under `~AppData\Local\Docker` and you can view them in the Windows Event Viewer. On Linux, it depends which *init* system you're using. If you're running a *systemd*, logs will go to *journald* and you can view them with the `journalctl -u docker.service` command. If you're not running *systemd* you should look under the following locations:

- Ubuntu systems running *upstart*: `/var/log/upstart/docker.log`
- RHEL-based systems: `/var/log/messages`
- Debian: `/var/log/daemon.log`

You can also tell Docker how verbose you want daemon logging to be. To do this, edit the daemon config file (`/etc/docker/daemon.json`) so that "debug" is set to "true" and "log-level" is set to one of the following:

- `debug` The most verbose option
- `info` The default value and second-most verbose option
- `warn` Third most verbose option
- `error` Fourth most verbose option
- `fatal` Least verbose option

The following snippet from a `daemon.json` enables debugging and sets the level to `debug`. It will work on all Docker platforms.

```
{
  <Snip>
  "debug":true,
  "log-level":"debug",
  <Snip>
}
```

If the `daemon.json` file doesn't exist, create it! Also, be sure to restart Docker after making changes to the file.

That was the `daemon logs`. What about container logs?

Logs from standalone containers can be viewed with the `docker logs` command, and Swarm service logs can be viewed with the `docker service logs` command. However, Docker supports lots of logging drivers and they don't all work with the Docker log commands.

As well as a driver and configuration for `daemon logs`, every Docker host has a default logging driver and configuration for containers. Some of the drivers include:

- `json-file` (default)
- `journald` (only works on Linux hosts running `systemd`)
- `syslog`
- `splunk`
- `gelf`

`json-file` and `journald` are probably the easiest to configure and they both work with `docker logs` and `docker service logs`.

If you're using other logging drivers, you can view logs using the 3rd-party platform's native tools.

The following snippet from a `daemon.json` shows a Docker host configured to use `syslog`.

```
{
  "log-driver": "syslog"
}
```

You can configure an individual container, or service, to start with a particular logging driver with the `--log-driver` and `--log-opts` flags. These will override anything set in `daemon.json`.

Container logs work on the premise that your application is running as PID 1 inside the container, sending logs to `STDOUT`, and sending errors to `STDERR`. The logging driver then forwards these “logs” to the locations configured via the logging driver.

The following is an example of running the `docker logs` command against a container called “vantage-db” configured to use the `json-file` logging driver.

```
$ docker logs vantage-db
1:C 2 Feb 09:53:22.903 # o000o000o000o Redis is starting o000o000o000o
1:C 2 Feb 09:53:22.904 # Redis version=4.0.6, bits=64, commit=00000000, modified=0, pid=1
1:C 2 Feb 09:53:22.904 # Warning: no config file specified, using the default config.
1:M 2 Feb 09:53:22.906 * Running mode=standalone, port=6379.
1:M 2 Feb 09:53:22.906 # WARNING: The TCP backlog setting of 511 cannot be enforced because...
1:M 2 Feb 09:53:22.906 # Server initialized
1:M 2 Feb 09:53:22.906 # WARNING overcommit_memory is set to 0!
```

There’s a good chance you’ll find network connectivity errors reported in the `daemon logs` or container logs.

Service discovery

As well as core networking, `libnetwork` also provides important network services.

Service discovery allows all containers and Swarm services to locate each other by name. The only requirement is that they be on the same network.

Under the hood, this leverages Docker’s embedded DNS server and the DNS resolver in each container. Figure 11.18 shows container “c1” pinging container “c2” by name. The same principle applies to Swarm Services.

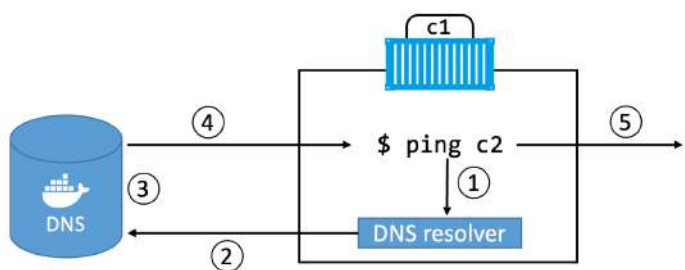


Figure 11.18

Let's step through the process.

- **Step 1:** The `ping c2` command invokes the local DNS resolver to resolve the name "c2" into an IP address. All Docker containers have a local DNS resolver.
- **Step 2:** If the local resolver doesn't have an IP address for "c2" in its local cache, it initiates a recursive query to the Docker DNS server. The local resolver is pre-configured to know how to reach the Docker DNS server.
- **Step 3:** The Docker DNS server holds name-to-IP mappings for all containers created with the `--name` or `--net-alias` flags. This means it knows the IP address of container "c2".
- **Step 4:** The DNS server returns the IP address of "c2" to the local resolver in container "c1". It does this because the two containers are on the same network — if they were on different networks this would not work.
- **Step 5:** The ping command issues the ICMP echo request packets to the IP address of "c2".

Every Swarm service and standalone container started with the `--name` flag will register its name and IP with the Docker DNS service. This means all containers and service replicas can use the Docker DNS service to find each other. However, service discovery is *network-scoped*, meaning name resolution only works for containers and Services on the same network. If two containers are on different networks, they will not be able to resolve each other.

One last point on service discovery and name resolution...

It's possible to configure Swarm services and standalone containers with customized DNS options. For example, the `--dns` flag lets you specify a list of custom DNS servers to use in case the embedded Docker DNS server cannot resolve a query. This is common when querying names of services outside of Docker. You can also use the `--dns-search` flag to add custom search domains for queries against unqualified names (i.e., when the query isn't a fully qualified DNS name).

This works on Linux by adding entries to the `/etc/resolv.conf` of every container.

The following example will start a new standalone container and add the infamous 8.8.8.8 Google DNS server, as well as `nigelpoulton.com` as search domain for unqualified queries. Do not run this command, it's just to show you how the options look.

```
$ docker run -it --name c1 \  
  --dns=8.8.8.8 \  
  --dns-search=nigelpoulton.com \  
  alpine sh
```

Ingress load balancing

Swarm supports two network publishing modes that make services accessible outside of the cluster:

- Ingress mode (default)
- Host mode

Services published via *ingress mode* can be accessed from any node in the Swarm — even nodes **not** running a service replica. Services published via *host mode* can only be accessed by hitting nodes running service replicas. Figure 11.19 shows the difference between the two modes.

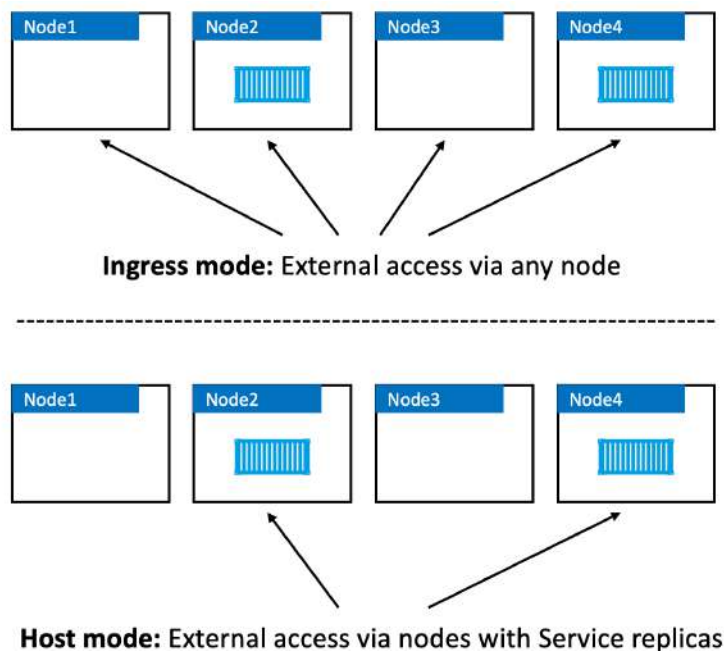


Figure 11.19

Ingress mode is the default. This means any time you publish a service with `-p` or `--publish` it will default to *ingress mode*. To publish a service in *host mode* you need to use the long format of the `--publish` flag **and** add `mode=host`. The following example uses host mode.

```
$ docker service create -d --name svc1 \
  --publish published=5001,target=80,mode=host \
  nginx
```

A few notes about the command. `docker service create` lets you publish a service using either a *long form syntax* or *short form syntax*. The short form looks like this: `-p 5001:80` and we've seen it a few times already. However, you cannot publish a service in *host mode* using short form.

Long form looks like this: `--publish published=5001,target=80,mode=host`. It's a comma-separated list with no whitespace after each comma. The options work as follows:

- `published=5001` makes the service available externally via port 5001
- `target=80` makes sure requests hitting the published port get mapped back to port 80 on the service replicas

- `mode=host` makes sure requests will only reach the service if they arrive on nodes running a service replica.

Ingress mode is what you'll normally use.

Behind the scenes, *ingress mode* uses a layer 4 routing mesh called the **service mesh** or the **swarm-mode service mesh**. Figure 11.20 shows the basic traffic flow of an external request hitting the cluster for a service exposed in ingress mode.

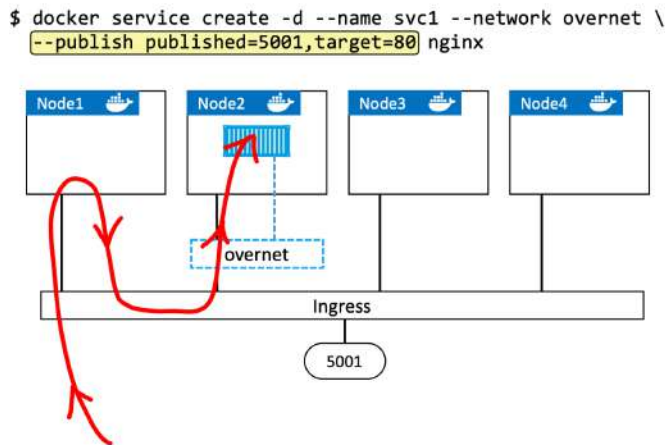


Figure 11.20

Let's quickly walk through the diagram.

1. The command at the top deploys a new Swarm service called "svc1". It's attaching the service to the overnet network and publishing it on port 5001.
2. Publishing a Swarm service like this (`--publish published=5001,target=80`) will publish it on the ingress network. As all nodes in a Swarm are attached to the ingress network meaning the port is published *swarm-wide*.
3. Logic is implemented on the cluster ensuring that any traffic hitting the ingress network, via **any node**, on 5001 will be routed to the "svc1" replicas on port 80.
4. At this point, a single replica for the "svc1" service is deployed.
5. The red line shows traffic hitting node1 on the published port and being routed to the service replica running on node2 via the ingress network.

It's vital to know that the incoming request can arrive on any of the four Swarm nodes we'll get the same result.

It's also important to know that if there are multiple replicas, as shown in Figure 11.21, traffic will be balanced across them all.

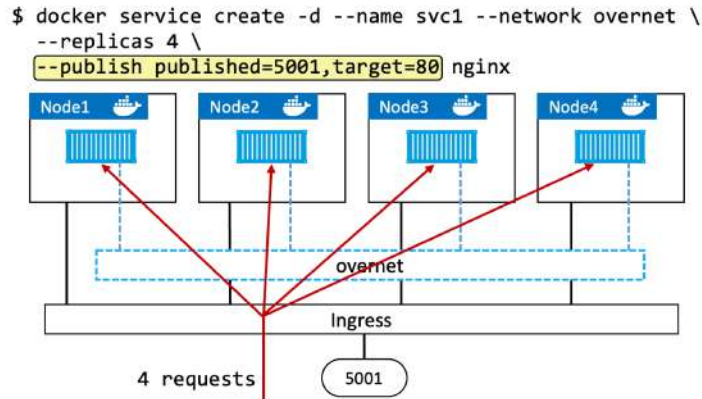


Figure 11.21

Docker Networking - The Commands

Docker networking has its own `docker network` sub-command. The main commands include:

- `docker network ls`: Lists all networks on the local Docker host.
- `docker network create`: Creates new Docker networks. By default, it creates them with the `nat` driver on Windows and the `bridge` driver on Linux. You can specify the driver (type of network) with the `-d` flag. `docker network create -d overlay overnet` will create a new overlay network called `overnet` with the native Docker overlay driver.
- `docker network inspect`: Provides detailed configuration information about a Docker network. Same as `docker inspect`.
- `docker network prune`: Deletes all unused networks on a Docker host.
- `docker network rm`: Deletes specific networks on a Docker host.

Chapter Summary

The Container Network Model (CNM) is the design document for Docker networking and defines the three major constructs that are used to build Docker networks — *sandboxes*, *endpoints*, and *networks*.

`libnetwork` is the reference implementation of the CMN. It's an open-source project that lives in the Moby project. It's used by Docker and is where all of the core Docker

networking code lives. It also provides network control plane and management plane services such as service discovery.

Drivers extend `libnetwork` by implementing specific network types such as bridge networks and overlay networks. Docker ships with built-in drivers, but you can also use 3rd-party drivers.

Single-host bridge networks are the most basic type of Docker network and are suitable for local development and very small applications. They do not scale and they require port mappings if you want to publish your services outside of the network.

Overlay networks are all the rage and are excellent container-only multi-host networks. We'll talk about them in-depth in the next chapter.

The `macvlan` driver allows us to connect containers to existing physical networks and VLANs. They make containers first-class citizens by giving them their own MAC and IP addresses. Unfortunately, they require promiscuous mode on the host NIC, meaning they won't work in the public cloud.

Docker also uses `libnetwork` to implement service discovery and an ingress routing mesh for container-based load balancing of ingress traffic.

12: Docker overlay networking

Overlay networks are at the center of most cloud-native microservices apps. In this chapter we'll get you up-to-speed with overlay networking on Docker.

Docker overlay networking on Windows has feature parity with Linux. This means the examples we'll use in this chapter will all work on Linux and Windows.

We'll split this chapter into the usual three parts:

- The TLDR
- The deep dive
- The commands

Let's do some networking magic.

Docker overlay networking - The TLDR

In the real world, it's vital that containers can communicate reliably and securely, even when they're on different hosts that are on different networks. This is where overlay networks come into play. They create a flat, secure, layer 2 networks spanning multiple hosts. Containers on different hosts can connect to the same overlay network and communicate directly.

Docker offers native overlay networking that is simple to configure and secure by default.

Behind the scenes, it's built on top of `libnetwork` and the native **overlay** driver. `Libnetwork` is the canonical implementation of the Container Network Model (CNM) and the overlay driver implements all of the network machinery.

Docker overlay networking - The deep dive

In March 2015, Docker, Inc. acquired a container networking startup called *Socket Plane*. Two of the reasons behind the acquisition were to bring *real networking* to Docker, and to make container networking simple enough that even developers could do it.

They over-achieved on both, and overlay networking continues to be at the heart of container networking in 2023 and for the foreseeable future.

However, hiding behind a few simple networking commands is a lot of complexity. The kind of stuff you need to understand before doing production deployments and attempting to troubleshoot issues.

The rest of this section will be broken into two parts:

- Building and testing Docker overlay networks
- Overlay networks explained

Building and testing Docker overlay networks

The following examples will use two Docker nodes configured as a swarm. The nodes are on two separate networks connected by a router.

