DOCKER DEEP DIVE

ZERO TO DOCKER IN A SINGLE BOOK

2025 Edition

By Docker Captain
Nigel Poulton

Docker Deep Dive

Jan 2025

Zero to Docker in a single book!

Nigel Poulton @nigelpoulton

About this edition

This edition was published in January 2025 and is up to date with the latest industry trends and the latest enhancements to Docker.

Major changes include:

- Added AI content — new AI chatbot app that uses Ollama and LLMs

- Updates to BuildKit, buildx, and Docker Build Cloud

- Updates to include the GA version of the **docker init** command

- Updates to Docker Debug content

- Updates to Wasm content

Enjoy the book and get ready to master containers!

Thank you to everyone who reads my books and watches my training videos. I'm incredibly passionate about all of my content, and it means a lot to me when you connect and give me feedback. I also love it when we bump into each other at events, airports, and all the other random places we meet. Please come and say hello—don't be shy!

*A special shout out to **Niki Kovacs** for his extensive feedback on the 2023 edition. I'm grateful to every reader who gives feedback, but Niki paid great attention to the book and was very generous in his feedback and patience with my slow responses. You can find Niki at:*

ttps://www.microlinux.fr

https://blog.microlinux.fr

https://twitter.com/microlinux_eu

About the author

Nigel Poulton (@nigelpoulton)

Nigel is a technology geek who is passionate about learning new technologies and making them easier for others to learn. He's the author of best-selling books on Docker and Kubernetes, and is the author of **AI Explained: Facts, Fiction, and Future**, an exciting read into the impacts of AI on society and the future of humanity. .

Nigel is a Docker Captain and has held senior technology roles at large and small enterprises.

In his free time, he listens to audiobooks and coaches youth football (soccer). He wishes he lived in the future and could understand the mysteries of life and the universe. He's

passionate about learning, cars, and football. He lives in England with his fabulous wife and three children.

- LinkedIn: [Nigel Poulton](#)

- Web: [nigelpoulton.com](#)

- BlueSky: [@nigelpoulton](#)

- X: [@nigelpoulton](#)

- Email: ddd@nigelpoulton.com

Table of Contents

0: About the book

This **2025 edition** gets you up to speed with Docker and containers fast. No prior experience required.

Why should I read this book or care about Docker?

Docker has already changed how we build, share, and run applications, and it's poised to play a major role with emerging technologies such as Wasm and AI.

So, if you want the best jobs working with the best technologies, you need a strong Docker skillset. It will even give you a head start learning and working with Kubernetes.

How I've organized the book

I've divided the book into two main sections:

1. The big picture stuff
2. The technical stuff

The big picture stuff gets you up to speed with the basics, such as *what Docker is, why we use containers,* and fundamental jargon such as *cloud-native, microservices,* and *orchestration.*

The technical stuff section covers everything you need to know about *images, containers, multi-container microservices apps,* and the increasingly important topic of *orchestration*. It even covers WebAssembly, AI, vulnerability scanning, debugging containers, high availability, and more.

Chapter breakdown

- **Chapter 1:** Summarizes the history and future of Docker and containers
- **Chapter 2:** Explains the most important container-related standards and projects
- **Chapter 3:** Shows you a few ways to get Docker
- **Chapter 4:** Walks you through deploying your first container
- **Chapter 5:** Deep dive into the Docker Engine architecture
- **Chapter 6:** Deep dive into images and image management
- **Chapter 7:** Deep dive into containers and container management
- **Chapter 8:** Deep dive into containerizing applications
- **Chapter 9:** Walks you through deploying and managing a multi-container AI chatbot app with Docker Compose
- **Chapter 10:** Walks you through building, containerizing, and running a Wasm app with Docker
- **Chapter 11:** Builds a highly-available and secure Swarm cluster
- **Chapter 12:** Walks you through deploying, scaling, and self-healing a multi-container app running on Swarm
- **Chapter 13:** Deep dive into Docker networking
- **Chapter 14:** Walks you through building and working with overlay networks

- **Chapter 15:** Introduces you to persistent and non-persistent data in Docker
- **Chapter 16:** Covers all the major Linux and Docker security technologies

Editions and updates

Docker, AI, and the cloud-native ecosystem are evolving fast, and 2-3-year-old books are dangerously out of date. As a result, I'm committed to updating the book every year.

If that sounds excessive, welcome to the new normal.

The book is available in hardback, paperback, and e-book on all good book publishing platforms.

Kindle updates

Unfortunately, Kindle readers cannot get updates — even if you delete the book and buy it again, you'll get the older version you originally purchased. I have no control over this and was devastated when this change happened.

Feedback

If you like the book and it helps your career, share the love by recommending it to a friend and leaving a review on Amazon or

Goodreads.

If you spot a typo or want to make a recommendation, email me at **ddd@nigelpoulton.com** and I'll do my best to respond.

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, you'll learn why we have containers, what they do for us, and where we can use them.

The bad old days

Applications are the powerhouse of every modern business. When applications break, businesses break.

Most applications run on servers, and in the past, we were limited to running one application per server. As a result, the story went something like this:

Every time a business needed a new application, it had to buy a new server. Unfortunately, we weren't very good at modeling the performance requirements of new applications, and we had to guess. This resulted in businesses buying bigger, faster, and more expensive servers than necessary. After all, nobody wanted underpowered servers incapable of handling the app, resulting in unhappy customers and lost revenue. As a result, we ended up with racks and racks of overpowered servers operating as low as 5-10% of their potential capacity. This was a tragic waste of company capital and environmental resources.

Hello VMware!

Amid all this, VMware, Inc. gave the world a gift — the *virtual machine (VM)* — a technology that allowed us to run multiple business applications on a single server safely.

It was a game-changer. Businesses could run new apps on the spare capacity of existing servers, spawning a golden age of maximizing the value of existing assets.

VMwarts

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

For example, every VM needs its own dedicated operating system (OS). Unfortunately, this has several drawbacks, including:

- Every OS consumes CPU, RAM, and other resources we'd rather use on applications
- Every VM and OS needs patching
- Every VM and OS needs monitoring

VMs are also slow to boot and not very portable.

Hello Containers!

While most of us were reaping the benefits of VMs, web scalers like Google had already moved on from VMs and were using containers.

A feature of the container model is that every container shares the OS of the host it's running on. This means a single host can run more containers than VMs. For example, a host that can run 10 VMs might be able to run 50 containers, making containers far more efficient than VMs.

Containers are also faster and more portable than VMs.

Linux containers

Modern containers started in the Linux world and are the product of incredible work from many people over many years. For example, Google contributed many container-related technologies to the Linux kernel. It's thanks to many contributions like these that we have containers today.

Some of the major technologies underpinning modern containers include *kernel namespaces, control groups (cgroups),* and *capabilities.*

However, despite all this great work, containers were incredibly complicated, and it wasn't until Docker came along that they

became accessible to the masses.

> **Note:** *I know that many container-like technologies pre-date Docker and modern containers. However, none of them changed the world the way Docker has.*

Hello Docker!

Docker was the magic that made Linux containers easy and brought them to the masses. We'll talk a lot more about Docker in the next chapter.

Docker and Windows

Microsoft worked 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 and require a host system with a Windows kernel. Windows 10, Windows 11, and all modern versions of Windows Server natively support Windows containers.

Windows systems can also run Linux containers via the *WSL 2 (Windows Subsystem for Linux)* subsystem.

This means Windows 10 and Windows 11 are great platforms for developing and testing Windows **and** Linux containers.

However, despite all the work developing *Windows containers*, almost all containers are Linux containers. This is because Linux containers are smaller and faster, and more 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 containers share the kernel of the host they're 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. However, as mentioned, you can run Linux containers on Windows systems that have the WSL 2 backend installed.

> **Terminology:** *A containerized app is an application running as a container. We'll cover this in a lot of detail later.*

What about Mac containers?

There is no such thing as Mac containers. However, Macs are great platforms for working with containers, and I do all of my daily work with containers on a Mac.

The most popular way of working with containers on a Mac is *Docker Desktop*. It works by running Docker inside a lightweight Linux VM on your Mac. Other tools, such as Podman and Rancher Desktop, are also great for working with containers on a Mac.

What about Wasm

Wasm (WebAssembly) is a modern binary instruction set that builds applications that are smaller, faster, more secure, and more portable than containers. You write your app in your favorite language and compile it as a Wasm binary that will run anywhere you have a Wasm runtime.

However, Wasm apps have many limitations, and we're still developing many of the standards. As a result, containers remain the dominant model for cloud-native applications.

The container ecosystem is also much richer and more mature than the Wasm ecosystem.

As you'll see in the Wasm chapter, Docker and the container ecosystem are adapting to work with Wasm apps, and you should expect a future where VMs, containers, and Wasm apps run side-by-side in most clouds and applications.

This book is up-to-date with the latest Wasm and container developments.

Docker an AI

Containers are already the industry standard for packaging and running applications, and Docker is the most popular developer tool for working with containers. As such, developers and businesses are turning to Docker as the preferred platform for developing and deploying AI and ML applications such as LLMs and other GenAI apps.

However, a vital part of any AI or ML application is access to specialized hardware such as Graphics Processing Units (GPU), Tensor Processing Units (TPU), and Neural Processing Units (NPU). At the time of writing, Docker supports NVIDIA GPUs. In the future, we should expect Docker to support other GPUs, TPUs, NPUs, and even [WebGPU](#).

Later in the book, you'll use Docker to deploy LLM applications that can run on GPUs or regular CPUs.

What about Kubernetes

Kubernetes is the industry standard platform for *deploying and managing* containerized apps.

Older versions of Kubernetes used Docker to start and stop containers. However, newer versions use *containerd*, which is a stripped-down version of Docker optimized for use by Kubernetes and other platforms.

The important thing to know is that all Docker containers work on Kubernetes.

Check out these books if you need to learn Kubernetes:

- **Quick Start Kubernetes:** This is ~100 pages and will get you up-to-speed with Kubernetes **in a single day!**
- **The Kubernetes Book**. This is the ultimate book for mastering Kubernetes.

★★★★⯪ 100+ ratings   ★★★★⯪ 1,300+ ratings

I update both books annually to ensure they're up-to-date with the latest and greatest developments in the cloud native ecosystem.

Chapter Summary

We used to live in a world where every business application needed a dedicated, overpowered server. VMware came along and allowed us to run multiple applications on new and existing servers. However, following the success of VMware and hypervisors, a newer, more efficient, and portable virtualization technology called *containers* came along. However, containers were complex and hard to implement until Docker came along and made them easy. Wasm and AI are powering new innovations, and the Docker ecosystem is evolving to work with

both. The book has entire chapters dedicated to working with AI apps and Wasm apps on Docker.

2: Docker and container-related standards and projects

This chapter introduces you to Docker and some of the most important standards and projects shaping the container ecosystem. The goal is to lay some foundations that we'll build on in later chapters.

This chapter has two main parts:

- Docker
- Container-related standards and projects

Docker

Docker is at the heart of the container ecosystem. However, the term *Docker* can mean two things:

1. The Docker platform
2. Docker, Inc.

The *Docker platform* is a neatly packaged collection of technologies for creating, managing, and orchestrating containers. *Docker, Inc.* is the company that created the Docker platform and continues to be the driving force behind developing new features.

Let's dive a bit deeper.