If you're following along, it's not vital that the nodes are on separate networks connected by a router, but they can be. All that is required is that both nodes are running Docker, have network connectivity, and can be configured into a swarm. This means you can follow along on Play with Docker, a couple of Multipass VMs on your local machine, or in the public cloud.

Following along on Docker Desktop is possible, but you won't get the full experience as you'll only have access to a single node.

The initial configuration is shown in See Figure 12.1. Everything will work if your nodes are on the same network, it just means your *underlay network* is simpler. We'll explain underlay networks later.

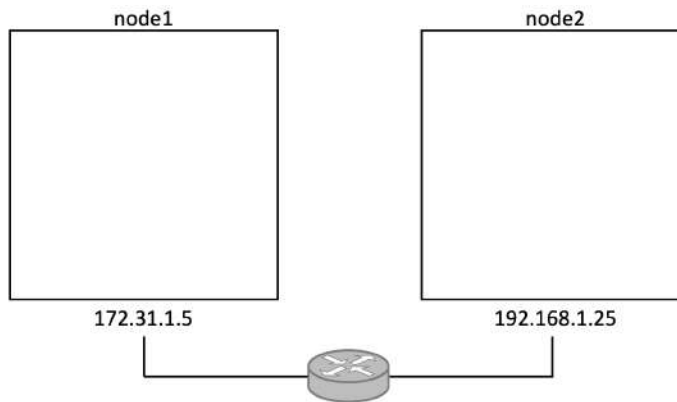


Figure 12.1

Build a Swarm

The first thing to do is configure the two nodes into a swarm. This is because swarm mode is a pre-requisite for Docker overlay networks.

We'll run a `docker swarm init` command on **node1** to make it a *manager*, then we'll run a `docker swarm join` command on **node2** to make it a *worker*. This isn't a production-grade setup, but it is enough for a learning lab. You're encouraged to test with more managers and workers and expand on the examples.

If you're following along in your own lab, you'll need to substitute the IP addresses and names with the correct values for your environment. You'll also need to ensure nothing is blocking the following ports between the two nodes:

- 2377/tcp for management plane comms
- 7946/tcp and 7946/udp for control plane comms (SWIM-based gossip)
- 4789/udp for the VXLAN data plane

Run the following command on **node1**.

```
$ docker swarm init \
  --advertise-addr=172.31.1.5 \
  --listen-addr=172.31.1.5:2377
```

Swarm initialized: current node (1ex3...o3px) is now a manager.

Copy the `docker swarm join` command included in the output and paste it into a terminal on **node2**.

```
$ docker swarm join \
  --token SWMTKN-1-0hz2ec...2vye \
  172.31.1.5:2377
This node joined a swarm as a worker.
```

We now have a two-node Swarm with **node1** as a manager and **node2** as a worker.

Create a new overlay network

Let's create a new *overlay network* called **uber-net**.

Run the following command from **node1** (manager).


```
$ docker network create -d overlay uber-net
c740ydi1lm89khn5kd52skrd9
```

That's it. You've just created a brand-new overlay network that's available to all hosts in the swarm and has its control plane encrypted with TLS (AES in GCM mode with keys automatically rotated every 12 hours). If you want to encrypt the data plane, you just add the `-o encrypted` flag to the command. However, data plane encryption isn't enabled by default because of the performance overhead. Be sure to test performance before enabling data plane encryption in your production environments. However, if you do enable it, it's protected by the same AES in GCM mode with key rotation.

If you're unsure about terms such as *control plane* and *data plane*... control plane traffic is cluster management traffic, whereas data plane traffic is application traffic. By default, Docker overlay networks encrypt cluster management traffic but not application traffic. You must explicitly enable encryption of application traffic.

You can list all networks on each node with the `docker network ls` command.

```
$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
ddac4ff813b7	bridge	bridge	local
389a7e7e8607	docker_gwbridge	bridge	local
a09f7e6b2ac6	host	host	local
ehw16ycy980s	ingress	overlay	swarm
2b26c11d3469	none	null	local
c740ydi1lm89	uber-net	overlay	swarm

The newly created network is at the bottom of the list called **uber-net**. The other networks were automatically created when Docker was installed and when the swarm was initialized.

If you run the `docker network ls` command on **node2**, you'll notice that it doesn't show the **uber-net** network. This is because new overlay networks are only extended to worker nodes when the worker is tasked with running a container on the network. This lazy approach to extending overlay networks improves scalability by reducing the amount of network gossip.

Attach a service to the overlay network

Now that we have an overlay network, let's attach a new *Docker service* to it. The example will create the service with two replicas so that one runs on **node1** and the other runs on **node2**. This will automatically extend the **uber-net** overlay to **node2**.

Run the following commands from **node1**.

```
$ docker service create --name test \
  --network uber-net \
  --replicas 2 \
  ubuntu sleep infinity
```

The command creates a new service called **test** and attaches both replicas to the **uber-net** overlay network. Because we're running two replicas on a two-node swarm, one will be scheduled to each node.

Verify the operation with a `docker service ps` command.

```
$ docker service ps test
```

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE
77q...rkx	test.1	ubuntu	node1	Running	Running
97v...pa5	test.2	ubuntu	node2	Running	Running

Run a `docker network ls` on **node2** to verify it can now see the network.

Standalone containers that are not part of a *swarm service* cannot attach to overlay networks unless the network was created with the `attachable=true` property. The following command can be used to create an attachable overlay network that standalone containers can connect to.

```
$ docker network create -d overlay --attachable uber-net
```

Congratulations. You've created a new overlay network spanning two nodes on separate physical underlay networks. You've also attached two containers to it. How easy was that!

You'll fully appreciate the simplicity of what you've done when your head explodes in the theory section and you realise the outrageous complexity of what's going on behind the scenes!

Test the overlay network

Let's test the overlay network with the `ping` command.

As shown in Figure 12.2, we've got two Docker hosts on separate networks, and a single overlay network spanning both. We've also got a container connected to the overlay network on each node. Let's see if they can ping each other.

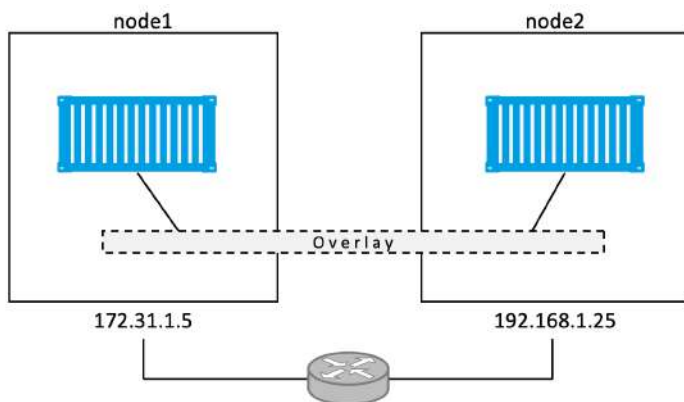


Figure 12.2

You can perform the test by pinging the remote container by name. However, the examples will use IP addresses as it gives us an excuse to learn how to find container IP addresses.

Run a `docker inspect` to see the subnet assigned to the overlay and the IP addresses assigned to the two `test` service replicas.

```
$ docker inspect uber-net
[
  {
    "Name": "uber-net",
    "Id": "c740yd1lm89khn5kd52skrd9",
    "Scope": "swarm",
    "Driver": "overlay",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "10.0.0.0/24",    <<---- Subnet info
          "Gateway": "10.0.0.1"    <<---- Subnet info
        }
      ]
    },
    "Containers": [
      {
        "Name": "test.1.mfd1kn0qzgosu2f6bhfk5jc2p",    <<---- Container name
        "IPv4Address": "10.0.0.3/24",                  <<---- Container IP
        <Snip>
      },
      {
        "Name": "test.2.m49f4psxp3daixlwfvy73v4j8",    <<---- Container name
        "IPv4Address": "10.0.0.4/24",                  <<---- Container IP
      },
    ]
  }
]
```

The output is heavily snipped for readability, but we can see it shows **uber-net**'s subnet is 10.0.0.0/24. This doesn't match either of the physical underlay networks shown in Figure 12.2 (172.31.1.0/24 and 192.168.1.0/24). You can also see the IP addresses assigned to the two containers.

Run the following two commands on both nodes. The first command gets the replica's container ID, the second gets the container's IP address. Be sure to use the container IDs from your own lab in the second command.

```
$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        NAME
396c8b142a85   ubuntu:latest  "sleep infinity"        2 hours ago   Up 2 hrs     test.1.mfd...

$ docker inspect \
  --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' 396c8b142a85
10.0.0.3
```

See how the names and IPs match the output from the `docker inspect` command.

Figure 12.3 shows the configuration so far. Subnet and IP addresses may be different in your lab.

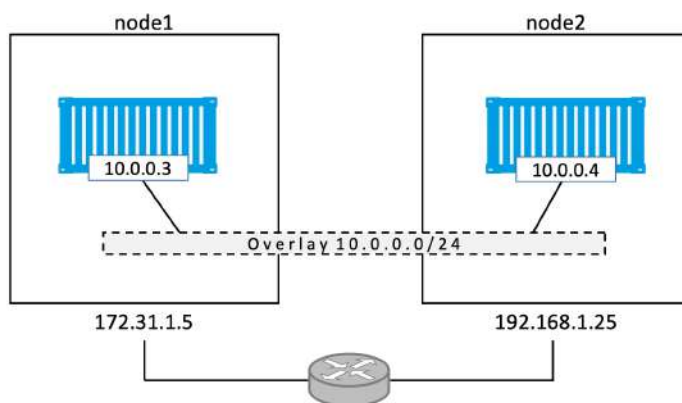


Figure 12.3

As you can see, there's a Layer 2 overlay network spanning both nodes and each container has an IP address on it. This means the container on **node1** will be able to ping the container on **node2** using its 10.0.0.4 address. This works despite the fact that both *nodes* are on different layer 2 underlay networks.

Let's prove it.

Log on to the container on **node1** and ping the remote container. You'll need to install the ping utility in the container to complete this task. Remember that the container IDs will be different in your environment.

```
$ docker exec -it 396c8b142a85 bash

# apt-get update && apt-get install iputils-ping -y
<Snip>
Reading package lists... Done
Building dependency tree
Reading state information... Done
<Snip>
Setting up iputils-ping (3:20190709-3) ...
Processing triggers for libc-bin (2.31-0ubuntu9) ...

# ping 10.0.0.4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=1.06 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=1.07 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=1.03 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=1.26 ms
^C
```

Congratulations. The container on **node1** can ping the container on **node2** via the overlay network. If you created the network with the `-o encrypted` flag, the exchange will have been encrypted.

You can also trace the route of the ping command from within the container. This will report a single hop, proving that the containers are communicating directly via the overlay network — blissfully unaware of any underlay networks that are being traversed.

You'll need to install `traceroute` in the container for this to work.