Docker, Inc.

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

The company started as a *platform as a service (PaaS)* provider called *dotCloud*. Behind the scenes, dotCloud delivered its services on top of containers and had an in-house tool to help them deploy and manage those containers. They called this in-house tool *Docker*.

The word *Docker* is a British expression short for *dock worker* referring to someone who loads and unloads cargo from ships.

In 2013, dotCloud dropped the struggling PaaS side of the business, rebranded as **Docker, Inc.**, and focused on bringing Docker and containers to the world.

The Docker technology

The Docker platform makes it easy to *build, share,* and *run* containers.

At a high level, there are two major parts to the Docker platform:

- The CLI (client)
- The engine (server)

The *CLI* is the familiar `docker` command-line tool for deploying and managing containers. It converts simple commands into API requests and sends them to the engine.

The *engine* comprises all the server-side components that run and manage containers.

Figure 2.1 shows the high-level architecture. The client and engine can be on the same host or connected over the network.
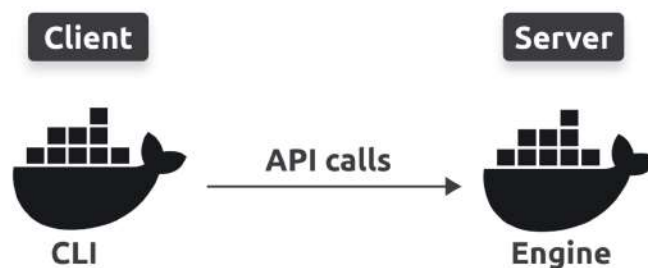
Figure 2.1 Docker client and engine.

In later chapters, you'll see that the client and engine are complex and comprise a lot of small specialized parts. Figure 2.2 gives you an idea of some of the complexity behind the engine. However, the client hides all this complexity so you don't have to care. For example, you type friendly `docker` commands into the CLI, the CLI converts them to API requests and sends them to the daemon, and the daemon takes care of everything else.
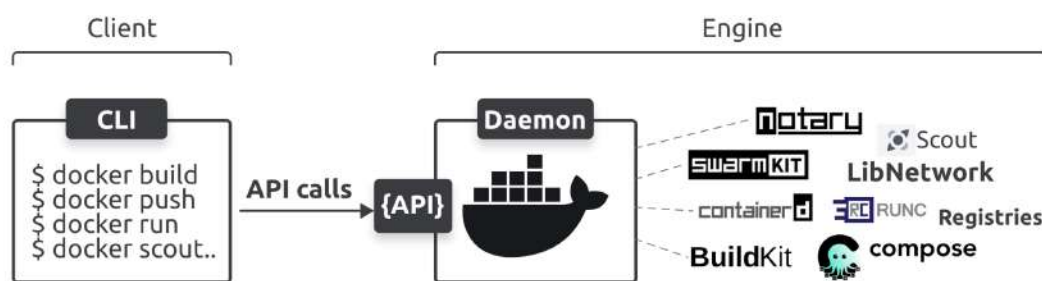
Figure 2.2 Docker CLI and daemon hiding complexity.

Let's switch focus and briefly look at some standards and governance bodies.

Container-related standards and projects

Several important standards and governance bodies influence container development and the container ecosystem. Some of these include:

- The OCI
- The CNCF
- The Moby Project

The Open Container Initiative (OCI)

The Open Container Initiative (OCI) is a governance council responsible for low-level container-related standards.

It operates under the umbrella of the Linux Foundation and was founded in the early days of the container ecosystem when some of the people at a company called CoreOS didn't like the way Docker was dominating the ecosystem. In response, CoreOS created an open standard called **appc** that defined specifications for things such as image format and container runtime. They also created a reference implementation called **rkt** (pronounced "rocket").

The appc standard did things differently from Docker and put the ecosystem in an awkward position with two competing *standards*.

While competition is usually a good thing, *competing standards* are generally bad, as they generate confusion that slows down user adoption. Fortunately, the main players in the ecosystem came together and formed the OCI as a vendor-neutral

lightweight council to govern container standards. This allowed us to archive the appc project and place all low-level container-related specifications under the OCI's governance.

At the time of writing, the OCI maintains three standards called *specs*:

- The [image-spec](#)
- The [runtime-spec](#)
- The [distribution-spec](#)

We often use a *rail tracks* analogy when explaining the OCI standards:

When the size and properties of rail tracks were standardized, it gave entrepreneurs in the rail industry confidence the trains, carriages, signaling systems, platforms, and other products they built would work with the standardized tracks — nobody wanted competing standards for track sizes.

The OCI specifications did the same thing for the container ecosystem and it's flourished ever since. Docker has also changed a lot since the formation of the OCI, and all modern versions of Docker implement all three OCI specs. For example:

- The Docker builder (BuildKit) creates *OCI compliant-images*

- Docker uses an *OCI-compliant runtime* to create *OCI-compliant containers*
- Docker Hub implements the OCI distribution spec and is an *OCI-compliant registry*

Docker, Inc. and many other companies have people serving on the OCI's technical oversight board (TOB).

The CloudNative Computing Foundation (CNCF)

The [Cloud Native Computing Foundation (CNCF)](#) is another Linux Foundation project that is influential in the container ecosystem. It was founded in 2015 with the goal of *"...advancing container technologies... and making cloud native computing ubiquitous"*.

Instead of creating and maintaining container-related specifications, the CNCF *hosts* important projects such as Kubernetes, containerd, Notary, Prometheus, Cilium, and lots more.

When we say the CNCF *hosts* these projects, we mean it provides a space, structure, and support for projects to grow and mature. For example, all CNCF projects pass through the following three phases or stages:

- Sandbox
- Incubating
- Graduated

Each phase increases a project's maturity level by requiring higher standards of governance, documentation, auditing, contribution tracking, marketing, community engagement, and more. For example, new projects accepted as sandbox projects may have great ideas and great technology but need help and resources to create strong governance, etc. The CNCF helps with all of that.

Graduated projects are considered *ready for production* and are guaranteed to have strong governance and implement good practices.

If you look back to Figure 2.2, you'll see that Docker uses at least two CNCF technologies — containerd and Notary.

The Moby Project

Docker created the *Moby project* as a community-led place for developers to build specialized tools for building container platforms.

Platform builders can pick the specific Moby tools they need to build their container platform. They can even compose their platforms using a mix of Moby tools, in-house tools, and tools from other projects.

Docker, Inc. originally created the Moby project, but it now has members including Microsoft, Mirantis, and Nvidia.

The Docker platform is built using tools from various projects, including the Moby project, the CNCF, and the OCI.

Chapter summary

This chapter introduced you to Docker and some of the major influences in the container ecosystem.

Docker, Inc., is a technology company based in Palo Alto that is changing how we do software. They were the *first movers* and instigators of the modern container revolution.

The Docker platform focuses on running and managing application containers. It runs on Linux and Windows, can be installed almost anywhere, and offers a variety of free and paid-for products.

The Open Container Initiative (OCI) governs low-level container standards and maintains specifications for runtimes, image

format, and registries.

The CNCF provides support for important cloud-native projects and helps them mature into production-grade tools.

The Moby project hosts low-level tools developers can use to build container platforms.

3: Getting Docker

There are lots of ways to get Docker and work with containers. This chapter will show you the following ways:

- Docker Desktop
- Multipass
- Server installs on Linux

I strongly recommend you install and use Docker Desktop. It's the best way to work with Docker, and you'll be able to use it to follow most of the examples in the book. I use it every day.

If you can't use Docker Desktop, we'll show you how to install Docker in a Multipass VM, as well as how to perform a simple installation on Linux. However, these installations don't have all the features of Docker Desktop.

Docker Desktop

Docker Desktop is a desktop app from Docker, Inc. and is the best way to work with containers. You get the Docker Engine, a slick UI, all the latest plugins and features, and an extension system with a marketplace. You even get Docker Compose and a Kubernetes cluster if you want to learn Kubernetes.

It's free for personal use and education, but you'll have to pay a license fee if you use it for work and your company has over 250 employees or does more than $10M in annual revenue.

Docker Desktop on Windows 10 and Windows 11 Professional and Enterprise editions supports *Windows containers* **and** *Linux containers.* Docker Desktop on Mac, Linux, and Home editions of Windows only support *Linux containers.* All of the examples in the book and almost all of the containers in the real world are Linux containers.

Let's install Docker Desktop on Windows and MacOS.

Windows prereqs

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 for *"install Docker Desktop on Windows"*. This will take you to the relevant download page, where you can download the installer and follow the instructions. When prompted, you should install and enable the WSL 2 backend (Windows Subsystem for Linux).

Once the installation is complete, you need 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 taskbar at the bottom of the screen.

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

```
$ docker version
<Snip>
Server: Docker Desktop 4.37.0 (177092)
 Engine:
  Version:          27.4.0
  API version:      1.47 (minimum version 1.24)
  Go version:       go1.22.9
  OS/Arch:          linux/amd64
<Snip>
```

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

Notice how the `Server` output shows `OS/Arch: linux/amd64`. This is because a default installation assumes you'll be working with Linux containers.

Some versions of Windows let you switch to *Windows containers* by right-clicking the Docker whale icon in the Windows notifications tray and selecting **Switch to Windows containers....** Doing this keeps existing Linux containers running in the background, but you won't be able to see or manage them until you switch back to Linux containers mode.

Make sure you're running in *Linux containers mode* so you can follow along with the examples later in the book.

Installing Docker Desktop on Mac

Docker Desktop for Mac is like Docker Desktop for Windows — a packaged product with a slick UI that gets you the full Docker experience on your laptop.

Before proceeding with the installation, you need to know that Docker Desktop on Mac installs the daemon and server-side components inside 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 run `docker` commands without ever knowing it's all running in a hidden VM. This is

also why Mac versions of Docker Desktop only work with Linux containers — everything's running inside a Linux VM.

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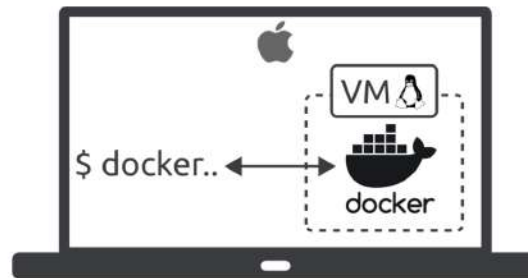
Figure 3.1

The simplest way to install Docker Desktop on your Mac is to search the web for *"install Docker Desktop on MacOS"*, follow the links to the download, and then complete the simple installer.

When the installer finishes, you'll have to 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:
 Version:           27.4.0-rc.3
 API version:       1.47
 OS/Arch:           darwin/arm64
<Snip>
Server: Docker Desktop 4.37.0 (177092)
 Engine:
  Version:          27.4.0-rc.3
  API version:      1.47 (minimum version 1.24)
  OS/Arch:          linux/arm64
 containerd:
  Version:          1.7.21
 runc:
  Version:          1.1.13
 docker-init:
  Version:          0.19.0
<Snip>
```

Notice that the `OS/Arch:` for the **Server** component shows as `linux/amd64` or `linux/arm64`. This is because the daemon runs inside the Linux VM mentioned earlier. The **Client** component is a native Mac application and runs directly on the Mac OS Darwin kernel. This is why it shows as `darwin/amd64` or `darwin/arm64`.

You can now use Docker on your Mac.

Installing Docker with Multipass

Only consider this section if you can't use Docker Desktop.

Multipass installations don't ship with out-of-the-box support for features such as `docker scout`, `docker debug`, and `docker init`.

Multipass is a free tool for creating cloud-style Linux VMs on your Linux, Mac, or Windows machine and is incredibly easy to install and use. It's an easy way to create multi-node production-like Docker clusters.

Go to `https://multipass.run/install` and install the right edition for your hardware and OS.

Once installed, you only need three commands:

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

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 yours launched properly.