```
# apt install inetutils-traceroute
<Snip>

# traceroute 10.0.0.4
traceroute to 10.0.0.4 (10.0.0.4), 30 hops max, 60 byte packets
 1  test-svc.2.97v...a5.uber-net (10.0.0.4)  1.110ms  1.034ms  1.073ms
```

So far, we've created an overlay network with a single command. Then we added containers to it. The containers were scheduled on two hosts on two different layer 2 underlay networks. We located the container's IP addresses and proved they could communicate directly via the overlay network.

Now that we've seen how easy it is to build and use a secure overlay network, let's find out how it's all put together behind the scenes.

Overlay networks explained

First and foremost, Docker overlay networking uses *VXLAN tunnels* to create virtual layer 2 overlay networks. So, before we go any further, let's do a quick VXLAN primer.

VXLAN primer

At the highest level, VXLANs let you create layer 2 networks on top of an existing layer 3 infrastructure. That's a lot of jargon that means you can create simple networks that hide horribly complex network topologies. The example we used earlier created a new 10.0.0.0/24 layer 2 network on top of a layer 3 IP network comprising two other layer 2 networks connected by a router. See Figure 12.4.

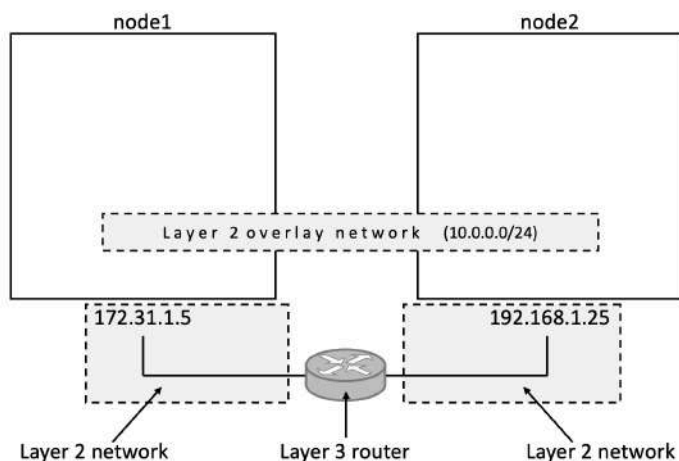


Figure 12.4

The beauty of VXLAN is that it's an encapsulation technology. This means existing routers and network infrastructure just see it as regular IP/UDP packets and handle without requiring any changes.

To create the overlay, a *VXLAN tunnel* is created through the underlay networks. The tunnel is what allows traffic to flow freely without having to interact with the complexity of the underlay networks. We use the terms *underlay networks* or *underlay infrastructure* to refer to the networks the overlay has to tunnel through.

Each end of the VXLAN tunnel is terminated by a VXLAN Tunnel Endpoint (VTEP). It's this VTEP that encapsulates and de-encapsulates the traffic entering and exiting the tunnel. See Figure 12.5.

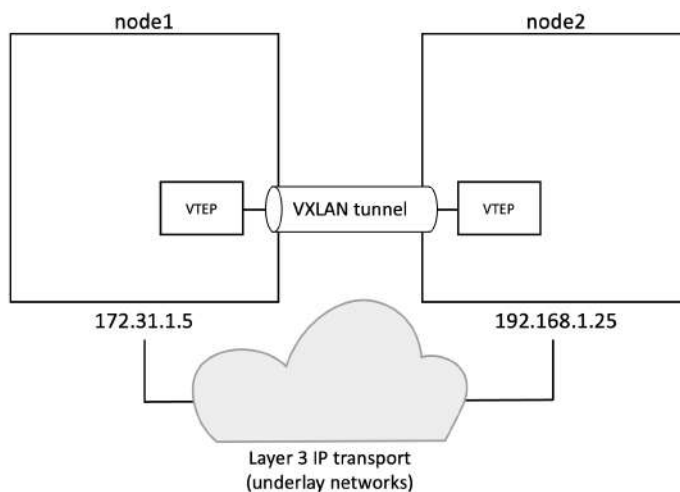


Figure 12.5

This image shows the layer 3 infrastructure as a cloud for two reasons:

- It can be a lot more complex than two networks and a single router as shown in previous diagrams
- The VXLAN tunnel abstracts the complexity and makes it opaque

Walk through our two-container example

The hands-on examples from earlier had two hosts connected via an IP network. Each host ran a single container and you created a single overlay network for the containers. However, lots of things happened behind the scenes to make this happen...

A new *sandbox* (network namespace) was created on each host.

A virtual switch called **Br0** was created inside the sandboxes. A VTEP is also created with one end plumbed into the **Br0** virtual switch and the other end plumbed into the host network stack. The end in the host network stack gets an IP address on the underlay network the host is connected to and is bound to a UDP socket on port 4789. The two VTEPs on each host create the overlay via a VXLAN tunnel as seen in Figure 12.6.

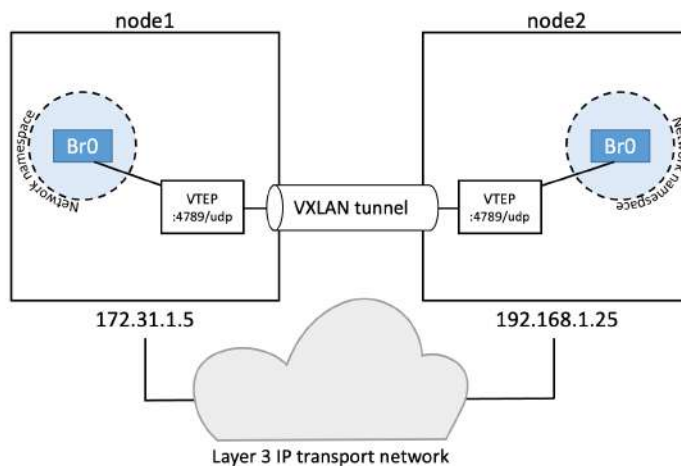


Figure 12.6

At this point, the VXLAN overlay is created and ready for use.

Each container then gets its own virtual Ethernet (veth) adapter that is also plumbed into the local **Br0** virtual switch. The final topology looks like Figure 12.7, and although it's complex, it should be getting easier to see how the two containers can communicate over the VXLAN overlay despite their hosts being on two separate networks.

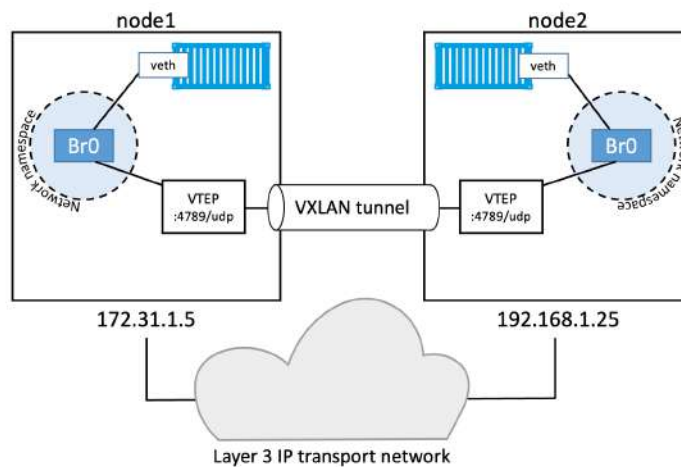


Figure 12.7

Communication example

Now that we've seen the main plumbing elements, let's see how the two containers communicate.

Warning! This section gets very technical. However, you don't need to understand it all for day-to-day operations.

For this example, we'll call the container on node1 "**C1**" and the container on node2 "**C2**". And let's assume **C1** wants to ping **C2** like we did in the practical example earlier. Figure 12.8 adds the containers and their IPs.

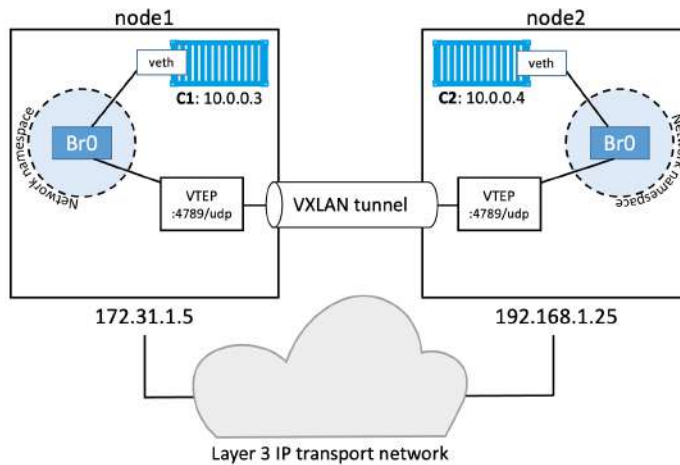


Figure 12.8

C1 creates the ping requests and sets the destination IP address to be the 10.0.0.4 address of **C2**.

C1 doesn't have an entry for **C2** in its local MAC address table (ARP cache) so it floods the packet on all interfaces. The VTEP interface is connected to **Br0** which knows how to forward the frame, so responds with its own MAC address. This is a *proxy ARP* reply and results in the VTEP *learning* how to forward the packet and updating its MAC table so that all future packets for **C2** will be transmitted directly to the local VTEP. The **Br0** switch knew about **C2** because all newly started containers have their network details propagated to other nodes in the swarm using the network's built-in gossip protocol.

The ping is sent to the VTEP interface which performs the encapsulation required to tunnel it through the underlay networks. At a fairly high level, this encapsulation adds a VXLAN header to individual Ethernet frames. This header contains the VXLAN network ID (VNID) which is used to map frames from VLANs to VXLANs and vice

versa. Each VLAN gets mapped to VNID so that packets can be de-encapsulated on the receiving end and forwarded to the correct VLAN. This maintains network isolation.

The encapsulation also wraps the frame in a UDP packet and adds the IP of the remote VTEP on node2 in the *destination IP field*. It also adds the UDP port 4789 socket information. This encapsulation allows the packets to be sent across the underlay networks without the underlays having to know anything about VXLAN.

When the packet arrives at node2, the kernel sees it's addressed to UDP port 4789. The kernel also knows it has a VTEP bound to this socket. As a result, it sends the packet to the VTEP, which reads the VNID, de-encapsulates the packet, and sends it on to its own local **Br0** switch on the VLAN corresponding the VNID. From there it is delivered to container **C2**.

And that... my friends... is how VXLAN technology is leveraged by native Docker overlay networking — a whole load of mind-blowing complexity beautifully simplified with a couple of Docker commands.

Hopefully that's enough to get you started with any production Docker deployments. It should also give you the knowledge required to talk to your networking team about the networking aspects of your Docker infrastructure. On the topic of talking to your networking team... I recommend you don't approach them thinking that you now know everything about VXLAN. If you do, you'll probably embarrass yourself. I'm speaking from experience ;-)

One final thing. Docker also supports Layer 3 routing within an overlay network. For example, you can create an overlay network with two subnets and Docker will take care of routing between them. The command to create a network like this could be `docker network create --subnet=10.1.1.0/24 --subnet=11.1.1.0/24 -d overlay prod-net`. This would result in two virtual switches, **Br0** and **Br1**, inside the *sandbox* and routing happens by automatically.

Docker overlay networking - The commands

- `docker network create` is the command we use to create a new container network. The `-d` flag specifies the driver to use, and the most common driver is `overlay`. You can also install and use drivers from 3rd parties. For overlay networks, the control plane is encrypted by default. You can encrypt the data plane by adding the `-o encrypted` flag but performance overhead might be incurred.
- `docker network ls` lists all of the container networks visible to a Docker host. Docker hosts running in *swarm mode* only see overlay networks if they are running containers attached to those networks. This keeps network-related gossip to a minimum.

- `docker network inspect` shows detailed information about a particular container network. This includes *scope*, *driver*, *IPv4* and *IPv6* info, *subnet configuration*, *IP addresses of connected containers*, *VXLAN network ID*, and *encryption state*.
- `docker network rm` deletes a network

Chapter Summary

In this chapter, we saw how easy it is to create new Docker overlay networks with the `docker network create` command. We then learned how they are put together behind the scenes using VXLAN technology.

13: Volumes and persistent data

Stateful applications that persist data are more and more important in the world of cloud-native and microservices applications. So, we'll turn our attention in this chapter to investigating how Docker handles applications that write persistent data.

We'll split the chapter into the usual three parts:

- The TLDR
- The deep dive
- The commands

Volumes and persistent data - The TLDR

There are two main categories of data — persistent and non-persistent.

Persistent is the data we need to *keep*. Things like customer records, financial data, research results, audit logs, and even some types of application *log* data. Non-persistent is the data we don't need to keep.

Both are important, and Docker has solutions for both.

To deal with non-persistent data, every Docker container gets its own non-persistent storage. This is automatically created for every container and is tightly coupled to the lifecycle of the container. As a result, deleting the container will delete the storage and any data on it.

To deal with persistent data, containers need to store it in a *volume*. Volumes are separate objects that have their lifecycles decoupled from containers. This means you can create and manage volumes independently, and they don't get deleted when their container is deleted.

That's the TLDR. Let's take a closer look.

Volumes and persistent data - The Deep Dive

Some people still think containers aren't good for stateful applications that persist data. This was true a few years ago. However, things have changed and containers are now excellent choices for apps that create persistent data.

We're about to see some of the ways that containers deal with persistent and non-persistent data, and you'll see lots of similarities with virtual machines.

We'll start out with non-persistent data.

Containers and non-persistent data

Containers are designed to be immutable. This is jargon that means read-only — it's a best practice not to change the configuration of a container after it's deployed. If something breaks or you need to change something, you create a brand-new container with the fixes or updates and replace the old container with this new one. You should never log into a running container and make configuration changes.

However, many applications require a read-write filesystem in order to run — they won't even run on a read-only filesystem. This means it's not as simple as making containers entirely read-only. To help with this, containers created by Docker have a thin read-write layer on top of the read-only images they're based on. Figure 13.1 shows two running containers sharing a single read-only image.

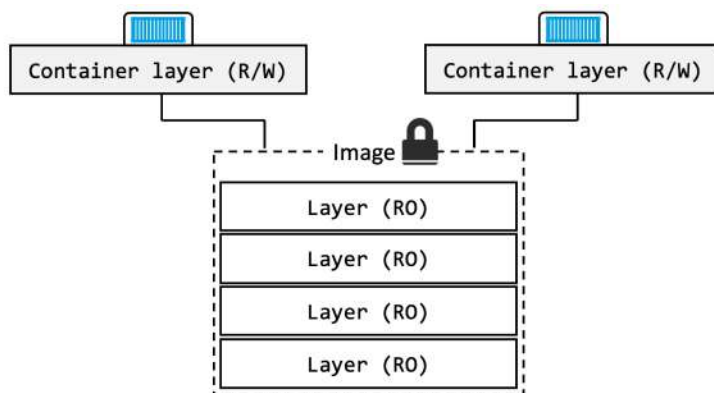


Figure 13.1 Ephemeral container storage

Each writable container layer exists in the filesystem of the Docker host and you'll hear it called various names. These include *local storage*, *ephemeral storage*, and *graphdriver storage*. It's typically located on the Docker host in these locations:

- Linux Docker hosts: `/var/lib/docker/<storage-driver>/...`
- Windows Docker hosts: `C:\ProgramData\Docker\windowsfilter\...`

This thin writable layer is an integral part of many containers and enables all read/write operations. If you, or an application, update files or add new files, they'll be written to this layer. However, it's tightly coupled to the container's lifecycle — it gets created

when the container is created and it gets deleted when the container is deleted. The fact that it's deleted along with a container means it's not an option for important data that you need to keep (persist).

If your containers don't create persistent data, this thin writable layer of *local storage* will be fine and you're good to go. However, if your containers need to persist data, you need to read the next section.

Containers and persistent data

Volumes are the recommended way to persist data in containers. There are three major reasons for this:

- Volumes are independent objects that are not tied to the lifecycle of a container
- Volumes can be mapped to specialized external storage systems
- Volumes enable multiple containers on different Docker hosts to access and share the same data

At a high-level, you create a volume, then you create a container and mount the volume into it. The volume is mounted into a directory in the container's filesystem, and anything written to that directory is stored in the volume. If you delete the container, the volume and its data will still exist.

Figure 13.2 shows a Docker volume existing outside of the container as a separate object. It is mounted into the container's filesystem at `/data`, and any data written to the `/data` directory will be stored on the volume and will exist after the container is deleted.

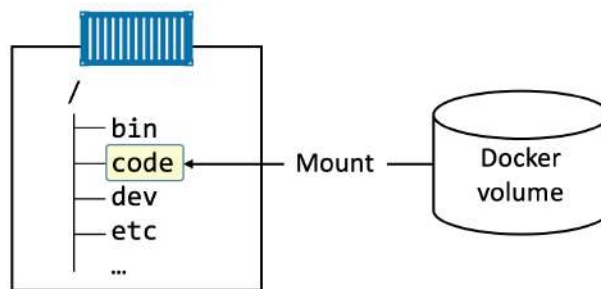


Figure 13.2 High-level view of volumes and containers

In Figure 13.2, the `/data` directory is a Docker volume that can either be mapped to an external storage system or a directory on the Docker host. Either way, its lifecycle is decoupled from the container. All other directories in the container use the thin writable container layer in the local storage area on the Docker host.

Creating and managing Docker volumes

Volumes are first-class objects in Docker. Among other things, this means they are their own object in the API and have their own `docker volume` sub-command.

Use the following command to create a new volume called `myvol`.

```
$ docker volume create myvol  
myvol
```

By default, Docker creates new volumes with the built-in `local` driver. As the name suggests, volumes created with the `local` driver are only available to containers on the same node as the volume. You can use the `-d` flag to specify a different driver.

Third-party volume drivers are available as plugins¹⁴. These provide Docker with advanced features and seamless access external storage systems such as cloud storage services and on-premises storage systems including SAN and NAS. This is shown in Figure 13.3.

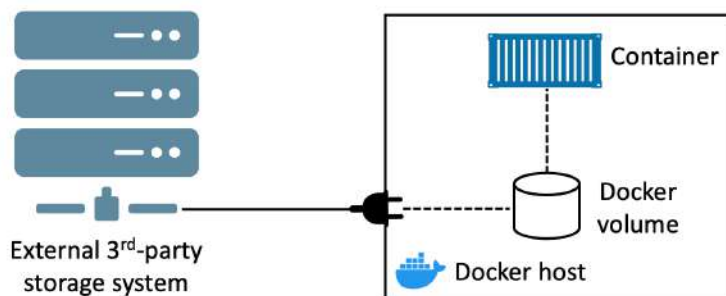


Figure 13.3 Plugging external storage into Docker

We'll look at an example with a third-party driver in a later section.

Now that the volume is created, you can see it with the `docker volume ls` command and inspect it with the `docker volume inspect` command.

¹⁴https://docs.docker.com/engine/extend/legacy_plugins/#volume-plugins

```
$ docker volume ls
DRIVER          VOLUME NAME
local           myvol

$ docker volume inspect myvol
[
  {
    "CreatedAt": "2023-05-23T10:00:18+01:00",
    "Driver": "local",
    "Labels": null,
    "Mountpoint": "/var/lib/docker/volumes/myvol/_data",
    "Name": "myvol",
    "Options": null,
    "Scope": "local"
  }
]
```

Notice that the `Driver` and `Scope` are both `local`. This means the volume was created with the `local` driver and is only available to containers on this Docker host. The `Mountpoint` property tells us where in the Docker host's filesystem the volume exists.

All volumes created with the `local` driver get their own directory under `/var/lib/docker/volumes` on Linux, and `C:\ProgramData\Docker\volumes` on Windows. This means you can see them in your Docker host's filesystem. You can even access them directly from your Docker host, although this is not recommended. We showed an example of this in the chapter on Docker Compose — we copied a file directly into a volume's directory on the Docker host and the file immediately appeared in the volume inside the container.

Now that the volume is created, it can be used by one or more containers. We'll see usage examples in a minute.

There are two ways to delete a Docker volume:

- `docker volume prune`
- `docker volume rm`

`docker volume prune` will delete **all volumes** that are not mounted into a container or service replica, so **use with caution!** `docker volume rm` lets you specify exactly which volumes you want to delete. Neither command will delete a volume that is in use by a container or service replica.

As the `myvol` volume is not in use, delete it with the `prune` command.