```
$ multipass ls

Name                      State            IPv4                Image
node1                     Running          192.168.64.37       Ubuntu 24.04 LT
                                           172.17.0.1
                                           172.18.0.1
```

You'll use the **192.168.x.x** IP address when working with the examples later in the book.

Connect to the VM with the following command.

```
$ multipass shell node1
```

Once connected, you can run the following commands to check your Docker version and list installed CLI plugins.

```
$ docker --version
Docker version 26.1.0, build 9714adc

$ docker info
```

```
Client: Docker Engine - Community
 Version:     27.3.1
 Context:     default
 Debug Mode: false
  Plugins:
  buildx: Docker Buildx (Docker Inc.)
    Version:  v0.17.1
    Path:      /usr/libexec/docker/cli-plugins/docker-buildx
  compose: Docker Compose (Docker Inc.)
    Version:  v2.29.7
    Path:      /usr/libexec/docker/cli-plugins/docker-compose
<Snip>
```

You can type `exit` to log out of the VM, and `multipass shell node1` to log back in. You can also type `multipass delete node1` and then `multipass purge` to delete it.

Installing Docker on Linux

Only consider this section if you can't use Docker Desktop, as it doesn't give you access to `docker scout`, `docker debug`, or `docker init`.

These instructions show you how to install Docker on Ubuntu Linux 24.04 and are just for guidance purposes. Lots of other installation methods exist, and you should search the web for the latest instructions.

```
$ sudo snap install docker
<Snip>
```

```
docker 27.2.0 from Canonical✓ installed
```

Run some commands to test the installation. You'll have to prefix them with `sudo`.

```
$ sudo docker --version
Docker version 27.2.0, build 3ab4256

$ sudo docker info
<Snip>
Server:
 Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
 Images: 0
 Server Version: 27.2.0
<Snip>
```

If you don't like adding `sudo` before Docker commands, you can run the following commands to create a **docker** group and add your user account to it.

```
$ sudo groupadd docker

$ sudo usermod -aG docker $(whoami)
```

You'll need to restart Docker for the changes to take effect. This is how you restart Docker on many Ubuntu Linux distributions.

Yours may be different.

```
$ sudo service docker start
```

Chapter Summary

You can run Docker almost anywhere, and installing it's easier than ever.

Docker Desktop gives you a fully functional Docker environment on your Linux, Mac, or Windows machine and is the best way to get a Docker development environment on your local 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 works with `docker scout`, `docker debug`, and `docker init`, and it even lets you spin up a Kubernetes cluster.

Multipass is a great way to spin up a local VM running Docker, and there are lots of ways to install Docker on Linux servers. These give you access to most of the free Docker features but lack some of the features of Docker Desktop.

4: The big picture

This chapter will give you some hands-on experience and a high-level view of images and containers. The goal is to prepare you for more detail in the upcoming chapters.

We'll break this chapter into two parts:

- The Ops perspective
- The Dev perspective

The ops perspective focuses on starting, stopping, deleting containers, and executing commands inside them.

The dev perspective focuses more on the application side of things and runs through taking application source code, building it into a container image, and running it as a container.

I recommend you read both sections and follow the examples, as this will give you the *dev* and *ops* perspectives. *DevOps* anyone?

The Ops Perspective

In this section, you'll complete all of the following:

- Check Docker is working

- Download an image
- Start a container from the image
- Execute a command inside the container
- Delete the container

A typical Docker installation installs the client and the engine on the same machine and configures them to talk to each other.

Run a `docker version` command to ensure both are installed and running.

```
$ docker version
Client:                                    <<---- Start of client respons
 Version:           27.4.0-rc.3            -----¬
 API version:       1.47                        |
 Go version:        go1.22.9                     | Client info block
 OS/Arch:           darwin/arm64                 |
 Context:           desktop-linux          -----┘

Server: Docker Desktop 4.37.0 (177092)     <<---- Start of server respons
 Engine:                                   -----¬
  Version:          27.4.0-rc.3                  |
  API version:      1.47 (minimum version 1.24)  |
  Go version:       go1.22.9                      |
  OS/Arch:          linux/arm64                   |
 containerd:                                | Server block
  Version:          1.7.21                       |
 runc:                                           |
  Version:          1.1.13                        |
 docker-init:                                     |
  Version:          0.19.0                  -----┘
```

If your response from the client **and** server looks like the output in the book, everything is working as expected.

If you're on Linux and get a `permission denied while trying to connect to the Docker daemon...` error, try again with `sudo` in front of the command — `sudo docker version`. If it works with `sudo`, you'll need to prefix **all** future `docker` commands with `sudo`.

Download an image

Images are objects that contain everything an app needs to run. This includes an OS filesystem, the application, and all dependencies. If you work in operations, they're similar to VM templates. If you're a developer, they're similar to *classes*.

Run a `docker images` command.

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

If you are working from a clean installation, you'll have no images, and your output will be the same as the book. If you're working with Multipass, you might see an image called `portainer/portainer-ce`.

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

```
$ docker pull nginx:latest
latest: Pulling from library/nginx
ad5932596f78: Download complete
e4bc5c1a6721: Download complete
1bd52ec2c0cb: Download complete
411a98463f95: Download complete
df25b2e5edb3: Download complete
e93f7200eab8: Download complete
Digest: sha256:fb197595ebe76b9c0c14ab68159fd3c08bd067ec62300583543f0ebda3!
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest
```

Run another `docker images` to confirm your *pull* command
worked.

```
$ docker images
REPOSITORY      TAG        IMAGE ID        CREATED        SIZE
nginx           latest     fb197595ebe7    10 days ago    280MB
```

We'll discuss where the image is stored and what's inside it in
later chapters. For now, all you need to know is that images
contain enough of an operating system (OS) and all the code
and dependencies required to run a desired application. The
NGINX image you pulled includes a stripped-down version of
Linux and the NGINX web server app.

Start a container from the image

If you've been following along, you'll have a copy of the `nginx:latest` image and you can use the `docker run` command to start a container from it.

Run the following `docker run` command to start a new container called **test** from the `ubuntu:latest` image.

```
$ docker run --name test -d -p 8080:80 nginx:latest
e08c3535...30557225
```

The long number confirms the container was created.

Let's quickly examine that `docker run` command.

`docker run` tells Docker to start a new container. The `--name` flag told Docker to call this container **test** and the `-d` flag told it to start the container in the background (detached mode) so it doesn't take over your terminal. The `-p` flag told Docker to map port `80` in the container to port `8080` on your Docker host. Finally, the command told Docker to base the container on the `nginx:latest` image.

Run a `docker ps` command to see the running container.

```
$ docker ps
CONTAINER ID    IMAGE          COMMAND        CREATED      STATUS       PORT
e08c35352ff3    nginx:latest   "/docker..."   3 mins ago   Up 2 mins    0.0.
```

You should recognize the `CONTAINER ID` from the long number printed after the `docker run` command. You should also recognize the `IMAGE` , `PORTS` , and `NAMES` columns from the flags in the `docker run` command. The `COMMAND` field lists the command Docker executed to start the NGINX app inside the container.

Execute a command inside the container

Run the following command to attach your shell to a new Bash process inside the container.

```
$ docker exec -it test bash
root@e08c35352ff3:/#
```

Your shell prompt will change to indicate you're connected to the container.

Run the following command to list files in your current directory.

```
root@e08c35352ff3:/# ls -l
total 64
lrwxrwxrwx   1 root root    7 Jan  2 00:00 bin -> usr/bin
drwxr-xr-x   2 root root 4096 Oct 31 11:04 boot
drwxr-xr-x   5 root root  340 Jan 12 15:09 dev
drwxr-xr-x   1 root root 4096 Jan  3 02:56 docker-entrypoint.d
-rwxr-xr-x   1 root root 1620 Jan  3 02:56 docker-entrypoint.sh
drwxr-xr-x   1 root root 4096 Jan 12 15:09 etc
<Snip>
```

If you're familiar with Linux, you'll recognize these are regular Linux files and directories.

Try running a `ps` command to list running processes.

```
root@e08c35352ff3:/# ps -elf
bash: ps: command not found
```

The command is not found because most containers only ship with essential apps and tools to keep them small and reduce attack vectors. Later in the book, we'll show you how to use Docker Desktop and Docker Debug to connect to running containers and execute commands not included as part of the container.

Type `exit` to terminate your bash process and connect your shell back to your local terminal. Your shell prompt will revert.

Run the following command to verify the **test** container is still running.

```
$ docker ps
CONTAINER ID    IMAGE          COMMAND         CREATED       STATUS       PORT
e08c35352ff3    nginx:latest   "/docker..."    7 mins ago    Up 7 mins    0.0.
```

The container is still running, and you can see it was created 7 minutes ago and has been running for 7 minutes.

Delete the container

Stop and kill the container using the `docker stop` and `docker rm` commands.

```
$ docker stop test
test
```

It can take a few seconds for the container to stop.

```
$ docker rm test
test
```

Verify the container deleted properly by running the `docker ps` command with the `-a` flag to list all containers, even those in the stopped state.

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

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

The Dev Perspective

Containers are all about applications.

You'll complete all of the following steps in this section:

- Clone an app from a GitHub repo
- Inspect the app's *Dockerfile*
- *Containerize* the app
- Run the app as a container

Run the following command to make a local clone of the repo. This will copy the application code to your machine so you can containerize it in a future step. You'll need the `git` CLI for this to work.

```
$ 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 into the `psweb` directory and list its contents.

```
$ cd psweb

$ ls -l
total 32
-rw-r--r--@ 1 nigelpoulton  staff  324  5 Feb 12:31 Dockerfile
-rw-r--r--  1 nigelpoulton  staff  378  5 Feb 12:31 README.md
-rw-r--r--  1 nigelpoulton  staff  341  5 Feb 12:31 app.js
-rw-r--r--@ 1 nigelpoulton  staff  355  5 Feb 12:47 package.json
drwxr-xr-x  3 nigelpoulton  staff   96  5 Feb 12:31 views
```

The app is a simple Node.js web app running some static HTML.

Inspect the app's Dockerfile

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

List the contents of the application's Dockerfile.

```
$ cat Dockerfile

FROM alpine
LABEL maintainer="nigelpoulton@hotmail.com"
RUN apk add --update nodejs npm curl
```

```
COPY . /src
WORKDIR /src
RUN  npm install
EXPOSE 8080
ENTRYPOINT ["node", "./app.js"]
```

You'll learn more about Dockerfiles later in the book. Right now, all you need to know is that each line represents an instruction Docker executes to build the app into an image.

If you've been following along, you've pulled the application code from a remote Git repo and looked at the application's Dockerfile.

Containerize the app

Run the following `docker build` command to create a new image based on the instructions in the Dockerfile. It will create a new Docker image called `test:latest`.

Be sure to run the command from within the `psweb` directory and include the trailing period.

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

```
=> => naming to docker.io/library/test:latest
=> => unpacking to docker.io/library/test:latest
```

When the build completes, check that you have an image called
`test:latest` .

```
$ docker images
REPO      TAG       IMAGE ID        CREATED         SIZE
test      latest    0435f2738cf6    21 seconds ago  160MB
```

Congratulations, you've *containerized* the app. That's jargon for
building it into a container image that contains the app and all
dependencies.

Run the app as a container

Run the following command to start a container called **web1**
from the image. If you're on a Windows machine, you'll need to
replace the backslashes with backticks or run the command on
a single line without the backslashes.

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

Open a web browser and navigate to the DNS name or IP address of your Docker host on port `8080`. If you're following along on Docker Desktop, connect to `localhost:8080` or `127.0.0.1:8080`. If you're following along on Multipass, connect to your Multipass VM's **192.168** address on port `8080`. Run an `ip a | grep 192` command from within the Multipass VM, or run a `multipass ls` from your local machine to find the address.

You will see the following web page.

Figure 4.1

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

Clean up

Run the following commands to terminate the container and delete the image.

```
$ docker rm web1 -f
web1

$ docker rmi test:latest
Untagged: test:latest
Deleted: sha256:0435f27...cac8e2b
```

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 source code from GitHub and building it into an image using instructions in a Dockerfile. You then ran the app as a container.

The things you've learned in this chapter will help you in the upcoming chapters.

Part 2: The technical stuff

5: The Docker Engine

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

This chapter has a strong *operations* focus, and you can use Docker without knowing everything you're about to learn. However, to truly master something, you need to understand what's going on under the hood. So, if you want to **master Docker**, you should read this chapter.

I've divided the chapter into the following sections:

- Docker Engine – The TLDR
- The Docker Engine
- The influence of the Open Container Initiative (OCI)
- runc
- containerd
- Starting a new container (example)
- What's the shim all about
- How it's implemented on Linux

Let's learn about the Docker Engine.

Docker Engine – The TLDR

Docker Engine is jargon for the server-side components of Docker that run and manage containers. If you've ever worked with VMware, the Docker Engine is similar to ESXi.

The Docker Engine is modular and built from many small specialized components pulled from projects such as the OCI, the CNCF, and the Moby project.

In many ways, the Docker Engine is like a car engine:

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

Figure 5.1 shows the components of the Docker Engine that create and run containers. Other components exist, but this simplified diagram focuses on the components that start and run containers.
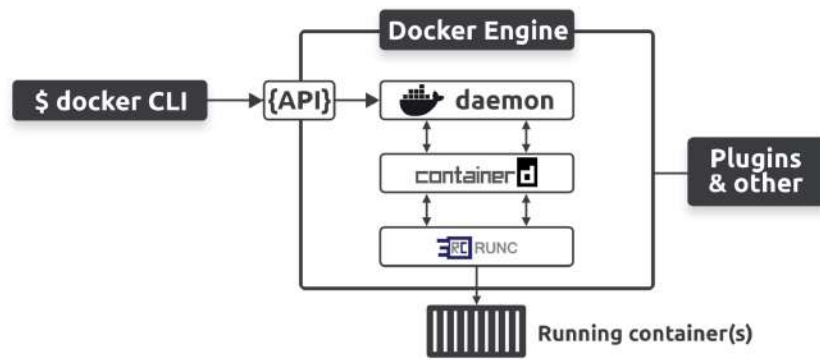
Figure 5.1

Throughout the book, we'll refer to *runc* and *containerd* with lowercase "r" and "c", which is how they're both written in the official project documentation. This means sentences starting with either *runc* or *containerd* will not begin with a capital letter.

The Docker Engine

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

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

The daemon was a monolithic binary containing all the code for the API, image builders, container execution, volumes, networking, and more.

LXC did the hard work of interfacing with the Linux kernel and constructing the required namespaces and cgroups to build and start containers.

Replacing LXC

Relying on LXC posed several problems for the Docker project.

First, LXC is Linux-specific, and Docker had aspirations of being multi-platform.

Second, Docker was evolving fast, and there was no way of ensuring LXC evolved in the ways Docker needed.

To improve the experience and help the project evolve more quickly, Docker replaced LXC with its own tool, *libcontainer*. The goal of libcontainer was to be a platform-agnostic tool that gave Docker access to the fundamental container building blocks in the host kernel.

Libcontainer replaced LXC in Docker a very long time ago.

Breaking up the monolithic Docker daemon

As previously mentioned, the Docker Engine was originally a monolith with almost all functionality coded into the *daemon*.

However, as time passed, this became more and more problematic for the following reasons:

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

The project recognized these challenges and began a long-running program to break apart and refactor the Engine so that every feature became its own small specialized tool. Platform builders could then re-use these tools to build other platforms.

This work of breaking apart the Docker daemon is an ongoing process, and all of the code for building images and executing containers has been removed and refactored into small, specialized tools. Notable examples include removing the high-level and low-level runtime functionality and re-implementing them in separate tools called **containerd** and **runc**, both of which are used by many different projects, including Docker, Kubernetes, Firecracker, and Fargate. More recently (starting with Docker Desktop 4.27.0), Docker has removed image management from the daemon and now uses containerd's image management capabilities.

Figure 5.2 shows another view of the Docker Engine components that are used to run containers and lists the primary responsibilities of each component.
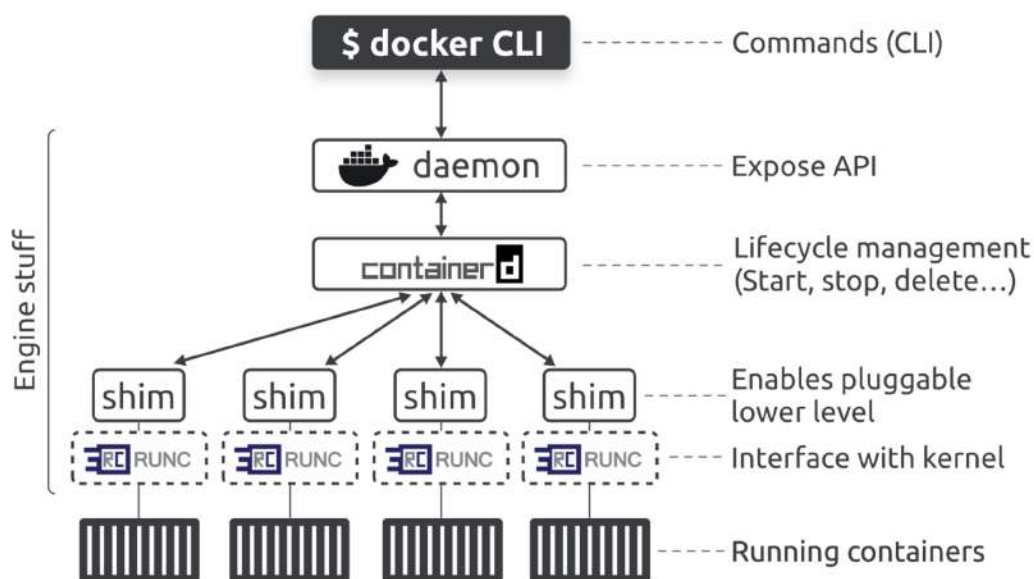
Figure 5.2 - Engine components and responsibilities

Other engine components exist.

The influence of the Open Container Initiative (OCI)

Around the same time that Docker, Inc. was refactoring the Engine, the OCI was in the process of defining two container-related standards:

1. Image Specification (image-spec)

2. Runtime Specification [(runtime-spec)](runtime-spec)

Both specifications were released as version 1.0 in July 2017 and are still vital today.

They've even added a third specification called the Distribution Specification (distribution-spec) governing how images are distributed via registries.

At the time of writing, the runtime-spec is at version 1.2.0, and the image-spec and distribution-spec are both at version 1.1.0. This demonstrates the slow-and-steady nature of these low-level specifications that are heavily relied upon by so many other projects — stability is the name of the game for low-level OCI specs.

Docker, Inc. was a founding member of the OCI and was heavily involved in defining the original specifications. It continues to be involved by contributing code and helping guide the future of the specifications.

All versions of Docker since 2016 have implemented the OCI specifications. For example, Docker uses *runc*, the reference implementation of the OCI runtime-spec, to create OCI-compliant containers (runtime-spec). It also uses BuildKit to

build OCI-compliant images (image-spec), and Docker Hub is an OCI-compliant registry (registry-spec).

runc

As previously mentioned, _runc_ (pronounced "run see" and always written with a lowercase "r") is the reference implementation of the OCI runtime-spec. Docker, Inc. was heavily involved in defining the spec and contributed the initial code for runc.

runc is a lightweight CLI wrapper for libcontainer that you can download and use to manage OCI-compliant containers. However, it's a very low-level tool and lacks almost all of the features and add-ons you get with the Docker Engine. Fortunately, as previously shown in Figure 5.2, Docker uses runc as its low-level runtime. This means you get OCI-compliant containers **and** the feature-rich Docker user experience.

On the jargon front, we sometimes say that runc operates at the _OCI layer_, and we often refer to it as a _low-level runtime._

Docker and Kubernetes both use runc as their default low-level runtime, and both pair it with the containerd high-level runtime:

- containerd operates as the high-level runtime *managing* lifecycle events
- runc operates as the low-level runtime *executing* lifecycle events by interfacing with the kernel to do the work of actually building containers and deleting them

You can see the latest releases here:

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

containerd

containerd (pronounced "container dee" and always written with a lowercase "c") is another tool that Docker created while stripping functionality out of the daemon.

We refer to containerd as a *high-level runtime* as it *manages* lifecycle events such as starting, stopping, and deleting containers. However, it needs a low-level runtime to perform the actual work. Most of the time, containerd is paired with runc as its low-level runtime. However, as you saw in Figure 5.3, it uses *shims* that make it possible to replace runc with other low-level runtimes. We'll go into more detail in the WebAssembly chapter when you'll see how to use Docker to run WebAssembly apps.

The original plan was for containerd to be a small specialized tool for managing container lifecycle events. However, it has since grown to include the ability to manage images, networks, and volumes.

One reason for adding more functionality is for projects such as Kubernetes that want containerd to be able to push and pull images. Fortunately, this extra functionality is modular, meaning projects like Kubernetes can include containerd but only take the pieces they need.

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

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

Starting a new container (example)

Now that you've seen the big picture, let's see how to use Docker to create a new container.

The most common way of starting containers is using the Docker CLI. Feel free to run the following command to start a

new container called **ctr1** based on the `nginx` image.

```
$ docker run -d --name ctr1 nginx
```

Run a **docker ps** command to see if the container is running.

```
$ docker ps
CONTAINER ID    IMAGE     COMMAND                  CREATED         STATUS
9cfb0c9aacb2    nginx     "/docker-entrypoint.…"   9 seconds ago   Up 9 sec
```

When you run commands like this, the Docker client converts them into API requests and sends them to the API exposed by the daemon.

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

The daemon receives the request, interprets it as a request to create a new container, and passes it 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 gRPC.

Despite its name, even containerd cannot create containers. 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 the constructs necessary to create a container (namespaces, cgroups, etc.). The container starts as a child process of runc, and as soon as the container starts, runc exits.

Figure 5.3 summarizes the process.