```
$ docker volume prune
```

```
WARNING! This will remove all volumes not used by at least one container.  
Are you sure you want to continue? [y/N] y
```

```
Deleted Volumes:  
myvol  
Total reclaimed space: 0B
```

Congratulations, you’ve created, inspected, and deleted a Docker volume. And you did it all without interacting with a container. This demonstrates the independent nature of volumes.

At this point, you know all the commands to create, list, inspect, and delete Docker volumes. However, it’s also possible to deploy volumes via Dockerfiles using the `VOLUME` instruction. The format is `VOLUME <container-mount-point>`. Interestingly, you cannot specify a directory on the host when defining a volume in a Dockerfile. This is because *host* directories are different depending on what OS your Docker host is running – it could break your builds if you specified a directory on a Docker host that doesn’t exist. As a result, defining a volume in a Dockerfile requires you to specify host directories at deploy-time.

Demonstrating volumes with containers and services

Let’s see how to use volumes with containers and services.

Use the following command to create a new standalone container that mounts a volume called `bizvol`.

```
$ docker run -it --name voltainer \  
  --mount source=bizvol,target=/vol \  
  alpine
```

The command uses the `--mount` flag to mount a volume called “bizvol” into the container at either `/vol`. The command completes successfully despite the fact there is no volume on the system called `bizvol`. This raises an interesting point:

- If you specify an existing volume, Docker will use the existing volume
- If you specify a volume that doesn’t exist, Docker will create it for you

In this case, `bizvol` didn’t exist, so Docker created it and mounted it into the new container. This means you’ll be able to see it with `docker volume ls`.

```
$ docker volume ls
DRIVER          VOLUME NAME
local           bizvol
```

Although containers and volumes have separate lifecycle's, you cannot delete a volume that is in use by a container. Try it.

```
$ docker volume rm bizvol
Error response from daemon: remove bizvol: volume is in use - [b44d3f82...dd2029ca]
```

The volume is brand new, so it doesn't have any data. Let's exec onto the container and write some data to it.

```
$ docker exec -it voltainer sh

/# echo "I promise to leave a review of the book on Amazon" > /vol/file1

/# ls -l /vol
total 4
-rw-r--r-- 1 root  root   50 May 23 08:49 file1

/# cat /vol/file1
I promise to leave a review of the book on Amazon
```

Type `exit` to return to the shell of your Docker host, and then delete the container with the following command.

```
$ docker rm voltainer -f
voltainer
```

Even though the container is deleted, the volume still exists:

```
$ docker ps -a
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS

$ docker volume ls
DRIVER          VOLUME NAME
local           bizvol
```

As the volume still exists, you can look at its mount point on the host to check if the data is still there.

Run the following commands from the terminal of your Docker host. The first one will show that the file still exists, the second will show the contents of the file. You may have to prefix the commands with `sudo`.

Be sure to use the `C:\ProgramData\Docker\volumes\bizvol_data` directory if you're following along on Windows. Also, this step won't work on Docker Desktop because Docker Desktop runs your entire Docker environment inside a VM.

```
$ ls -l /var/lib/docker/volumes/bizvol/_data/
total 4
-rw-r--r-- 1 root root 50 Jan 12 14:25 file1

$ cat /var/lib/docker/volumes/bizvol/_data/file1
I promise to leave a review of the book on Amazon
```

Great, the volume and the data still exist.

It's even possible to mount the `bizvol` volume into a new service or container. The following command creates a new container that mounts `bizvol` at `/vol`.

```
$ docker run -it \
  --name hellcat \
  --mount source=bizvol,target=/vol \
  alpine sh
```

Your terminal is now attached to the `hellcat` container.

```
# cat /vol/file1
I promise to write a review of the book on Amazon
```

Excellent, the volume has preserved the original data and made it available to a new container.

Type `exit` to leave the container and jump over to Amazon to leave the book review :-D

Sharing storage across cluster nodes

Integrating external storage systems with Docker makes it possible to share volumes between cluster nodes. These external systems can be cloud storage services or enterprise storage systems in your on-premises data centers. As an example, a single storage LUN or NFS share can be presented to multiple Docker hosts, allowing it to be used by containers and service replicas no-matter which Docker host they're running on. Figure 13.4 shows a single external shared volume being presented to two Docker nodes. These Docker nodes can then make the shared volume available to either, or both containers.

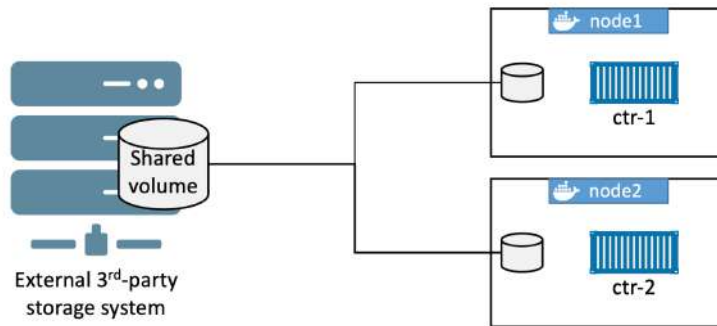


Figure 13.4

Building a setup like this requires a lot of things. You need access to a specialised storage systems and knowledge of how it works and presents storage. You also need to know how your applications read and write data to the shared storage. Finally, you need a volume driver plugin that works with the external storage system.

Volume drivers are available as plugins that run as containers, and the best place to find them is Docker Hub. Just open a browser page to hub.docker.com and filter the view on Plugins. Once you've located the appropriate plugin for your storage system, you install it with `docker plugin install`.

Figure 13.5 shows the NetApp Trident plugin on Docker Hub. Notice the `docker plugin install` command.



Figure 13.5

Potential data corruption

A major concern with any configuration that shares a single volume among multiple containers is **data corruption**.

Assume the following example based on Figure 13.4.

The application running in ctr-1 on node1 updates some data in the shared volume. However, instead of writing the update directly to the volume, it keeps it in a local buffer for faster recall (this is common in many operating systems). At this point, the application in ctr-1 *thinks* the data has been written to the volume. However, before ctr-1 on node1 flushes its buffers and commits the data to the volume, the app in ctr-2 on node2 updates the same data with a different value and commits it directly to the volume. At this point, both applications *think* they've updated the data in the volume, but in reality, only the application in ctr-2 has. A few seconds later, ctr-1 on node1 flushes the data to the volume, overwriting the changes made by the application in ctr-2. However, the application in ctr-2 is totally unaware of this! This is one of the ways data corruption happens.

To prevent this, you need to write your applications in a way to avoid things like this.

Volumes and persistent data - The Commands

- `docker volume create` is the command to create new volumes. By default, volumes are created with the `local` driver but you can use the `-d` flag to specify a different driver.
- `docker volume ls` will list all volumes on the local Docker host.
- `docker volume inspect` shows detailed volume information. Use this command to see many interesting volume properties, including where a volume exists in the Docker host's filesystem.
- `docker volume prune` will delete **all** volumes that are not in use by a container or service replica. **Use with caution!**
- `docker volume rm` deletes specific volumes that are not in use.
- `docker plugin install` installs new volume plugins from Docker Hub.
- `docker plugin ls` lists all plugins installed on a Docker host.

Chapter Summary

There are two main types of data: persistent and non-persistent.

Persistent data is data that you need to keep, non-persistent is data that you don't need to keep. By default, all containers get a layer of writable non-persistent storage that lives and dies with the container — we call this *local storage* and it's ideal for non-persistent data. However, if your containers create data that you need to keep, you should store the data in a Docker volume.

Docker volumes are first-class objects in the Docker API and managed independently of containers with their own `docker volume` sub-command. This means that deleting a container will not delete the volumes it was using.

Third party volume plugins can provide Docker access to specialised external storage systems. They're installed from Docker Hub with the `docker plugin install` command and are referenced at volume creation time with the `-d` command flag.

Volumes are the recommended way to work with persistent data in a Docker environment.

14: Deploying apps with Docker Stacks

Deploying and managing cloud-native microservices applications at scale is hard.

Fortunately, Docker Stacks are here to help.

We'll split this chapter into the usual three parts:

- The TLDR
- The deep dive
- The commands

Deploying apps with Docker Stacks - The TLDR

Testing and deploying simple apps on your laptop is easy, but that's for amateurs.

Deploying and managing multi-service apps in real-world production environments... that's for pros.

This is where Docker Stacks come into play. They let you define complex multi-service apps in a single declarative file. They also provide a simple way to deploy and manage entire application lifecycles — initial deployment > health checks > scaling > updates > rollbacks and more.

The process is simple. Define what you want in a *Compose file* and deploy and manage it with the `docker stack` command. That's it!

The Compose file includes the entire stack of microservices that make up the app. It also includes infrastructure such as volumes, networks, secrets, and more. The `docker stack deploy` command is used to deploy and manage the entire app from that single file. Simple.

To accomplish all of this, stacks build on top of Docker Swarm, meaning you get all of the security and advanced features that come with Swarm.

In a nutshell, Docker is great for application development and testing. Docker Stacks are great for scale and production.

Deploying apps with Docker Stacks - The Deep Dive

If you know Docker Compose, you'll find Docker Stacks really easy.

From an architecture perspective, stacks are at the top of the Docker application hierarchy. They build on top of *services*, which in turn build on top of containers.

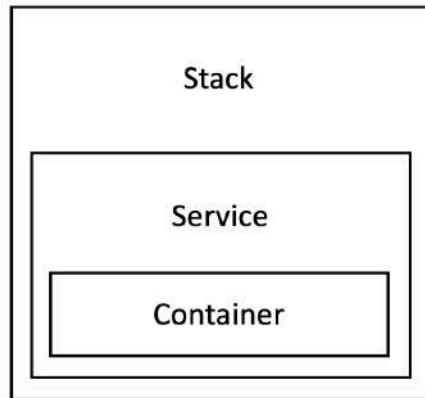


Figure 14.1

We'll divide this section of the chapter as follows:

- Overview of the sample app
- Stack files
- Deploying stacks
- Managing stacks

Overview of the sample app

For the rest of the chapter, we'll be using an application with two services, an encrypted overlay network, a volume, and a port mapping. The application architecture is shown in Figure 14.2.

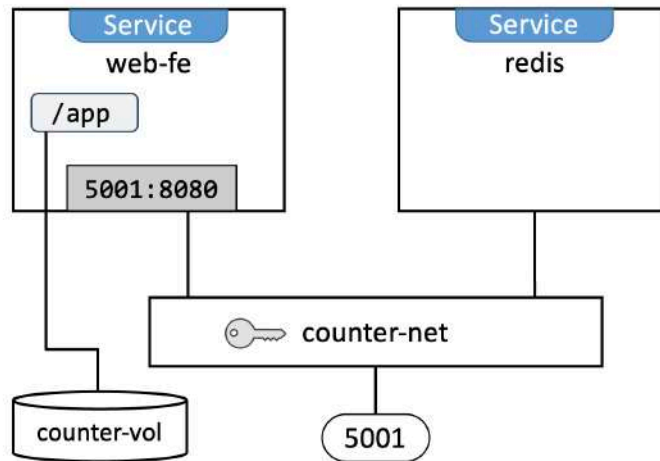


Figure 14.2

Terminology: When referring to *services* we're talking about the Docker service object that is one or more identical containers managed as a single object on a swarm cluster.

If you haven't already done so, clone the book's GitHub repo so that you have all of the application source files on your local machine.

```
$ git clone https://github.com/nigelpoulton/ddd-book.git
Cloning into 'ddd-book'...
remote: Enumerating objects: 8904, done.
remote: Counting objects: 100% (74/74), done.
remote: Compressing objects: 100% (52/52), done.
remote: Total 8904 (delta 21), reused 70 (delta 18), pack-reused 8830
Receiving objects: 100% (8904/8904), 74.00 MiB | 4.18 MiB/s, done.
Resolving deltas: 100% (1378/1378), done.
```

Feel free to look at the application. However, we'll be focussing on the `compose.yaml` file. Sometimes we'll refer to the Compose file as the *stack file*, and sometimes we'll refer to the application as the *stack*.

At the highest level, the Compose defines 3 top-level keys.

```
networks:
volumes:
services:
```

Networks defines the networks required by the app, **volumes** defines volumes, and **services** is where you define the microservices that make up the app. The file is a simple example of *infrastructure as code* — the application and its infrastructure is all defined in a configuration that's used to deploy and manage it.

If you expand each top-level key, you'll see how things map to Figure 14.2 with one network, one volume, and two services.

```
networks:
  counter-net:
volumes:
  counter-vol:
services:
  web-fe:
  redis:
```

The stack file is also a great source of documentation as it captures and defines most of the app.

Let's take a closer look at each section of the stack file.

Looking closer at the stack file

Stack files are almost identical to Compose files. The differences come at runtime — Swarm and Stacks might support a different set of features than Compose. For example, Stacks don't support building images from Dockerfiles but Compose does.

One of the first things Docker does when deploying an app from a stack file is create the required networks listed under the `networks` key. If the networks don't already exist, Docker creates them.

Let's look at the networks and networking defined in our stack file.

Networks and networking

The sample app defines a single network called `counter-net`. We're forcing it to be an overlay network and we're encrypting the data plane.

```
networks:
  counter-net:
    driver: overlay
    driver_opts:
      encrypted: 'yes'
```

It needs to be an overlay network so it can span all nodes in the swarm.

Encrypting the data ensures traffic is private. However, this incurs a performance penalty that varies based on factors such as traffic type and traffic flow. It's not uncommon for the performance penalty to be around 10%, but you should perform extensive testing against your particular applications.

The stack also defines a port mapping for the web-fe service:

```
services:
  web-fe:
    <Snip>
    ports:
      - target: 8080
        published: 5001
```

This publishes port 5001 on the swarm-wide ingress network and redirects traffic to port 8080 in any of the service replicas. This results in all traffic hitting any swarm node on port 5001 being routed to port 8080 on the service replicas.

Let's look at the volumes and mounts.

Volumes and mounts

The app defines a single volume called `counter-vol` and mounts it into the `/app/` directory on all redis replicas. Any read or write operations to the `/app` folder will be read and written to the volume.

```
volumes:
  counter-vol:

services:
  redis:
    <Snip>
    volumes:
      - type: volume
        source: counter-vol
        target: /app
```

Let's look at the services.

Services

Services are where most of the action happens.

Our application defines two and we'll look at each in turn.

The web-fe service

As you can see, the web-fe service defines an image, an app, a replica count, an update configuration, a restart policy, a network, a published port, and a volume.