$ docker CLI — Convert **docker run** command into API request and send to API (daemon)

daemon — Receive API request and instruct **containerd** to start new container

container d — Receive instruction to create new container and pass instruction to **runc**

shim / **RUNC** — **runc** builds new container and exits **shim** becomes container's parent process and communicates with containerd
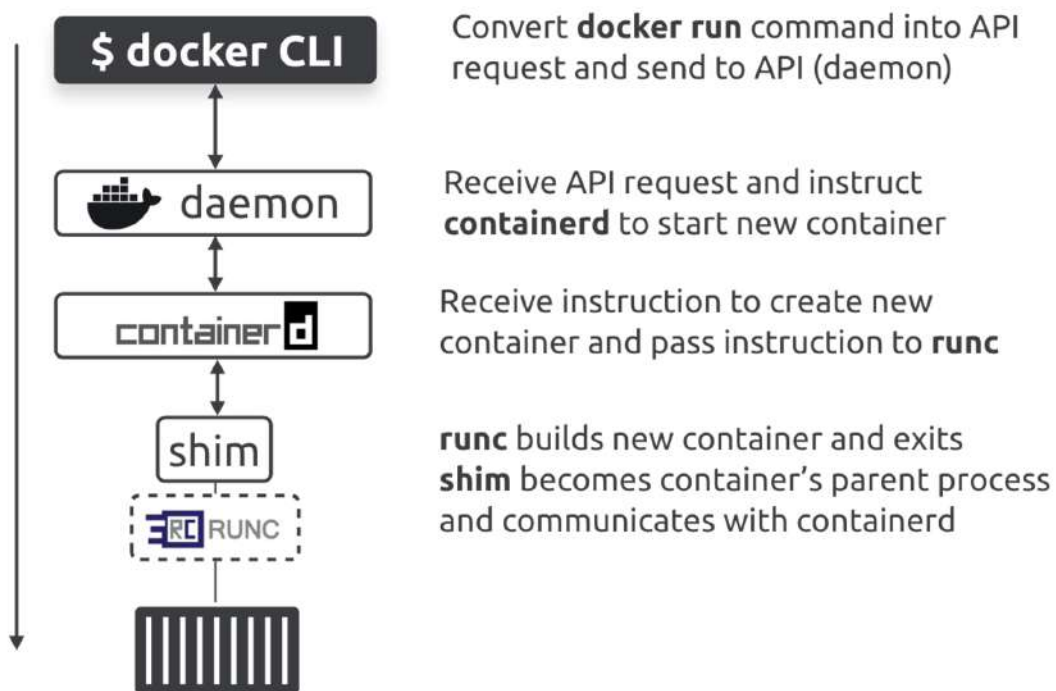
Figure 5.3

Decoupling the container creation and management from the Docker daemon and implementing it in containerd and runc makes it possible to stop, restart, and even update the daemon without impacting running containers. We sometimes call this *daemonless containers.*

If you started the NGINX container earlier, you should delete it using the following command.

```
$ docker rm ctr1 -f
```

What's the shim all about?

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

Shims are a popular software engineering pattern, and the Docker Engine uses them in between containerd and the OCI layer, bringing the following benefits:

- Daemonless containers
- Improved efficiency
- Pluggable OCI layer

We've already said that daemonless containers is the ability to stop, restart, and even update the Docker daemon without

impacting running containers.

On the efficiency front, containerd forks a shim and a runc
process for every new container. However, each runc process
exits as soon as the container starts running, leaving the shim
process as the container's parent process. The shim is
lightweight and sits between containerd and the container. It
reports on the container's status and performs low-level tasks
such as keeping the container's STDIN and STDOUT streams
open.

Shims also make it possible to replace runc with other low-level
runtimes.

How it's implemented on Linux

On a Linux system, Docker implements the components we've
discussed as the following separate binaries:

- `/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-based Docker host by
running a `ps` command. Some of the processes will only be

present when the system has running containers, and you can't see them if you're using Docker Desktop on a Mac because the Docker Engine is running inside a VM.

Do we still need the daemon

At the time of writing, Docker has stripped most of the functionality out of the daemon. However, it still serves the Docker API.

Chapter summary

The Docker Engine comprises the server-side components of Docker and implements most of the code to build, share, and run containers. It implements the OCI standards and is a modular app comprising many small, specialized components.

The *Docker daemon* component implements the Docker API, but most other functionality has been stripped out and implemented as standalone composable tools such as containerd and runc.

containerd performs image management tasks and oversees container lifecycle management, such as starting, stopping, and deleting containers. Docker, Inc. originally wrote it and then contributed to the CNCF. It's classed as a high-level runtime and

used by many other projects, including Kubernetes, Firecracker, and Fargate.

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 is based on code from libcontainer, you can run it as a standalone CLI tool to create containers, and it's used almost everywhere that containerd is used.

Shims make it possible to use containerd with other low-level runtimes.

6: Working with Images

This chapter is a dive deep into Docker images. You'll learn what images are, how to work with them, and how they work under the hood. You'll learn how to build your own in *Chapter 8: Containerizing an application.*

I've arranged the chapter as follows:

- Docker images – The TLDR
- Intro to images
- Pulling images
- Image registries
- Image naming and tagging
- Images and layers
- Pulling images by digest
- Multi-architecture images
- Vulnerability scanning with Docker Scout
- Deleting images

Docker images – The TLDR

Before getting started, all of the following terms mean the same thing, and we'll use them interchangeably: *Image, Docker image, container image*, and *OCI image*.

An image is a read-only package containing everything you need to run an application. This means they include application code, dependencies, a minimal set of OS constructs, and metadata. You can start multiple containers from a single image.

If you're familiar with VMware, images are a bit like VM templates — a VM template is like a stopped VM, whereas an image is like a stopped container. If you're a developer, images are similar to *classes* — you can create one or more objects from a class, whereas you can create one or more containers from an image.

The easiest way to get an image is to *pull* one from a *registry*. [Docker Hub](#) is the most common registry, and *pulling* an image downloads it to your local machine where Docker can use it to start one or more containers. Other registries exist, and Docker works with them all.

Docker creates images by stacking independent *layers* and representing them as a single unified object. One layer might have the OS components, another layer might have application dependencies, and another layer might have the application. Docker stacks these layers and makes them look like a unified system.

Images are usually small. For example, the official NGINX image is around 80MB, and the official Redis image is around 40MB. However, Windows images can be huge.

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

Intro to images

We've already said that images are like stopped containers. You can even stop a container and create a new image from it. With this in mind, images are *build-time* constructs, whereas containers are *run-time* constructs. Figure 6.1 shows the *build* and *run* nature of each and that you can start multiple containers from a single image.

Image
(Build)
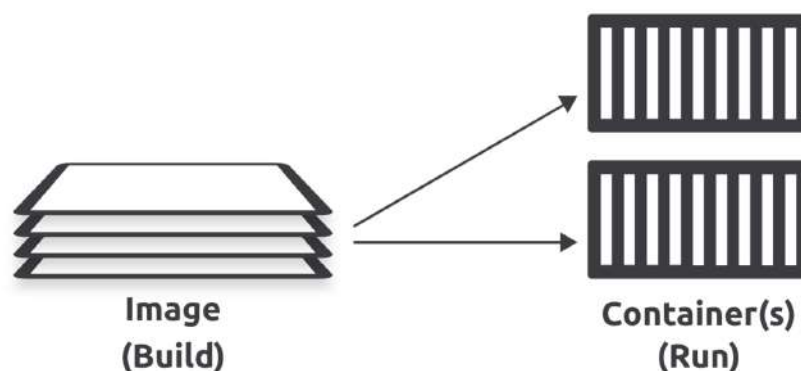
Container(s)
(Run)

Figure 6.1

The `docker run` command is the most common way to start a container from an image. Once the container is running, the image and the container are bound, and you cannot delete the image until you stop and delete the container. If multiple containers use the same image, you can only delete the image after you've deleted all the containers using it.

Containers are designed to run a single application or microservice. As such, they should only contain application code and dependencies. You should not include non-essentials such as build tools or troubleshooting tools.

For example, the official Alpine Linux image is currently about 3MB. This is because it doesn't ship with six different shells, three different package managers, and a bunch of tools you "might" need once every ten years. In fact, it's increasingly common for images to ship without a shell or a package manager — if the application doesn't need it at run-time, the image doesn't include it. We call these *slim images.*

Another thing that keeps images small is the lack of an OS kernel. This is because containers use the kernel of the host they're running on. The only OS-related components in most images are filesystem objects, and you'll sometimes hear people say images contain *just enough OS.*

Unfortunately, Windows images can be huge. For example, some Windows-based images can be gigabytes in size and take a long time to push and pull.

Pulling images

A clean Docker installation has an empty *local repository*.

Local repository is jargon for an area on your local machine where Docker stores images for more convenient access. We sometimes call it the *image cache*, and on Linux it's usually located in `/var/lib/docker/<storage-driver>`. However, if you're using Docker Desktop, it will be inside the Docker VM.

Run the following command to inspect the contents of your local repository. This example has three images relating to three Docker Desktop extensions I'm running. Yours will be different and may be empty.

```
$ docker images
REPOSITORY                              TAG       IMAGE ID        CREATED
docker/disk-usage-extension             0.2.9     f4c95478a537    26 hours a
docker/logs-explorer-extension          0.2.6     417dd9a8f96d    26 hours a
portainer/portainer-docker-extension    2.19.4    908d04d20e86    2 months a
```

The process of getting images is called *pulling.*

Run the following commands to pull the **redis** image and verify it exists in your local repository.

> **Note:** *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.*

```
$ docker pull redis
Using default tag: latest          <<---- Assume the 'latest' tag
latest: Pulling from library/redis <<---- Assume you want to pull from
08df40659127: Download complete    <<---- Pulling layer
4f4fb700ef54: Already exists       <<---- Pulling layer (local copy mu
4fe7fa4aab04: Download complete    <<---- Pulling layer
57dea0f129a5: Download complete    <<---- Pulling layer
f546e941f15b: Download complete    <<---- Pulling layer
f7f7da262cdb: Download complete    <<---- Pulling layer
f45ab649e450: Download complete    <<---- Pulling layer
983f900bbc88: Download complete    <<---- Pulling layer
Digest: sha256:76d5908f5e19fcdd73daf956a38826f790336ee4707d9028f32b24ad9a
Status: Downloaded newer image for redis:latest
docker.io/library/redis:latest     <<---- docker.io = Docker Hub

$ docker images
REPOSITORY    TAG      IMAGE ID       CREATED       SIZE
redis         latest   11c3e418c296   2 weeks ago   223MB
<Snip>
```

The image now exists in your local repository. However, I've annotated a few interesting lines from the **docker pull** output.

We'll cover them in more detail later in the chapter but they're worth a quick mention now.

Docker is opinionated and made two assumptions when pulling the image:

1. It assumed you wanted to pull the image tagged as `latest`
2. It assumed you wanted to pull the image from Docker Hub

You can override both, but Docker will use these as defaults if you don't override them.

The Redis image in the example has eight layers. However, Docker only pulled seven layers because it already had a local copy of one of them. This is because my system runs the *Portainer* Docker Desktop extension, which is based on an image that shares a common layer with the Redis image. You'll learn about this very soon, but images can share layers, and Docker is clever enough only to pull the layers it doesn't already have.

Image registries

We store images in centralized places called *registries*. The job of a registry is to securely store images and make them easy to access from different environments.

Figure 6.2 shows the central nature of registries in the build > share > run pipeline.
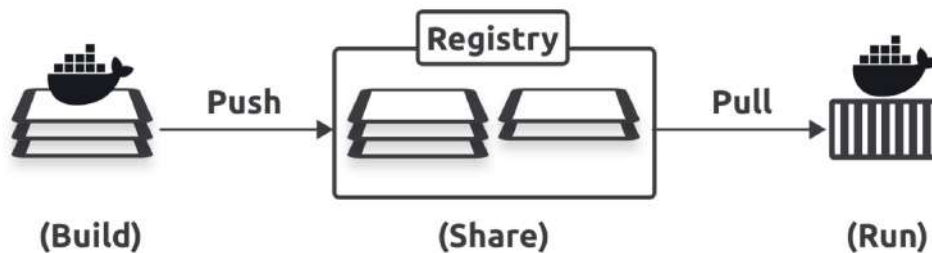
Figure 6.2

Most modern registries implement the OCI distribution-spec, and we sometimes call them *OCI registries*. Most registries also implement the Docker Registry v2 API, meaning you can use the Docker CLI and other API tools to query them and work with them in standard ways. Some offer advanced features such as image scanning and integration with build pipelines.

The most common registry is Docker Hub, but others exist, including 3rd-party internet-based registries and secure on-premises registries. However, as previously mentioned, Docker is opinionated and will default to Docker Hub unless you tell it the name of a different registry. We'll use Docker Hub for the rest of the book, but the principles apply to other registries.

Image registries contain one or more *image repositories*, and image repositories contain one or more images. Figure 6.3 shows an image registry with three repositories, each with one or more images.
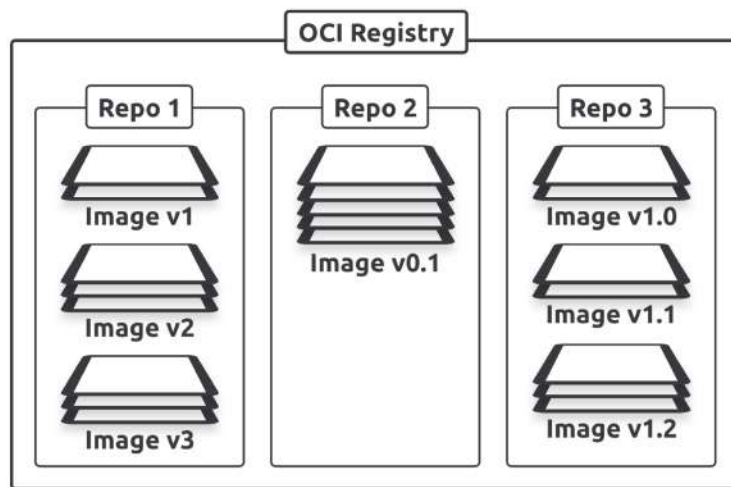
Figure 6.3 - Registry architecture

Official repositories

Docker Hub has the concept of *official repositories* that are home to images vetted and curated by Docker and the application vendor. This means they *should* contain up-to-date high-quality code that is secure, well-documented, and follows good practices.

Most of the popular applications and operating systems have *official repositories* on Docker Hub, and they're easy to identify 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/

Figure 6.4 shows the official Alpine and NGINX repositories on Docker Hub. Both have the green *Docker Official Image* badge and have over a billion pulls each. Also, notice how both are available for a wide range of CPU architectures.
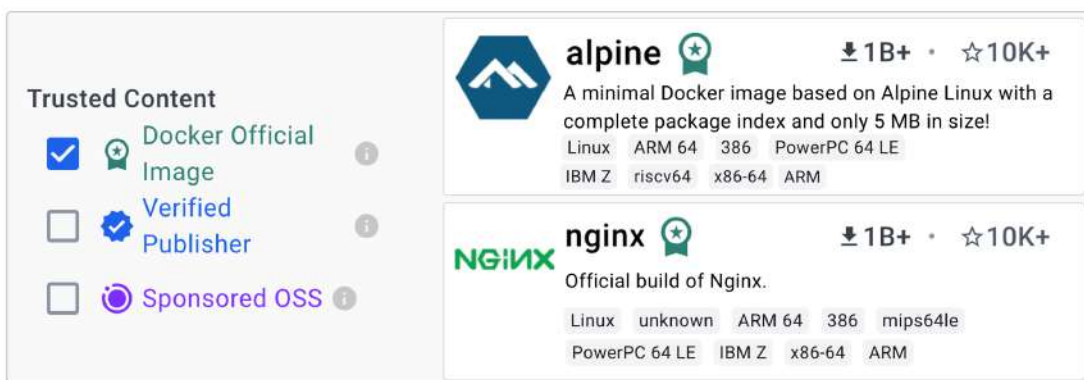
Figure 6.4 - Official repos on Docker Hub

Unofficial repositories

The next list shows two of my personal repositories in the "wild west" of *unofficial repositories* that you should be very careful when using.

- nigelpoulton/gsd —
  https://hub.docker.com/r/nigelpoulton/gsd-book/
- nigelpoulton/k8sbook —
  https://hub.docker.com/r/nigelpoulton/k8sbook/

Notice how they exist below the **nigelpoulton** second-level namespace. This is one of several indications they are not official repositories.

While there are lots of great images in unofficial repositories, you should always start with the assumption that anything from an unofficial repository is **unsafe**. This is based on the good practice of never trusting software from the internet. In fact, you should also exercise caution when downloading and using Docker Official Images.

Image naming and tagging

Most of the time, you'll work with images based on their names, and you can learn a lot about an image from its name. Figure

6.5 shows a fully qualified image name, including the registry name, user/organization name, repository name, and tag. Docker automatically populates the registry and tag values if you don't specify them.
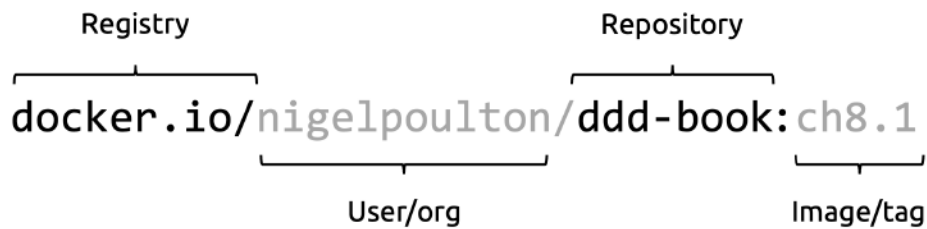
Figure 6.5 - Fully qualified image name

Addressing images from official repositories is easy. All you need to supply is the repository name and image name separated by a colon. Sometimes we call the image name the *tag*. The format for a `docker pull` command pulling an image from an official repository is:

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

The example from earlier pulled the Redis image with the following command. It pulled the image tagged as `latest` from the top-level `redis` repository.

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

The following examples show how to pull a few different official images.

```
$ docker pull redis:8.0-M02
//Pulls the image tagged as '8.0-M02' from the official 'redis' repository

$ docker pull busybox:glibc
//Pulls the image tagged as 'glibc' from the official 'busybox' repository

$ docker pull alpine
//Pulls the image tagged as 'latest' from the official 'alpine' repository
```

A couple of things are worth noting.

- As previously mentioned, if you **don't** specify an image tag after the repository name, Docker assumes you want the image tagged as `latest`. The command will fail if the repository has no image tagged as `latest`.
- Images tagged as `latest` are not guaranteed to be the most up-to-date in the repository.

Pulling images from *unofficial repositories* is almost the same as pulling from official repositories — you just need to add a Docker Hub username or organization name before the repository 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 ID is `nigelpoulton` .

```
$ docker pull nigelpoulton/tu-demo:v2
```

To pull an image from a different registry, you just add the registry's DNS name before the repository name. For example, the following command pulls the `latest` image from Brandon Mitchell's `regclient/regsync` repo on GitHub Container Registry (ghcr.io).

```
$ docker pull ghcr.io/regclient/regsync:latest
latest: Pulling from regclient/regsync
f140ae7f526a: Download complete
c1cb552669af: Download complete
Digest: sha256:88b3d4dc3d7bf2d8ea6f641bea2be15142a9222db66d4b6f2043fc5cc19
Status: Downloaded newer image for ghcr.io/regclient/regsync:latest
ghcr.io/regclient/regsync:latest
```

Notice how the pull looks the same as it did with Docker Hub. This is because GHCR supports the OCI registry-spec and implements the Docker Registry v2 API.

Images with multiple tags

You can give a single image as many tags as you want.

At first glance, the following output might look like it's listing three images. However, on closer inspection it's just two — the `b4210d0aa52f` image is tagged as `latest` and `v1`.

```
$ docker images
REPOSITORY              TAG       IMAGE ID        CREATED          SIZE
nigelpoulton/tu-demo    latest    b4210d0aa52f    2 days ago       115MB
nigelpoulton/tu-demo    v1        b4210d0aa52f    2 days ago       115MB
nigelpoulton/tu-demo    v2        6ba12825d092    12 minutes ago   115MB
```

This is a great example of the `latest` tag not relating to the newest image in the repo. In this example, the `latest` tag refers to the same image as the `v1` tag, which is actually older than the `v2` image.

Images and layers

As already mentioned, images are a collection of loosely connected read-only layers where each layer comprises one or more files.

Figure 6.6 shows an image with four layers. Docker takes care of stacking them and representing them as a single unified image.
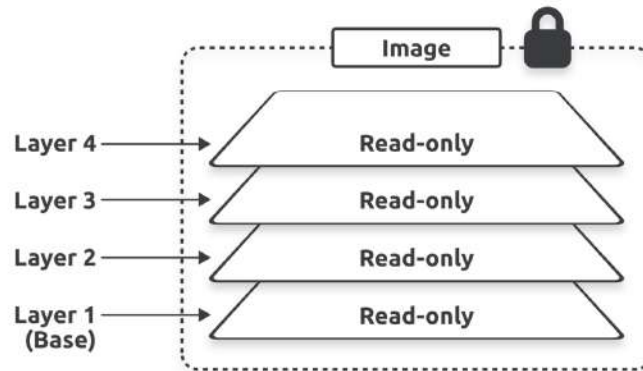
Figure 6.6 - Image and stacked layers

You're about to look at all of the following ways to inspect layer information:

- Pull operations
- The `docker inspect` command
- The `docker history` command

Run the following command to pull the `node:latest` image and observe it pulling the individual layers. Some newer versions may have more or less layers, but the principle is the same.

```
$ docker pull node: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 node:latest
docker.io/node:latest
```

Each line ending with *Pull complete* represents a layer that Docker pulled. This image has five layers and is shown in Figure 6.7 with layer IDs.
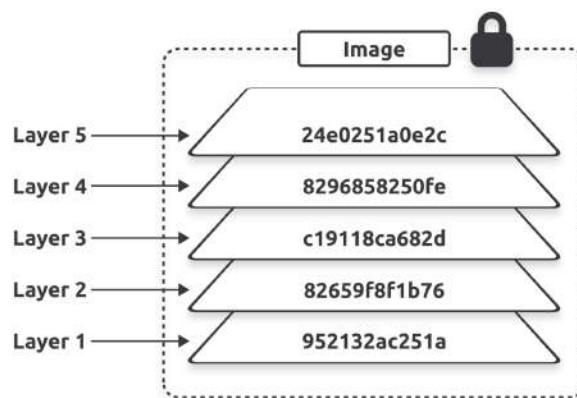
Figure 6.7 - Image layers and IDs