```
web-fe:
  image: nigelpoulton/ddd-book:swarm-app
  command: python app.py
  deploy:
    replicas: 10
    update_config:
      parallelism: 2
      delay: 10s
      failure_action: rollback
    placement:
      constraints:
        - 'node.role == worker'
    restart_policy:
      condition: on-failure
      delay: 5s
      max_attempts: 3
      window: 120s
  networks:
    - counter-net
  ports:
    - published: 5001
      target: 8080
  volumes:
    - type: volume
      source: counter-vol
      target: /app
```

The **image** key is the only mandatory key in the service object and it defines the image used to build the service replicas. Remember, a service is one or more identical containers.

Docker is *opinionated* and assumes you want to pull images from Docker Hub. However, you can use 3rd-party registries by adding the registry's DNS name before the image name. For example, adding `gcr.io` before an image name will pull it from Google's container registry.

One difference between Docker Stacks and Docker Compose is that stacks don't support **builds**. This means all images have to be built before we deploy the stack.

The **command** key defines the app to run in each replica. Our example is telling Docker to run `python app.py` as the main process in every service replica.

```
web-fe:
  <Snip>
  command: python app.py
```

The **deploy.replicas** key is telling swarm to deploy and manage 4 service replicas. All replicas are identical other than names and IPs.

If you need to change the number of replicas after you've deployed the service, you should do so declaratively. This means updating `deploy.replicas` field in the stack file with the new value and then redeploying the stack. We'll see this later, but re-deploying a stack does not affect services that you haven't made a change to.

```
web-fe:
  deploy:
    replicas: 4
```

The **deploy.update_config** block says to perform updates by updating two replicas at a time, wait 10 seconds in between each set, and perform a rollback if the update encounters a problem. Rolling back will start new replicas based on the previous definition of the service. The default value for `failure_action` is `pause`, which will stop further replicas being updated. The other option is `continue`.

```
web-fe:
  deploy:
    update_config:
      parallelism: 2
      delay: 10s
      failure_action: rollback
```

The **deploy.placement** block forces all replicas onto worker nodes.

```
web-fe:
  deploy:
    placement:
      constraints:
        - 'node.role == worker'
```

The **deploy.restart_policy** block says to restart replicas if they fail. It also says to try a maximum of 3 times, wait 5 seconds in-between each restart attempt, and wait up to 120 seconds to decide if the restart worked.

```
web-fe:
  deploy:
    restart_policy:
      condition: on-failure
      max_attempts: 3
      delay: 5s
      window: 120s
```

The **networks** key tells swarm to attach all replicas to the counter-net network.

```
web-fe:
  networks:
    - counter-net
```

The **ports** block publishes the app on the ingress network on port 5001 and the counter-net network on 8080. This ensures traffic hitting the swarm on 5001 gets redirected to the service replicas on 8080.

```
web-fe:
  ports:
    - published: 5001
      target: 8080
```

Finally, the **volumes** block mounts the counter-vol volume into /app in each service replica.

```
web-fe:
  volumes:
    - type: volume
      source: counter-vol
      target: /app
```

The redis service

The redis service is much simpler. It pulls the `redis:alpine` image, starts a single replica, and attaches it to the counter-net network. This is the same network as the web-fe service, meaning the two services will be able to communicate with each other by name (“redis” and “web-fe”).

```
redis:
  image: "redis:alpine"
  networks:
    counter-net:
```

As mentioned previously, Compose files are a great source of application documentation. We know this application has 2 services, 2 networks, and 1 volume. We know how the services communicate, how they're exposed outside of the swarm, and we know a bit about how they'll be deployed, updated, and how they'll restart from failures.

Let's deploy the app.

Deploying the app

We'll deploy the app as a Docker Stack. This means our Docker nodes need to be configured as a swarm.

Building a lab for the sample app

In this section we'll build a three-node swarm. You can follow along on Play with Docker, Multipass VMs or just about any Docker environment. You can even follow along on Docker Desktop. However, Docker Desktop is limited to a single node running as a manager, meaning you'll have to delete the node role constraint:

```
web-fe:
  deploy:
    placement:
      constraints:
        - 'node.role == worker'
        <<---- Delete if using Docker Desktop
        <<---- Delete if using Docker Desktop
        <<---- Delete if using Docker Desktop
```

1. Initialize a new Swarm.

Run the following command on the node that you want to be the swarm manager.

```
$ docker swarm init
Swarm initialized: current node (lhma...w4nn) is now a manager.
<Snip>
```

2. Add worker nodes.

Copy the `docker swarm join` command that was output by the previous command. Paste it into the two nodes you want to join as workers.


```
//Worker 1 (wrk1)
wrk-1$ docker swarm join --token SWMTKN-1-2hl6...-...3lqg 172.31.40.192:2377
This node joined a swarm as a worker.

//Worker 2 (wrk2)
wrk-2$ docker swarm join --token SWMTKN-1-2hl6...-...3lqg 172.31.40.192:2377
This node joined a swarm as a worker.
```

3. Verify that the Swarm is configured with one manager and two workers.

Run this command from the manager node.

```
$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
lhm...4nn *	mgr1	Ready	Active	Leader
b74...gz3	wrk1	Ready	Active	
o9x...um8	wrk2	Ready	Active	

The Swarm is now ready. Let's deploy the stack.

Deploying the sample app

Stacks are deployed using the `docker stack deploy` command. In its basic form it accepts two arguments:

- name of the stack file
- name of the stack

We'll use the `compose.yaml` file in the `swarm-app` folder of the book's GitHub repo and we'll call the app `ddd`. Feel free to give yours a different name.

Run the following commands from `swarm-app` directory on the Swarm manager. If the manager doesn't have a copy of the GitHub repo, clone it with this command.

```
$ git clone https://github.com/nigelpoulton/ddd-book.git
```

Deploy the stack.

```
$ docker stack deploy -c compose.yaml ddd
Creating network ddd_counter-net
Creating service ddd_web-fe
Creating service ddd_redis
```

You can run `docker network ls`, `docker volume ls`, and `docker service ls` commands to see the networks, volumes, and services that were deployed as part of the app.

A few things to note from the output of the command.

The networks and volumes were created before the services. This is because services use these and will fail to start if they don't exist.

Docker prefixes the name of the stack to every resource it creates. In our example, the stack is called `ddd`, meaning all resources are named `ddd_<resource>`. For example, the `counter-net` network is called `ddd_counter-net`.

You can verify the status of a stack with a couple of commands. `docker stack ls` lists very basic info on all stacks on the system. `docker stack ps <stack-name>` gives more detailed information about a specific stack. Let's see them both.

```
$ docker stack ls
NAME      SERVICES
ddd       2

$ docker stack ps ddd
NAME                IMAGE                        NODE    DESIRED STATE  CURRENT STATE
ddd_redis.1         redis:alpine                mgr1    Running        Running 4 mins
ddd_web-fe.1        nigelpoulton/ddd...        wrk1    Running        Running 4 mins
ddd_web-fe.2        nigelpoulton/ddd...        wrk2    Running        Running 4 mins
ddd_web-fe.3        nigelpoulton/ddd...        wrk2    Running        Running 4 mins
<Snip>
ddd_web-fe.10       nigelpoulton/ddd...        wrk1    Running        Running 4 mins
```

The `docker stack ps` command is a good place to start when troubleshooting services that fail to start. It gives an overview of every service in the stack, including which node replicas are scheduled on, current state, desired state, and error messages. The following output shows two failed attempts to start a replica for the `web-fe` service on the `wrk2` node.

```
$ docker stack ps ddd
NAME      NODE    DESIRED  CURRENT  ERROR
web-fe.1  wrk-2   Shutdown Failed    "task: non-zero exit (1)"
\_web-fe.1 wrk-2   Shutdown Failed    "task: non-zero exit (1)"
```

Use the `docker service logs` command for more detailed logs. You pass it the service name or ID, or a replica ID. If you pass it the service name or ID, you'll get the logs for all service replicas. If you pass it a particular replica ID, you'll only get the logs for that replica.

The following example shows the logs for all replicas in the `ddd_web-fe` service.

```
$ docker service logs ddd_web-fe
ddd_web-fe.9.i23puo71kq12@node2 | * Serving Flask app 'app'
ddd_web-fe.5.z4otpnjrv58@node2 | * Debug mode: on
<Snip>
ddd_web-fe.6.novrix5iuxy@node2 | * Debug mode: on
ddd_web-fe.6.novrix5iuxy@node2 | * Debugger is active!
ddd_web-fe.6.novrix5iuxy@node2 | * Debugger PIN: 127-233-151
```

You can follow the logs (`--follow`), tail them (`--tail`), and you may be able to get extra details (`--details`).

Point a browser at the app to verify it's up and working. As it's exposed on the swarm ingress on port 5001 you can point a browser to any cluster node on that port. If you're on Docker Desktop you can use `localhost:5001`.

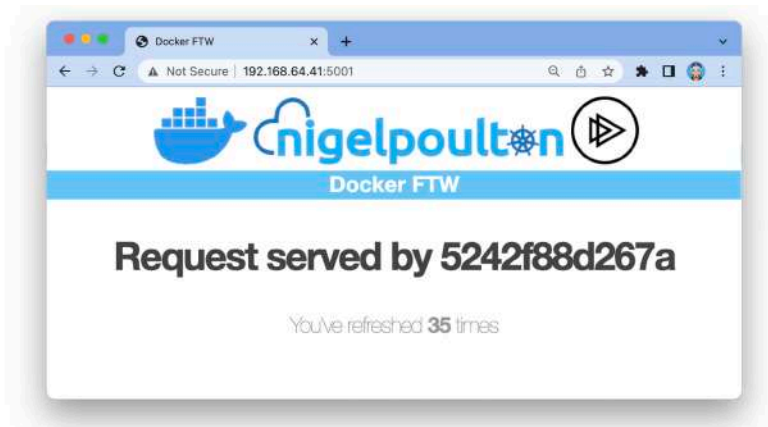


Figure 14.3

Now that the stack is up and running, let's see how declaratively manage it.

Managing a stack

We know a *stack* is a set of related services and infrastructure that gets deployed and managed as a unit. And while that's a fancy sentence with a few buzzwords, it reminds us that stacks are built from normal Docker resources — networks, volumes, secrets, services, containers etc. This means we can inspect them individual components with their normal docker commands such as `docker network`, `docker volume`, `docker service` etc.

With this in mind, it's possible to use the `docker service` command to manage services that are part of the stack. A simple example would be using the `docker service scale` command to increase the number of replicas in the `web-fe` service. However, using the

command line like this is called the *imperative* method and it's **not the recommended method!**

The recommended method is the *declarative method*. This uses the stack file as the ultimate source of truth and demands that all changes be made by updating the stack file and redeploying the app from the updated file.

Here's a quick example of why the imperative method (making changes via the CLI and individual docker commands) is bad:

Imagine we have a stack deployed from the `compose.yaml` file that we cloned from GitHub earlier in the chapter. This means we have four replicas of the `web-fe` service. If we use the `docker service scale` command scale up to 10 in order to meet increased demand, the current state of the app will no longer match the Compose file. If that doesn't sound like a big problem, imagine we then edit the stack file to use a newer image and rollout the change the recommended way with the Compose file and the `docker stack deploy` command. As part of this rollout, the number of `web-fe` replicas in the cluster will be rolled back to just four because we didn't update to the stack file to match the environment. For this kind of reason, it's recommended to make all changes via the stack file, and to manage the stack file in a proper version control system.

Let's walk through the process of making a couple of declarative changes to the stack.

We'll make the following changes:

- Increase the number of `web-fe` replicas from 4 to 10
- Update the app based on a newer image called `:swarm-appv2`

Figure 14.4 shows the old view and the new view.

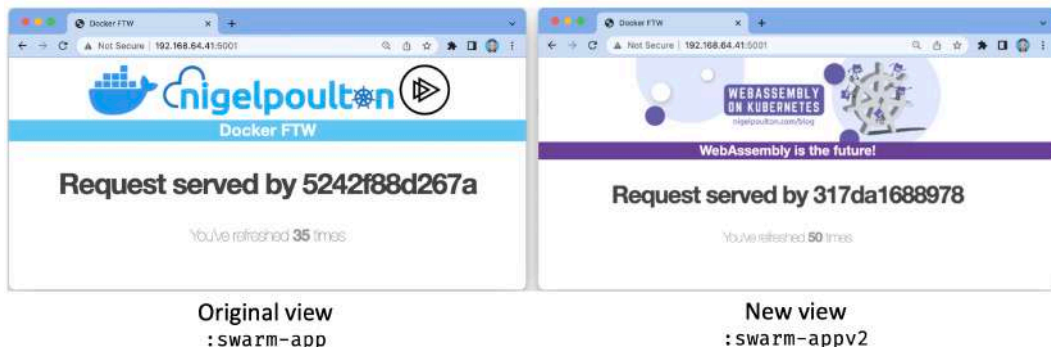


Figure 14.4

Update the `compose.yaml` file to reflect the changes. The relevant sections should look like this:

```
<Snip>
services:
  web-fe:
    image: nigelpoulton/ddd-book:swarm-appv2    <<---- changed to swarm-appv2
    command: python app.py
    deploy:
      replicas: 4      <<---- Changed from 4 to 10
<Snip>
```

Save the file and redeploy the app.

```
$ docker stack deploy -c compose.yaml ddd
Updating service ddd_redis (id: ozljsazuv7mmh14ep70pv43cf)
Updating service ddd_web-fe (id: zbbplw0hul2gbr593mwslz5i)
```

Re-deploying the app like this will only update the changed components.

Run a `docker stack ps` to see the progress of the update.

```
$ docker stack ps ddd
```

NAME	IMAGE	NODE	DESIRED	CURRENT STATE
ddd_redis.1	redis:alpine	mgr1	Running	Running 8 minutes ago
ddd_web-fe.1	nigel...app	node2	Running	Running 8 minutes ago
ddd_web-fe.2	nigel...appv2	node2	Running	Running 13 seconds ago
_ddd_web-fe.2	nigel...app	node2	Shutdown	Shutdown 26 seconds ago
ddd_web-fe.3	nigel...app	node2	Running	Running 8 minutes ago

<Snip>

The output has been trimmed so that it fits on the page, and only some of the replicas are shown.

Two things happened when we re-deployed the stack:

- The web-fe service was scaled up from 4 replicas to 10
- The web-fe service was changed to use the `swarm-appv2` image

Scaling up from 4 - 10 added 6 new replicas. These will be deployed with the new image version. The existing 4 replicas will also be deleted and replaced with new ones running the new version. This is because Docker treats replicas as immutable objects and never makes changes to live replicas – it always deletes existing replicas and replaces them with new ones.

Also, the process of updating the 4 existing replicas follows the update rules defined in the Compose file — update two replicas, wait 10 seconds, update the other 2, wait 10 seconds... If any issues occur, the swarm will attempt a rollback to the previous configuration.

```

web-fe:
  deploy:
    update_config:
      parallelism: 2
      delay: 10s
      failure_action: rollback

```

The cluster will eventually converge and *current observed state* will match the new *desired state* of 10 replicas all on the new image. At that point, what is deployed and observed on the cluster will exactly match what is defined in the stack file. This is a happy place to be :-D

Check the update worked by refreshing your browser.

The update doesn't appear to have worked as the original view is still showing! Let's check...

The `docker stack ps` command is a good place to start troubleshooting. The following command shows that we've gone down to four `web-fe` replicas and they're all using the correct `swarm-appv2` image. So what could be wrong?

```

$ docker stack ps ddd
NAME                IMAGE                NODE    DESIRED   CURRENT STATE
ddd_redis.1         redis:alpine         mgr1    Running   Running 18 mins
ddd_web-fe.1        nigel...swarm-appv2 node2    Running   Running 10 mins
ddd_web-fe.2        nigel...swarm-appv2 node2    Running   Running 10 mins
ddd_web-fe.5        nigel...swarm-appv2 node2    Running   Running 10 mins
ddd_web-fe.4        nigel...swarm-appv2 node2    Running   Running 10 mins

```

The issue is with the volume.

When the replicas were updated to run the new image, old replicas were deleted and new ones started. However, the volume and data from the old replicas still exists and gets mounted into the new replicas. This is overwriting the new version of the app with the old version that's still in the volume. Let's walk through the process.

The new image has the updated app with the new web view. Old replicas were deleted and new ones deployed with the new version of the app. However, at runtime the existing volume (with the old version of the app) was mounted into the new replicas and overwrote the new web view. This is a *feature* of volumes and something you should be aware of.

Let's assume you realise the web view is static content and doesn't need a volume, so you decide to remove the volume from the app. The declarative way to do this is to edit the Compose file again, remove the volume and volume mount, and re-deploy the app. Let's do it.

Edit the `compose.yaml` file and make the following changes.

```

volumes:          <<---- Delete this line
  counter-vol:    <<---- Delete this line
<Snip>
services:
  web-fe:
    image: nigelpoulton/ddd-book:swarm-appv2
<Snip>
  volumes:        <<---- Delete this line
    - type: volume <<---- Delete this line
      source: counter-vol <<---- Delete this line
      target: /app <<---- Delete this line

```

Save you changes and re-deploy.

```

$ docker stack deploy -c compose.yaml ddd
Updating service ddd_redis (id: ozljsazuv7mmh14ep70pv43cf)
Updating service ddd_web-fe (id: zbbplw0hul2gbr593mwwslz5i)

```

The stack will update two replicas at a time and wait 10 seconds between each. Once the stack has converged and all replicas are updated you should see the new version of the app in your browser. Hit refresh a few times to make sure it works.

The volume will still exist and will need deleting manually.

This *declarative update pattern* should be used for all updates. I.e., **all changes should be made declaratively via the stack file and rolled out using the `docker stack deploy` command.**

The correct way to delete a stack is with the `docker stack rm` command. Be warned though! It deletes the stack without asking for confirmation.

```

$ docker stack rm ddd
Removing service ddd_redis
Removing service ddd_web-fe
Removing network ddd_counter-net

```

Notice that the network and services were deleted but the volume wasn't. This is because volumes are long-term persistent data stores and exist independent of the lifecycle of containers, services, and stacks.

Congratulations. You know how to deploy and manage a multi-service app using Docker Stacks.

Deploying apps with Docker Stacks - The Commands

- `docker stack deploy` is the command for deploying **and** updating stacks of services defined in a stack file (usually called `compose.yaml`).

- `docker stack ls` lists all stacks on the Swarm, including how many services they have.
- `docker stack ps` gives detailed information about a deployed stack. It accepts the name of the stack as its main argument, lists which node each replica is running on, and shows *desired state* and *current state*.
- `docker stack rm` deletes a stack from the Swarm. It does not ask for confirmation before deleting the stack.

Chapter Summary

Stacks are the native Docker solution for deploying and managing cloud-native microservices applications. They require swarm mode and offer a simple declarative interface for managing the entire lifecycle of applications and infrastructure.

You start with application code and a set of infrastructure requirements — things like networks, ports, volumes, and secrets. You containerize the application and group together all of the app services and infrastructure requirements into a single declarative stack file. You set the number of replicas, as well as rollout and restart policies. You then deploy the application from the stack file using the `docker stack deploy` command.

Future updates to the app should be done declaratively by checking the stack file out of source control, updating it, re-deploying the app from it, and checking it back into source control.

Because the stack file defines things like number of service replicas, you should maintain separate stack files for each of your environments, such as dev, test, and prod.

15: Security in Docker

Good security is all about *layers* and *defence in depth*. Docker supports all the major Linux security technologies as well as plenty of its own.

In this chapter, we'll look at some of the technologies that make running containers very secure.

Large parts of the chapter will be specific to Linux. However, the **Docker security technologies** section is platform agnostic and applies equally to Linux and Windows.

Security in Docker - The TLDR

Security is about layers, and more layers = more secure. Fortunately, we can apply lots of layers of security to Docker. Figure 15.1 shows some of the security-related technologies we'll cover in the chapter.

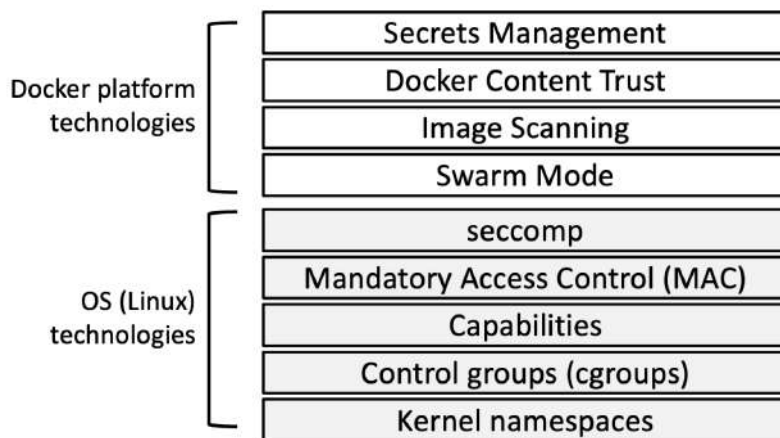


Figure 15.1

Docker on Linux leverages most of the common Linux security and workload isolation technologies. These include *namespaces*, *control groups*, *capabilities*, *mandatory access control (MAC)*, and *seccomp*. For each one, Docker ships with “sensible defaults” for a *moderately secure* out-of-the-box experience. However, you can customize each one to your own specific requirements.

Docker also adds some of its own excellent security technologies. One of the best things about the Docker security technologies is that they're **amazingly simple to use**.

Docker Swarm Mode is secure by default. You get all of the following out of the box: cryptographic node IDs, mutual authentication, automatic CA configuration, automatic certificate rotation, encrypted cluster store, encrypted networks, and more.

Image vulnerability scanning analyses images, detects known vulnerabilities, and provides detailed reports and fixes.

Docker Content Trust (DCT) lets us sign our own images and verify the integrity and publisher of images we consume.

Docker secrets let us securely share sensitive data with applications. They're stored in the encrypted cluster store, encrypted over the network, kept in in-memory filesystems when in use, and operate a least-privilege model.

Others exist, but the important thing to know is that Docker works with the major Linux security technologies as well as providing its own extensive and growing set of security technologies. While the Linux security technologies tend to be complex, the native Docker security technologies tend to be simple.

Security in Docker - The deep dive

We all know that security is important. We also know that security can be complicated and boring.

When Docker decided to bake security into the platform, it decided to make it simple and easy. They knew that if security was hard, people wouldn't use it. As a result, most of the security technologies offered by the Docker platform are easy to use. They also ship with sensible defaults — meaning we get a *fairly secure* platform at zero effort. Of course, the defaults aren't perfect, but they're a good starting point.

We'll organize the rest of this chapter as follows:

- Linux security technologies
 - Namespaces
 - Control Groups
 - Capabilities
 - Mandatory Access Control
 - seccomp
- Docker platform security technologies
 - Swarm Mode
 - Vulnerability scanning
 - Docker Content Trust
 - Docker secrets

Linux security technologies

All *good* container platforms use *namespaces* and *cgroups* to build containers. The *best* container platforms integrate with other Linux security technologies such as *capabilities*, *Mandatory Access Control systems* like SELinux and AppArmor, and *seccomp*. As expected, Docker integrates with them all.

In this section, we'll take a quick look at some of the major Linux security technologies used by Docker. We won't go into detail, as I want the main focus of the chapter to be on the security technologies Docker adds.

Namespaces

Kernel namespaces are the main technology used to build containers.

They virtualise operating system constructs such as process trees and filesystems in the same way that hypervisors virtualise physical resources such as CPUs and disks. In the VM model, hypervisors create virtual machines by grouping together things like virtual CPUs, virtual disks, and virtual network cards. Each VM looks, smells, and feels exactly like a physical machine. In the container model, *namespaces* create virtual operating systems by grouping together things like virtual process trees, virtual filesystems, and virtual network interfaces. Each virtual OS is called a container and looks, smells, and feels exactly like a regular OS.

This virtual OS ("container") lets us do really cool things like run multiple web servers on the same host without having port conflicts. It also lets us run multiple apps on the same host without them fighting over shared config files and shared libraries.

A couple of quick examples:

- Namespaces let us run multiple web servers, each on port 443, on a single host with a single OS. To do this we run each web server inside its own *network namespace*. This works because each *network namespace* gets its own IP address and full range of ports. You may have to map each one to a separate port on the Docker host, but each can run without being re-written or reconfigured to use a different port.
- We can run multiple applications, each with their own versions of shared libraries and configuration files. To do this, we run each application inside of its own *mount namespace*. This works because each *mount namespace* can have its own isolated copy of any directory such as */etc*, */var*, or */dev*.

Figure 15.2 shows a high-level example of two web server applications running on a single host and both using port 443. Each web server app is running inside of its own network namespace.

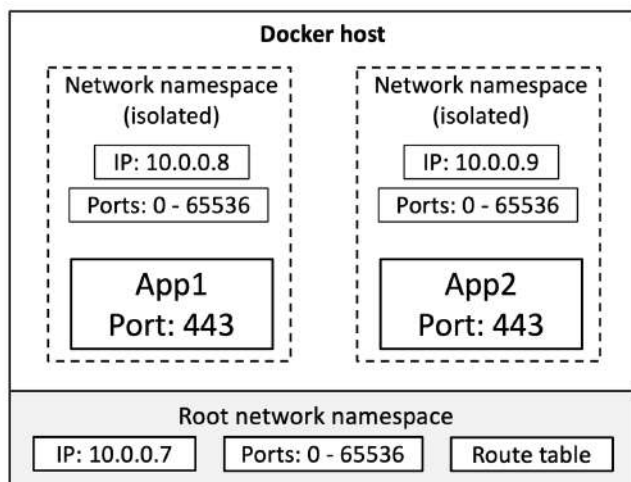


Figure 15.2

Note: The isolation provided by namespaces isn't strong. They need help from some of the other technologies we're going to mention.

Working directly with namespaces is hard. Fortunately, Docker does all the hard work for us and hides all the complexity behind the `docker run` command and an easy-to-use API.

Docker on Linux currently utilizes the following kernel namespaces:

- Process ID (pid)
- Network (net)
- Filesystem/mount (mnt)
- Inter-process Communication (ipc)
- User (user)
- UTS (uts)

We'll explain what each one does in a moment. However, the most important thing to understand is that **containers are an organized collection of namespaces**. For example, every container has its own pid, net, mnt, ipc, uts, and possibly user namespace. In fact, an organized collection of these namespaces is what we call a "container". Figure 15.3 shows a single Linux host running two containers. The host has its own collection of namespaces we call the "root namespaces". Each container has its own collection of isolated namespaces.

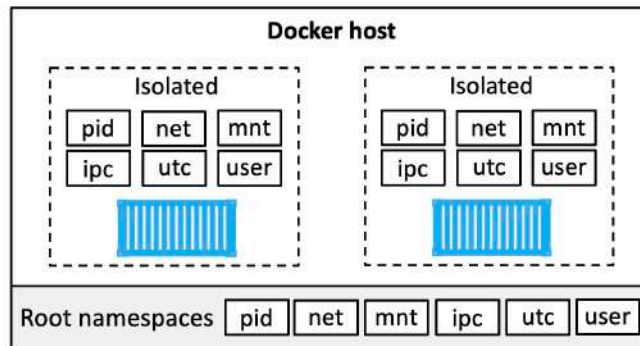


Figure 15.3

Let's briefly look at how Docker uses each namespace:

- **Process ID namespace:** Docker uses the `pid` namespace to provide isolated process trees for each container. This means every container gets its own PID 1. It also means one container cannot see or access the processes running in other containers. Nor can a container see or access the processes running on the host.
- **Network namespace:** Docker uses the `net` namespace to provide each container its own isolated network stack. This stack includes interfaces, IP addresses, port ranges, and routing tables. For example, every container gets its own `eth0` interface with its own unique IP and range of ports.
- **Mount namespace:** Every container gets its own unique isolated root (`/`) filesystem. This means every container can have its own `/etc`, `/var`, `/dev` and other important filesystem constructs. Processes inside a container cannot access the filesystems on the host or other containers — they can only see and access their own isolated filesystem.
- **Inter-process Communication namespace:** Docker uses the `ipc` namespace for shared memory access within a container. It also isolates the container from shared memory outside the container.
- **User namespace:** Docker lets you use user namespaces to map users inside a container to different users on the Linux host. A common example is mapping a container's root user to a non-root user on the Linux host.
- **UTS namespace:** Docker uses the `uts` namespace to provide each container with its own hostname.

Remember, a container is a collection of namespaces that looks like a regular OS, and Docker makes it really easy to use.

Control Groups

If namespaces are about isolation, *control groups* (*cgroups*) are about limits.

Think of containers as similar to rooms in a hotel. While each room might appear isolated, each one shares a common set of infrastructure resources — things like water supply, electricity supply, shared swimming pool, shared gym, shared elevators, shared breakfast bar... Cgroups let us set limits so that (sticking with the hotel analogy) no single container can use all of the water or eat everything at the breakfast bar.

In the real world, not the hotel analogy, containers are isolated from each other but all share a common set of resources — things like CPU, RAM, network and disk I/O. Cgroups let us set limits so a single container cannot consume them all and cause a denial of service (DoS) attack.

Capabilities

It's a bad idea to run containers as `root` — `root` is the most powerful user account on a Linux system and therefore very dangerous. However, it's not as simple as just running containers as regular non-root users. For example, on most Linux systems, non-root users tend to be so powerless they're practically useless. What's needed, is a way to pick-and-choose the specific root powers a container needs in order to run.

Enter *capabilities*!

Under the hood, the Linux `root` user is a combination of a long list of *capabilities*. Some of these *capabilities* include:

- `CAP_CHOWN`: lets you change file ownership
- `CAP_NET_BIND_SERVICE`: lets you bind a socket to low numbered network ports
- `CAP_SETUID`: lets you elevate the privilege level of a process
- `CAP_SYS_BOOT`: lets you reboot the system.

The list goes on and is long.

Docker works with *capabilities* so that you can run containers as `root` but strip out all the capabilities that aren't needed. For example, if the only root capability a container needs is the ability to bind to low numbered network ports, we start a container, drop all root capabilities, then add back just the `CAP_NET_BIND_SERVICE` capability.

This is an excellent example of implementing *least privilege* — we get a container running with only the capabilities we actually need. Docker also imposes restrictions so that containers cannot re-add dropped capabilities.

While this is great, configuring the correct set of capabilities requires a lot of effort and testing.

Mandatory Access Control systems (MAC)

Docker works with major Linux MAC technologies such as AppArmor and SELinux.