Another way to see image layers is to inspect the image with the `docker inspect` command. The following example inspects the same `node:latest` image pulled in the previous step.

```
$ docker inspect node:latest
[
    {
        "Id": "sha256:bd3d4369ae.......fa2645f5699037d7d8c6b415a10",
        "RepoTags": [
```

```
            "node: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 the five layers. However, it shows their SHA256 hashes, which are different from the short IDs shown in the `docker pull` output.

The `docker inspect` command is great for getting detailed image information.

You can also use the `docker history` command to inspect an image and see its layer data. However, this command 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`) only add metadata and don't create layers.

Base layers

All Docker images start with a *base layer*, and every time you add new content, Docker adds a new layer.

Consider the following oversimplified example of building a simple Python application. Your corporate policy mandates all applications be built on top of the official Ubuntu 24:04 image. This means the official Ubuntu 24:04 image will be the base layer for this app. Installing your company's approved version of Python will add a second layer, and your application source code will add a third. The final image will have three layers, as shown in Figure 6.8. Remember, this is an oversimplified example for demonstration purposes.
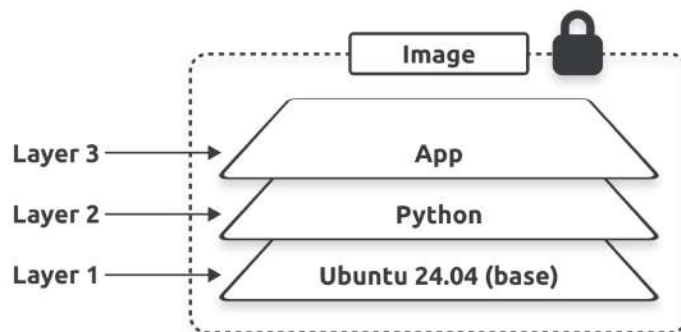
Figure 6.8

It's important to understand that an *image* is the combination of all layers stacked in the order they were built. Figure 6.9 shows an image with two layers. Each layer has three files, meaning the image has six files.

It also shows that the *layers* are stored as independent objects, and the *image* is just metadata identifying the required layers and explaining how to stack them.
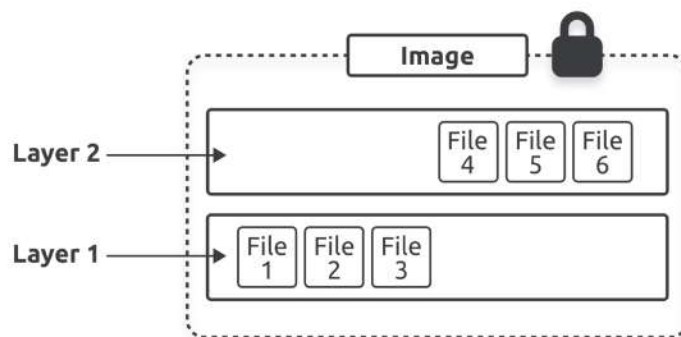
Figure 6.9

In the slightly more complex example of the three-layer image in Figure 6.10, the overall image only presents six 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 means you update files and make other changes to images by adding new layers containing the changes.
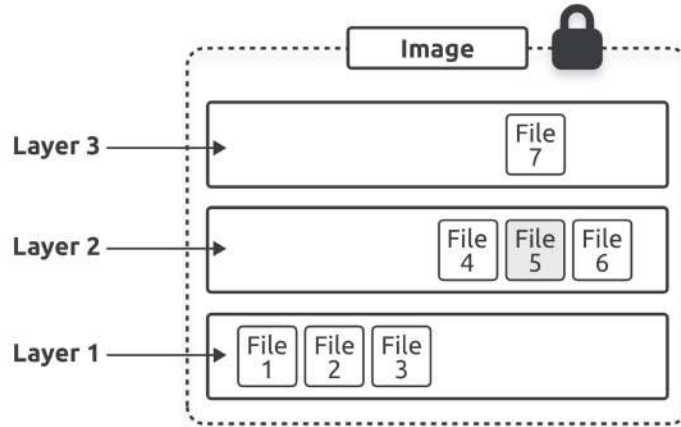
Figure 6.10 - Stacking layers

Under the hood, Docker uses *storage drivers* to stack layers and present them as a unified filesystem and image. Almost all Docker setups use the **overlay2** driver, but **zfs**, **btrfs**, and **vfs** are alternative options. However, whichever storage driver you use, the developer and user experience are always the same.

Figure 6.11 shows how the three-layer image from Figure 6.10 will appear on the system — all three layers stacked and merged into a single unified view.
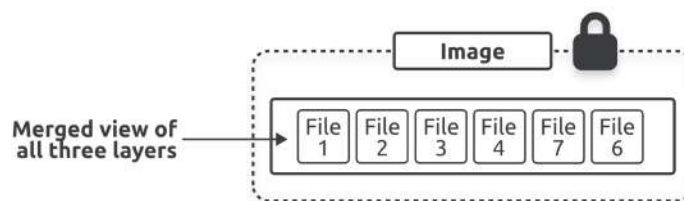
Figure 6.11 - Unified view of multi-layer image

Sharing image layers

As previously mentioned, images can share layers, leading to efficiencies in space and performance.

One of the earlier `docker pull` commands generated an *Already exists* message for one of the layers it pulled. This occurred because one of my Docker Desktop extensions had already pulled an image that used the exact same layer. As a result, Docker skipped that layer as it already had a local copy.

Here's the code from earlier, and Figure 6.12 shows two images sharing the same layer.

```
$ docker pull redis:latest
latest: Pulling from library/redis
25d3892798f8: Download complete
e5d458cf0bea: Download complete
4f4fb700ef54: Already exists          <<---- This line
<Snip>
```
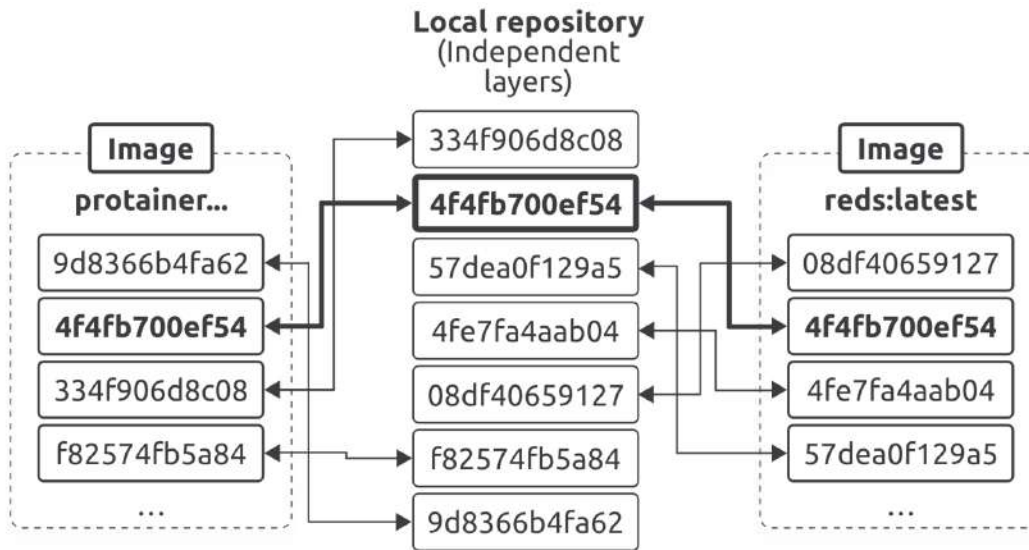
Figure 6.12 - Two images sharing a layer

Layers are also shared on the registry side. This means you can store lots of similar images in a registry, and the registry will save space by never storing more than a single copy of any layer.

Pulling images by digest

So far, you've seen how to pull and work with images using names (tags). While this is the most common method, it has a problem — tags are arbitrary and mutable. This means it's possible to tag an image incorrectly or give a new image the same tag as an older one. An extremely common example is the `latest` tag. For example, pulling the `alpine:latest` tag a year ago will not pull the same image as pulling the same tag today.

Consider a quick example outlining one potential implication of trusting mutable tags. Imagine you have an image called `golftrack:1.5` and you get a warning that it has a critical vulnerability. You build a new image containing the fix and push the new image to the same repository with the **same tag**.

Take a moment to consider what just happened and the implications.

You have an image called `golftrack:1.5` that's being used by lots of containers in your production environment, and it has a critical bug. You create a new version containing the fix. So far, so good, but then you make the mistake. You push the new image to the same 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 — both images have the same tag!

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

Docker uses a *content addressable storage* model where every image gets a cryptographic *content hash* that we usually call the *digest*. As these are hashes of an image's contents, it's impossible

for two different images to have the same digest. It's also impossible to change an image without creating a new digest. Fortunately, Docker lets you work with image digests instead of just names.

If you've already pulled an image by name, you can see its digest by running a `docker images` command with the `--digests` flag as shown.

```
$ docker images --digests alpine
REPOSITORY    TAG        DIGEST                       IMAGE ID       CREATED
alpine        latest     sha256:c5b1261d...8e1ad6b    c5b1261d6d3e   2 weeks
```

If you want to find an image's digest **before pulling** it, you can use the `docker buildx imagetools` command. The following example retrieves the image digest for the `nigelpoulton/k8sbook/latest` image on Docker Hub.

```
$ docker buildx imagetools inspect nigelpoulton/k8sbook:latest
Name:      docker.io/nigelpoulton/k8sbook:latest
MediaType: application/vnd.docker.distribution.manifest.list.v2+json
Digest:    sha256:13dd59a0c74e9a147800039b1ff4d61201375c008b96a29c5bd17244
<Snip>
```

You can now use the digest to pull the image. I've trimmed the command and the output for readability.

```
$ docker pull nigelpoulton/k8sbook@sha256:13dd59a0...bce2e14b
docker.io/nigelpoulton/k8sbook@sha256:13dd59a0...bce2e14b: Pulling from ni
59f1664fb787: Download complete
a052f1888b3e: Download complete
94a9f4dfa0e5: Download complete
bb7e600677fa: Download complete
edfb0c26f1fb: Download complete
5b1423465504: Download complete
2f232a362cd9: Download complete
Digest: sha256:13dd59a0...bce2e14b
Status: Downloaded newer image for nigelpoulton/k8sbook@sha256:13dd59a0...
docker.io/nigelpoulton/k8sbook:latest@sha256:13dd59a0...bce2e14b
```

It's also possible to directly query the registry API for image data, including digest. The following `curl` command queries Docker Hub for the digest of the same image.