Depending on your Linux distribution, Docker applies default profiles to all new containers. According to the Docker documentation, these default profiles are “moderately protective while providing wide application compatibility”.

Docker also lets you start containers without policies, as well as giving you the ability to customize policies to meet specific requirements. This is very powerful but can be prohibitively complex.

seccomp

Docker uses seccomp to limit the syscalls a container can make to the host’s kernel. At the time of writing, Docker’s default seccomp profile disables 44 syscalls. Modern Linux systems have over 300 syscalls.

As per the Docker security philosophy, all new containers get a default seccomp profile configured with *sensible defaults*. As with MAC policies, default seccomp policies are designed to provide *moderate security without impacting application compatibility*.

As always, you can customize seccomp profiles, and you can pass a flag to Docker so that containers can be started without one.

As with many of the technologies already mentioned, seccomp is extremely powerful. However, the Linux syscall table is long, and configuring the appropriate seccomp policies can be prohibitively complex.

Final thoughts on the Linux security technologies

Docker supports most of the important Linux security technologies and ships with sensible defaults that add security but aren’t too restrictive. Figure 15.4 shows how these technologies help build a *defense in depth* security posture.

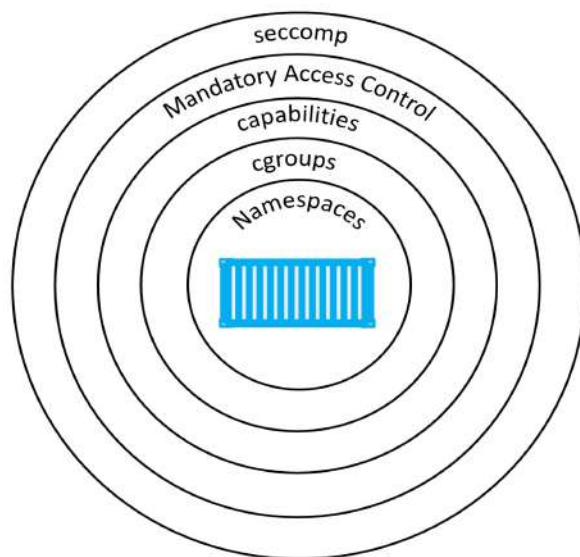


Figure 15.4

Some of these technologies can be complicated to customize as they require deep knowledge of how the Linux kernel works. They're getting simpler to configure, and many platforms, including Docker, ship with defaults that are a good place to start.

Docker security technologies

Let's take a look at some of the major security technologies offered by Docker.

Security in Swarm Mode

Docker Swarm allows you to cluster multiple Docker hosts and deploy applications declaratively. Every Swarm comprises *managers* and *workers* that can be Linux or Windows. Managers host the control plane and are responsible for configuring the cluster and dispatching work tasks. Workers are the nodes that run application containers.

As expected, *swarm mode* includes many security features that are enabled out-of-the-box with sensible defaults. These include:

- Cryptographic node IDs
- TLS for mutual authentication
- Secure join tokens
- CA configuration with automatic certificate rotation

- Encrypted cluster store
- Encrypted networks

Let's walk through the process of building a secure swarm and configuring some of the security aspects.

To follow along with the complete set of examples you'll need three Docker hosts. The examples use three hosts called "mgr1", "mgr2", and "wrk1". There is network connectivity between all three hosts and all three can ping each other by name.

Configure a secure Swarm

Run the following command from the node you want to be the first manager in the new swarm. We'll run the example from **mgr1**.

```
$ docker swarm init
```

```
Swarm initialized: current node (7xam...662z) is now a manager.
To add a worker to this swarm, run the following command:
```

```
docker swarm join --token \
  SWMTKN-1-1dmtwu...r17stb-ehp8g...hw738q 172.31.5.251:2377
```

```
To add a manager to this swarm, run 'docker swarm join-token manager'
and follow the instructions.
```

That's it! That's literally all you need to do to configure a secure swarm.

mgr1 is configured as the first manager of the swarm and also as the root certificate authority (CA). The swarm itself has been given a cryptographic cluster ID. **mgr1** has issued itself with a client certificate that identifies it as a manager, certificate rotation has been configured with the default value of 90 days, and a cluster database has been configured and encrypted. A set of secure tokens have also been created so that additional managers and workers can be securely joined. All of this with a **single command!**

Figure 15.5 shows how the lab currently looks. Some of the details may be different in your lab.

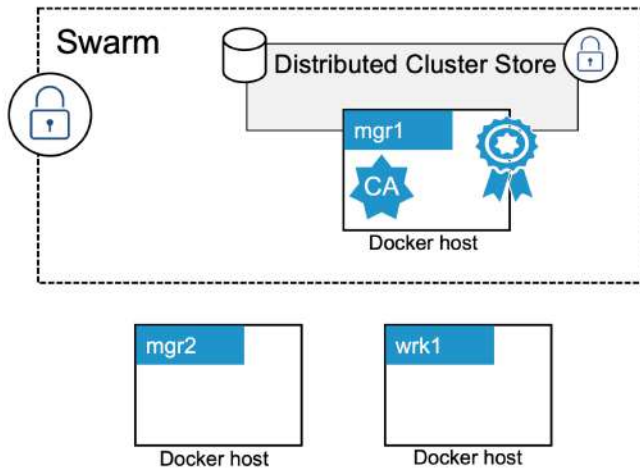


Figure 15.5

Let's join **mgr2** as an additional manager.

Joining new managers to a swarm is a two-step process. The first step extracts the token. The second step runs the `docker swarm join` command on the node we're adding. As long as we include the manager join token as part of the command, **mgr2** will join the swarm as a manager.

Run the following command from **mgr1** to extract the manager join token.

```
$ docker swarm join-token manager
To add a manager to this swarm, run the following command:
```

```
docker swarm join --token \
  SWMTKN-1-1dmtwu...r17stb-2axi5...8p7glz \
  172.31.5.251:2377
```

The output gives us the exact command to run on nodes to join them as managers. The join token and IP address will be different in your lab.

The format of the join **command** is:

- `docker swarm join --token <manager-join-token> <ip-of-existing-manager>:<swarm-port>`

The format of the **token** is:

- `SWMTKN-1-<hash-of-cluster-certificate>-<manager-join-token>`

Copy the command and run it on "mgr2":

```
$ docker swarm join --token SWMTKN-1-ldmtwu...r17stb-2axi5...8p7glz \
> 172.31.5.251:2377
```

This node joined a swarm as a manager.

mgr2 has joined the swarm as an additional manager. In production clusters you should always run either 3 or 5 managers for high availability.

Verify it was successfully added by running a `docker node ls` on either of the two managers.

```
$ docker node ls
ID                HOSTNAME    STATUS    AVAILABILITY    MANAGER STATUS
7xamk...ge662z    mgr1        Ready     Active           Leader
i0ue4...zcjm7f *  mgr2        Ready     Active           Reachable
```

The output shows that **mgr1** and **mgr2** are both part of the swarm and are both managers. The updated configuration is shown in Figure 15.6.

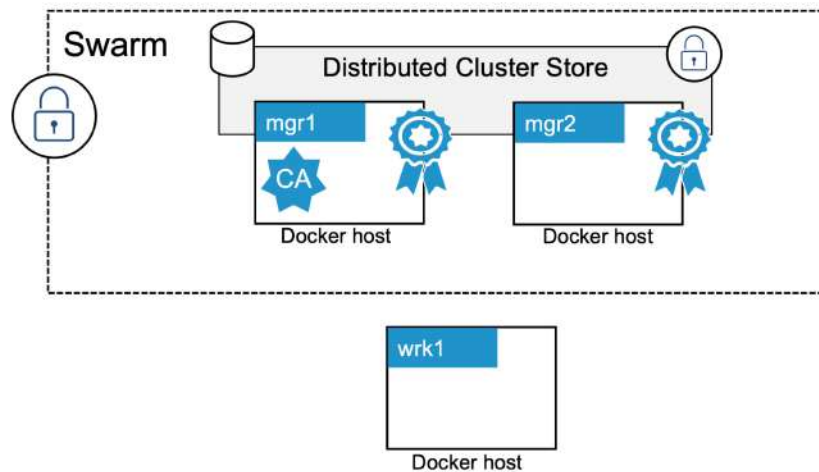


Figure 15.6

Adding a swarm worker is a similar two-step process – extract the join token and run the command on the node.

Run the following command on either of the managers to expose the worker join token.

```
$ docker swarm join-token worker
```

To add a worker to this swarm, run the following command:

```
docker swarm join --token \
  SWMTKN-1-ldmtw...17stb-ehp8g...w738q \
  172.31.5.251:2377
```

Copy the command and run it on **wrk1** as shown:

```
$ docker swarm join --token SWMTKN-1-ldmtw...17stb-ehp8g...w738q \
> 172.31.5.251:2377
```

This node joined a swarm as a worker.

Run another `docker node ls` command from either of the managers.

```
$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
7xamk...ge662z *	mgr1	Ready	Active	Leader
ailrd...ofzv1u	wrk1	Ready	Active	
i0ue4...zcjm7f	mgr2	Ready	Active	Reachable

We now have a swarm with two managers and one worker. The managers are configured for high availability (HA) and the cluster store is replicated to both. The final configuration is shown in Figure 15.7.

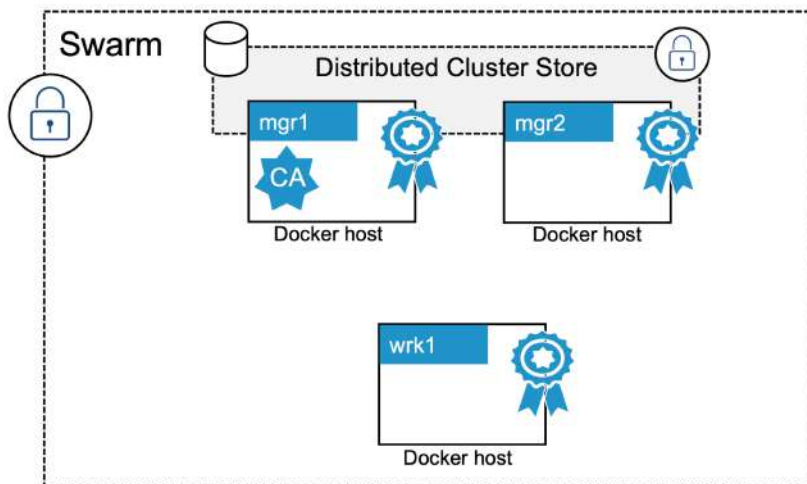


Figure 15.7

Looking behind the scenes at Swarm security

Now that we've built a secure Swarm, let's take a minute to look behind the scenes at some of the security technologies involved.

Swarm join tokens

The only thing that's needed to join new managers and workers to an existing swarm is the correct join token. This means it's vital that we keep our join-tokens safe. Never post them on public GitHub repos or even internal source code repos that are not restricted.

Every swarm maintains two distinct join tokens:

- One for joining new managers
- One for joining new workers

Every join token has 4 distinct fields separated by dashes (-):

PREFIX - VERSION - SWARM ID - TOKEN

The prefix is always `SWMTKN`. This allows you to pattern-match against it and prevent people from accidentally posting it publicly. The `VERSION` field indicates the version of the swarm. The `Swarm ID` field is a hash of the swarm's certificate. The `TOKEN` field is worker or manager token.

As the following shows, the manager and worker join tokens for a Swarm are identical except for the final `TOKEN` field.

- `MANAGER: SWMTKN-1-1dmtwusdc...r17stb-2axi53zjbs45lqxykaw8p7glz`
- `WORKER: SWMTKN-1-1dmtwusdc...r17stb-ehp8gljtji64jbl45zl6hw738q`

If you suspect that either of your join tokens has been compromised, you can revoke them and issue new ones with a single command. The following example revokes the existing *manager* join token and issues a new one.

```
$ docker swarm join-token --rotate manager
```

```
Successfully rotated manager join token.
```

You don't need to update existing managers, but any new managers will need to be added with the new token.

Notice that the only difference between the old and new tokens is the last field. The hash of the Swarm ID remains the same.

Join tokens are stored in the cluster store which is encrypted by default.

TLS and mutual authentication

Every manager and worker that joins a swarm is issued a client certificate that is used for mutual authentication. It identifies the node, the swarm that it's a member of, and whether it's a manager or worker.

You can inspect a node's client certificate on Linux with the following command.

```
$ sudo openssl x509 \
-in /var/lib/docker/swarm/certificates/swarm-node.crt \
-text

Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      7c:ec:1c:8f:f0:97:86:a9:1e:2f:4b:a9:0e:7f:ae:6b:7b:b7:e3:d3
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: CN = swarm-ca
    Validity
      Not Before: May 23 08:23:00 2023 GMT
      Not After : Aug 21 09:23:00 2023 GMT
    Subject: O = tcz3w1t7yu0s4wacovn1rtgp4, OU = swarm-manager,
      CN = 2gxz2h1f0rnmc3atm35qcd1zw
    Subject Public Key Info:
<SNIP>
```

The Subject data in the output uses the standard O, OU, and CN fields to specify the Swarm ID, the node's role, and the node ID.

- The Organization (O) field stores the Swarm ID
- The Organizational Unit (OU) field stores the node's role in the swarm
- The Canonical Name (CN) field stores the node's crypto ID.

This is shown in Figure 15.8.

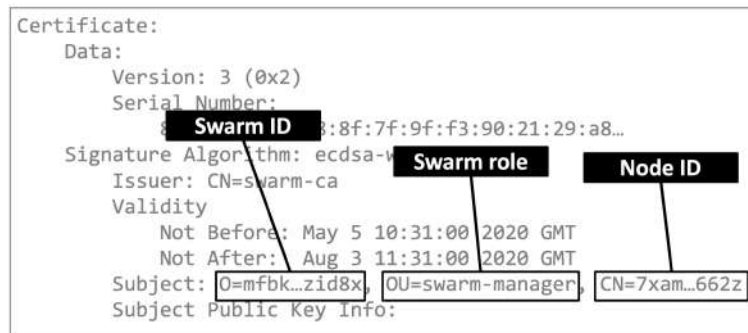


Figure 15.8

You can also see the certificate rotation period in the `Validity` section.

You can match these values to the corresponding values shown in the output of a `docker info` command.

```
$ docker info
<SNIP>
Swarm: active
  NodeID: 2gxz2h1f0rnm3atm35qcd1zw      # Relates to the CN field
  Is Manager: true                      # Relates to the OU field
  ClusterID: tcz3wlt7yu0s4wacovnlrtgp4  # Relates to the O field
<SNIP>
CA Configuration:
  Expiry Duration: 3 months              # Relates to validity field
  Force Rotate: 0
  Root Rotation In Progress: false
<SNIP>
```

Configuring some CA settings

You can configure the certificate rotation period for the Swarm with the `docker swarm update` command. The following example changes the certificate rotation period to 30 days.

```
$ docker swarm update --cert-expiry 720h
```

Swarm allows nodes to renew certificates early so that all nodes don't try and update at the same time.

You can configure an external CA when creating a new swarm by passing the `--external-ca` flag to the `docker swarm init` command.

The `docker swarm ca` command can also be used to manage CA related configuration. Run the command with the `--help` flag to see a list of things it can do.


```
$ docker swarm ca --help
```

```
Usage:  docker swarm ca [OPTIONS]
```

```
Display and rotate the root CA
```

```
Options:
```

<code>--ca-cert pem-file</code>	Path to the PEM-formatted root CA certificate to use for the new cluster
<code>--ca-key pem-file</code>	Path to the PEM-formatted root CA key to use for the new cluster
<code>--cert-expiry duration</code>	Validity period for node certificates (ns us ms s m h) (default 2160h0m0s)
<code>-d, --detach</code>	Exit immediately instead of waiting for the root rotation to converge
<code>--external-ca external-ca</code>	Specifications of one or more certificate signing endpoints
<code>-q, --quiet</code>	Suppress progress output
<code>--rotate</code>	Rotate the swarm CA - if no certificate or key are provided, new ones will be generated

The cluster store

The cluster store is where swarm config and state are stored. It's also critical to other Docker technologies such as overlay networks and secrets. This is why swarm mode is required for so many advanced and security-related features.

The store is currently based on the popular etcd distributed database and is automatically configured to replicate to all managers. It is also encrypted by default.

Day-to-day maintenance of the cluster store is taken care of automatically by Docker. However, in production environments, you should have strong backup and recovery solutions in place.

That's enough for now about swarm mode security. The remainder of the chapter will focus on Docker-related security technologies that don't require swarm mode.

Image vulnerability scanning

Vulnerability scanning is a major weapon against vulnerabilities and security issues in images.

Scanners work by building a list of all software in an image and then comparing the packages against databases of known vulnerabilities. Most vulnerability scanners will rank vulnerabilities and provide advice and help on fixes.

As good as vulnerability scanning is, it's important to understand the limitations. For example, scanning is focussed on images and doesn't detect security problems with

networks, nodes, or orchestrators. Also, not all image scanners are equal — some perform deep binary-level scans to detect packages, whereas others simply look at package names and do not closely inspect content.

At the time of writing, Docker Hub offers image scanning for certain paid accounts. This may change in the future. Some on-premises private registries offer built-in scanning, and there are third-party services that offer image scanning services. Docker Desktop also supports extensions that scan images.

Figure 15.9 shows what a scan result looks like on Docker Hub. Figure 15.10 shows what it looks like with the Trivy Docker Desktop extension.

Severity & Vulnerability	Package	Version	Fixed in
<div>▼ C 9.8 Out-of-bounds Write</div> <div> <div>meta-common-packages meta</div> <div> <div>zlib/zlib1g</div> <div>1:1.2.11.dfsg-2+deb11u1</div> </div> </div>	CVE-2022-37434 zlib	1:1.2.11.dfsg-2+deb11u1	-- -
<div>▶ L 9.8 Out-of-bounds Read</div>	CVE-2019-8457 db5.3	5.3.28+dfsg1-0.8	-- -

Figure 15.9

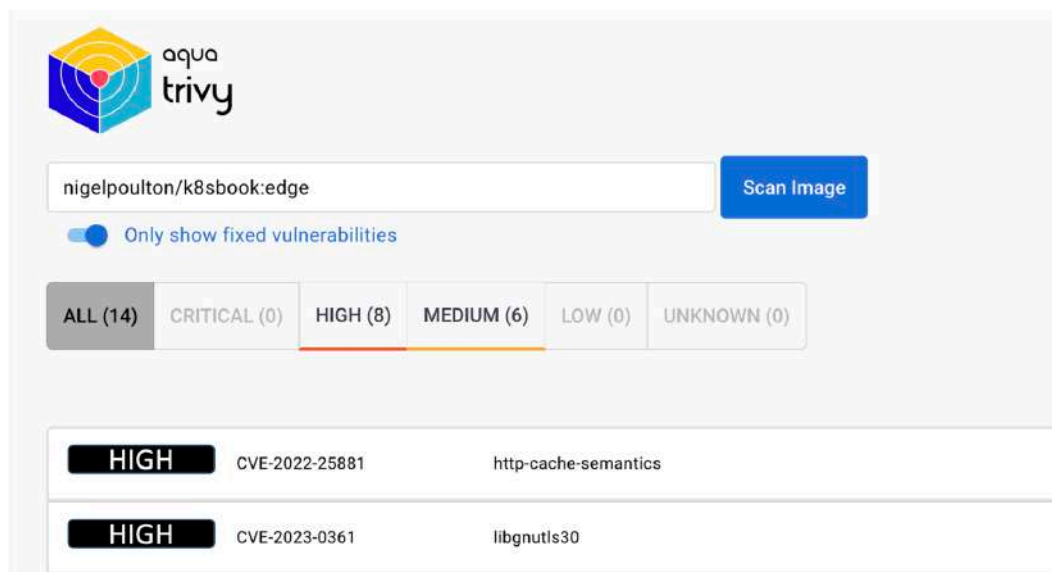


Figure 15.10

In summary, image vulnerability scanning can be a great tool for deeply inspecting your images for known vulnerabilities. Beware though, with great knowledge comes great responsibility — once you become aware of vulnerabilities, you become responsible for mitigating or fixing them.

Signing and verifying images with Docker Content Trust

Docker Content Trust (DCT) makes it simple and easy to verify the integrity and the publisher of images. This is especially important when pulling images over untrusted networks such as the internet.

At a high level, DCT allows developers to sign images when they are pushed to Docker Hub or other container registries. These images can then be verified when they are pulled and ran. This high-level process is shown in Figure 15.11

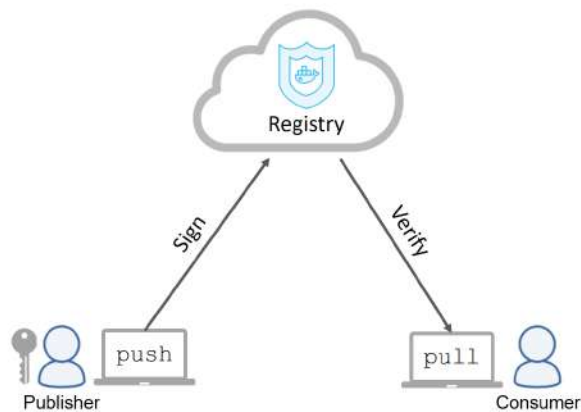


Figure 15.11

DCT can also be used to provide *context*. This includes whether or not an image has been signed for use in a particular environment such as “prod” or “dev”, or whether an image has been superseded by a newer version and is therefore stale.

The following steps will walk you through configuring Docker Content Trust, signing and pushing an image, and then pulling the signed image.

To follow along, you’ll need a cryptographic key-pair to sign images. If you don’t already have one, you can use the `docker trust` command to generate one. The following command generates a new key-pair called “nigel” and loads it to the local trust store ready for use.

```
$ docker trust key generate nigel
Generating key for nigel...
Enter passphrase for new nigel key with ID 1f78609:
Repeat passphrase for new nigel key with ID 1f78609:
Successfully generated and loaded private key.... public key available: /root/nigel.pub
```

If you already have a key-pair, you can import and load it with `docker trust key load key.pem --name nigel`.

Now that we’ve loaded a valid key-pair, we’ll associate it with the image repository we’ll push signed images to. This example uses the `nigelpoulton/ddd-trust` repo on Docker Hub and the `nigel.pub` key that was created in the previous step. Your key file and repo will be different and the repository doesn’t have to exist before you run the command.

```
$ docker trust signer add --key nigel.pub nigel nigelpoulton/ddd-trust
Adding signer "nigel" to nigelpoulton/dct...
Initializing signed repository for nigelpoulton/dct...
Enter passphrase for root key with ID aee3314:
Enter passphrase for new repository key with ID 1a18dd1:
Repeat passphrase for new repository key with ID 1a18dd1:
Successfully initialized "nigelpoulton/dct"
Successfully added signer: nigel to nigelpoulton/dct
```

The following command will sign the `nigelpoulton/ddd-trust:signed` image **and** push it to Docker Hub. You'll need to tag an image on your system with the name of the repo you just associated your key-pair with. I'll push the signed image.

```
$ docker trust sign nigelpoulton/ddd-trust:signed
docker trust sign nigelpoulton/ddd-trust:signed
Signing and pushing trust data for local image nigelpoulton/ddd-trust:signed...
The push refers to repository [docker.io/nigelpoulton/ddd-trust]
94dd7d531fa5: Mounted from library/alpine
signed: digest: sha256:30e6d35703c578e...4fcbbcb0f281 size: 528
Signing and pushing trust metadata
Enter passphrase for nigel key with ID 4d6f1bf:
Successfully signed docker.io/nigelpoulton/ddd-trust:signed
```

The push operation will create the repo on Docker Hub and push the image. You can inspect its signing data with the following command.

```
$ docker trust inspect nigelpoulton/ddd-trust:signed --pretty
```

```
Signatures for nigelpoulton/ddd-trust:signed
```

SIGNED TAG	DIGEST	SIGNERS
signed	30e6d35703c578e...4fcbbcb0f281	nigel

```
List of signers and their keys for nigelpoulton/ddd-trust:signed
```

SIGNER	KEYS
nigel	4d6f1bf55702

```
Administrative keys for nigelpoulton/ddd-trust:signed
```

Repository Key:	5e72e54afafb8444f...6b2744b32010ad22
Root Key:	40418fc47544ca630...69a2cb89028c22092

You can force a Docker host to always sign and verify image push and pull operations by exporting the `DOCKER_CONTENT_TRUST` environment variable with a value of 1. In the real world, you'll want to make this a more permanent feature of Docker hosts.

```
$ export DOCKER_CONTENT_TRUST=1
```

Once DCT is enabled like this, you'll no longer be able to pull and work with unsigned images. You can test this behavior by attempting to pull an unsigned image.

Docker Content Trust is an important technology for helping you verify the images you pull from container registries. It's simple to configure in its basic form, but more advanced features, such as *context*, can be more complex to configure.

Docker Secrets

Many applications have sensitive data such as passwords, certificates, and SSH keys.

Early versions of Docker had no way of making sensitive data like this available to apps in a secure way. We often inserted them into apps via plain text environment variables (we've all done it). Fortunately, modern Docker installations support *Docker secrets*.

Note: Secrets require swarm as they leverage the cluster store.

Behind the scenes, secrets are encrypted at rest, encrypted over the network, mounted into containers via in-memory filesystems, and operate a least-privilege model where they're only made available to services that have been explicitly granted access. There's even a `docker secret` sub-command.

Figure 15.12 shows a high-level workflow:

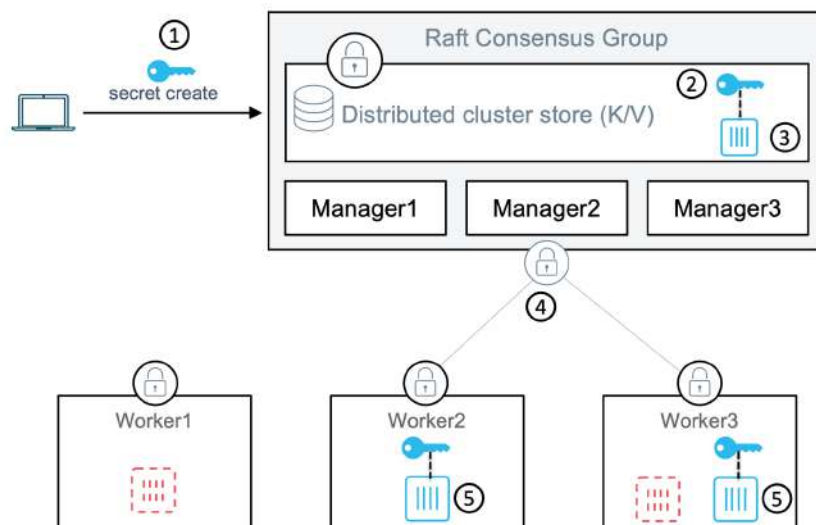


Figure 15.12

The following steps walk through the workflow shown in Figure 15.12. The secret is shown as the key symbol and the container icons with the dashed line are not part of the service that has access to the secret.

1. The secret is created and posted to the Swarm
2. It's stored in the encrypted cluster store
3. The service is created and the secret is attached to it
4. The secret is encrypted over the network while it's delivered to service replicas
5. The secret is mounted into service replicas as an unencrypted file in an in-memory filesystem

As soon as replicas complete, the in-memory filesystem is torn down and the secret flushed from the node. The containers drawn with a dashed lines are not part of the same service and cannot access the secret.

The reason secrets are mounted in their un-encrypted form is so that applications can use them without needing keys to decrypt them.

You can create and manage secrets with the `docker secret` command. You can then attach them to services by passing the `--secret` flag to the `docker service create` command.

Chapter Summary

Docker can be configured to be extremely secure. It supports all of the major Linux security technologies such as kernel namespaces, cgroups, capabilities, MAC, and seccomp. It ships with sensible defaults for all of these, but you can customize them and even disable them.

Over and above the general Linux security technologies, Docker includes an extensive set of its own security technologies. Swarms are built on TLS and are secure out of the box. Scanning tools perform binary-level scans of images and provide detailed reports of known vulnerabilities and suggested fixes. Docker Content Trust lets you sign and verify content, and Docker secrets allow you to securely share sensitive data with swarm services.

The net result is that your Docker environment can be configured to be as secure or insecure as you desire — it all depends on how you configure it.

16: What next

Massive thanks for reading my book. You're on your way to mastering containers!

About the front cover

I love the cover of this book and I'm grateful for the hundreds of people who voted for the design.

The YAML code on the left represents the technical nature of the book. The Docker whale represents the main topic. The vertical symbols on the right are container-related icons done in the style of *digital rain* from the Matrix movies. There's also a hidden message written in Klingon.

Get involved with the community

There's a vibrant container community full of helpful people. Get involved with Docker groups and chats on the internet, and look-up your local Docker or cloud-native meetup (search for "Docker meetup near me").

Kubernetes

Now that you know a thing or two about Docker, a great next-step is Kubernetes – it's a lot like Swarm but has a larger scope and a more active community.

If you liked this book, you'll love my books on Kubernetes.

Feedback and reviews

Books live and die by Amazon reviews and stars.

I've spent well-over a year of my life writing this book and keeping it up-to-date. Soooo... I'd love it if you left a review on Amazon.

Ping me at ddd@nigelpoulton.com if you want to suggest content or fixes for future editions.

The ~~end~~. beginning...

... of the most exciting chapter of your career!

Thanks for reading Docker Deep Dive, I really hope it was useful! It would mean a lot if you would rate and review on Amazon and Goodreads.

I'm always open to feedback and connecting with others in the field.

You can find me on Twitter, Mastodon, LinkedIn and email.



@nigelpoulton

nigelpoulton.com/books

ddd@nigelpoulton.com



Quick Start Kubernetes is the perfect resource to get you up to speed in less than a day. This beginner-friendly guide covers everything from the basics of Kubernetes architecture to features like Pods, Deployments, and Services. At around 100 pages, it's easy to read and understand.



The Kubernetes Book is the go-to guide for mastering Kubernetes and the highest-rated book on Amazon for systems administration and cloud computing. It offers the most precise explanations and practical examples to help software developers, systems administrators, cloud engineers, and architects learn Kubernetes.



The KCNA Book is a comprehensive guide to prepare readers for the Kubernetes and Cloud Native Associate (KCNA) certification, offering concise explanations, quizzes, and a sample exam to master Kubernetes and cloud-native technologies.