```
$ curl "https://hub.docker.com/v2/repositories/nigelpoulton/k8sbook/tags/
  |jq '.results[].digest'

"sha256:13dd59a0c74e9a147800039b1ff4d61201375c008b96a29c5bd17244bce2e14b"
```

Image hashes and layer hashes

You already know that images are just a loose collection of independent layers. This means an *image* is just a manifest file with some metadata and a list of layers. The actual application and all its dependencies live in the *layers* that are fully independent and have no concept of being part of an image.

With this in mind, images and layers have their own digests as follows:

- Images digests are a crypto hash of the image's **manifest file**
- Layer digests are a crypto hash of the layer's contents

This means all changes to layers or image manifests result in new hashes, giving us an easy and reliable way to know if changes have been made.

Content hashes vs distribution hashes

Docker compares hashes before and after every push and pull to ensure no tampering occurs while data is crossing the network. However, it also compresses images during push and pull operations to save network bandwidth and storage space on the registry. As a result of this compression, the before and after hashes won't match.

To get around this, each layer gets two hashes:

- Content hash (uncompressed)
- Distribution hash (compressed)

Every time Docker pushes or pulls a layer from a registry, it includes the layer's *distribution hash* and uses this to verify no tampering occurred. This is one reason why the hashes in

different CLI and registry outputs don't always match —
sometimes you're looking at the content hash, and other times
you're looking at the distribution hash.

Multi-architecture images

One of the best things about Docker is its simplicity. However, as
technologies grow, they inevitably 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, s390x and more. Suddenly,
there were multiple versions of the same image for all the
different architectures, and developers and users had to put in
significant extra work to get the right version. This broke the
smooth Docker experience.

Multi-architecture images to the rescue!

Fortunately, Docker and the registry API adapted and became
clever enough to hide images for multiple architectures behind
a single tag. This means you can do a `docker pull alpine` on
any architecture and get the correct version of the image. For
example, if you're on an AMD64 machine, you'll get the AMD64
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 an image tag. Each supported architecture then has its own *manifest* that lists the layers used to build it.

Run the following command to see the different architectures supported behind the `alpine:latest` tag.

```
$ docker buildx imagetools inspect alpine
Name:      docker.io/library/alpine:latest
MediaType: application/vnd.docker.distribution.manifest.list.v2+json
Digest:    sha256:c5b1261d6d3e43071626931fc004f70149baeba2c8ec672bd4f27761

Manifests:
  Name:      docker.io/library/alpine:latest@sha256:6457d53f...628977d0
  MediaType: application/vnd.docker.distribution.manifest.v2+json
  Platform:  linux/amd64

  Name:      docker.io/library/alpine:latest@sha256:b229a851...d144c1d8
  MediaType: application/vnd.docker.distribution.manifest.v2+json
  Platform:  linux/arm/v6

  Name:      docker.io/library/alpine:latest@sha256:ec299a7b...33b4c6fe
  MediaType: application/vnd.docker.distribution.manifest.v2+json
  Platform:  linux/arm/v7
```

```
    Name:       docker.io/library/alpine:latest@sha256:a0264d60...93467a46
    MediaType: application/vnd.docker.distribution.manifest.v2+json
    Platform:   linux/arm64/v8

    Name:       docker.io/library/alpine:latest@sha256:15c46ced...ab073171
    MediaType: application/vnd.docker.distribution.manifest.v2+json
    Platform:   linux/386

    Name:       docker.io/library/alpine:latest@sha256:b12b826d...ba52a3a2
    MediaType: application/vnd.docker.distribution.manifest.v2+json
    Platform:   linux/ppc64le
```

Your output may include additional annotations, but if you look closely, you'll see a single *manifest list* pointing to six *manifests*.

`MediaType: application/vnd.docker.distribution.manifest.list.v2+json` is the manifest list.

Each `MediaType: application/vnd.docker.distribution.manifest.v2+json` line refers to a manifest for each specific architecture.

Figure 6.13 shows how manifest lists and manifests are related. On the left, you can see a manifest list with entries for the different architectures supported by the image. The arrows show that each entry in the manifest list points to a manifest

defining the image config and the list of layers making up the image for that architecture.
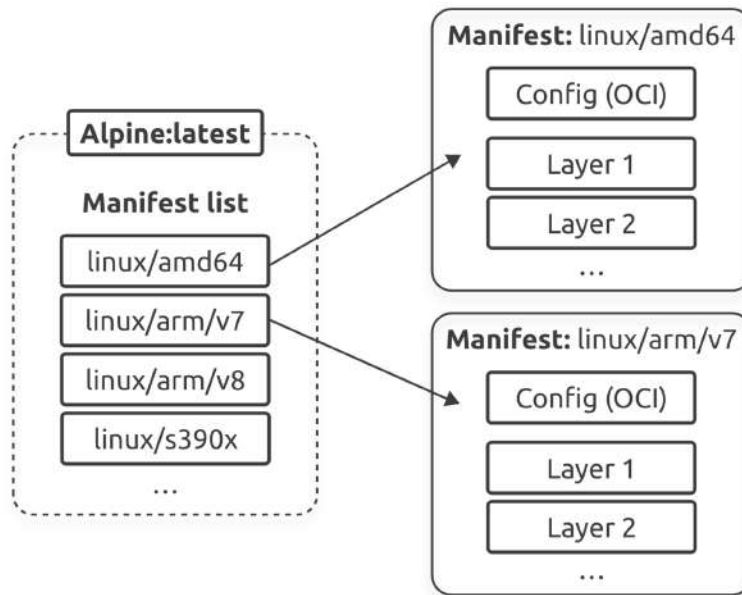
Figure 6.13 - Manifest lists and manifests

Let's step through a quick example.

Assume you're using Docker Desktop on an M4 Mac where Docker runs inside a **linux/arm** VM. You ask Docker to pull an image, and Docker makes the relevant calls to the Registry API to request the appropriate manifest list. Assuming it exists, Docker then parses it for a **linux/arm** entry. If **linux/arm** entry exists, Docker retrieves its manifest, parses it for the crypto IDs

of its layers, pulls each layer, and assembles them into the image.

Let's see it in action.

The following examples are from Docker Desktop on an ARM-based Mac and Docker Desktop on an AMD-based Windows machine running in *Windows containers mode.* Both start a new container based the official `golang` image and execute the `go version` command. The outputs show the version of Go and the host's platform and CPU architecture. Notice how both commands are exactly the same, and Docker takes care of pulling the correct image.

Both images are large and may take a while to download. You do not need to complete these commands yourself.

Linux on arm64 example:

```
$ docker run --rm golang go version
<Snip>
go version go1.23.4 linux/arm64
```

Windows on x64 example:

```
> docker run --rm golang go version
<Snip>
```

```
go version go1.23.4 windows/amd64
```

You've already seen how to use the `docker buildx imagetools` command to see the manifest list and manifests for an image. You can get similar information from the `docker manifest` command. The following example inspects the manifest list for the official **golang** image on Docker Hub. You can see it has images for Linux and Windows on a variety of CPU architectures. You can run the same command without the `grep` filter to see the full JSON manifest list. Windows users should replace the `grep` command with `Select-String architecture,os`

```
$ 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": "10.0.20348.2227"
```

```
            "architecture": "amd64",
            "os": "windows",
            "os.version": "10.0.17763.5329"
```

Pulling the right image for your system is one thing, but what about **building** images for all these different architectures?

The `docker buildx` command makes it easy to create multi-architecture images. For example, you can use Docker Desktop on `linux/arm` to build images for `linux/amd` and possibly other architectures. We'll perform builds like these in future chapters, but `docker buildx` offers two ways to create multi-architecture images:

- Emulation
- Build Cloud

Emulation mode performs builds for different architectures on your local machine by running the build inside a QEMU virtual machine emulating the target architecture. It works most of the time but is slow and doesn't have a shared cache.

Build Cloud is a service from Docker, Inc. that performs builds in the cloud on native hardware without requiring emulation. It's very fast, lets you share a common build cache with teammates, and is seamlessly integrated into Docker Desktop and any

version of the Docker Engine using a version of buildx supporting the cloud driver. It also integrates with GitHub actions and other CI solutions. At the time of writing, Docker Build Cloud is a subscription service you have to pay for.

We'll use both in future chapters, but I ran the following command to build AMD and ARM versions of the `nigelpoulton/tu-demo` image using Docker Build Cloud.

```
$ docker buildx build \
   --builder=cloud-nigelpoulton-ddd-cloud \
   --platform=linux/amd64,linux/arm64 \
   -t nigelpoulton/tu-demo:latest --push .
```

Vulnerability scanning with Docker Scout

Lots of tools and plugins exist that scan images for known vulnerabilities.

We'll look at Docker Scout, as it's built into almost every level of Docker, including the CLI, Docker Desktop, Docker Hub, and the `scout.docker.com` portal. It's a very slick service, but it requires a paid subscription. Other similar products and services exist, but most require paid subscriptions.

Recent versions of Docker Desktop have the Scout CLI plugin pre-installed and ready to go. If you're running a different

version of Docker, you may be able to install the CLI plugin from the [GitHub repo](#).

You can use the **docker scout quickview** command to get a quick vulnerability overview of an image. The following command analyses the **nigelpoulton/tu-demo:latest** image. If a local copy doesn't exist, it pulls it from Docker Hub and performs the analysis locally.

```
$ docker scout quickview nigelpoulton/tu-demo:latest

   ✓ SBOM of image already cached, 66 packages indexed

  Target              |  nigelpoulton/tu-demo:latest  |    0C     1H     1M
    digest            |  b4210d0aa52f                 |
  Base image          |  python:3-alpine              |    0C     1H     1M
  Updated base image  |  python:3.11-alpine           |    0C     1H     1M
                      |                               |
```

The output shows zero critical vulnerabilities (0C), one high (1H), one medium (1M), and zero low (0L).

You can use the **docker scout cves** command to get more detailed information, including remediation advice.

```
$ docker scout cves nigelpoulton/tu-demo:latest

   ✓ SBOM of image already cached, 66 packages indexed
```

```
      X Detected 1 vulnerable package with 2 vulnerabilities
## Overview
                       |           Analyzed Image
───────────────────────|──────────────────────────────
  Target                |   nigelpoulton/tu-demo:latest
    digest              |    b4210d0aa52f
    platform            |  linux/arm64/v8
    vulnerabilities     |    0C     1H     1M     0L
    size                |  26 MB
    packages            |  66

## Packages and Vulnerabilities
    0C     1H     1M     0L   expat 2.5.0-r2
pkg:apk/alpine/expat@2.5.0-r2?os_name=alpine&os_version=3.19

    X  HIGH CVE-2023-52425
      https://scout.docker.com/v/CVE-2023-52425
      Affected range : <2.6.0-r0
      Fixed version  : 2.6.0-r0
<Snip>
```

I've snipped the output so it only shows the critical and high
vulnerabilities, but several things are clear:

- It has detected one vulnerable package containing two
  vulnerabilities
- The affected package is called **expat** and the vulnerable
  version we're running is `2.5.0-r2`
- It lists the vulnerability as `CVE-2023-52425`
- It includes a link to a Scout report containing more info
- It suggests we update to version `2.6.0-r0` which contains
  the fix

Figure 6.14 shows how this looks in Docker Desktop, and you get similar integrations and views in Docker Hub.

Figure 6.14 - Docker Scout integration with Docker Desktop

The `scout.docker.com` portal provides an overview dashboard, allows you to configure policies, and lets you set up integrations with Docker Hub and other registries to remotely scan and monitor multiple repositories.

Deleting Images

You can delete images using the **docker rmi** command. **rmi** is short for *remove image.*

Deleting images removes them from your local repository and they'll no longer show up in your `docker images` commands. The operation also deletes all directories on your local filesystem containing layer data. However, Docker won't delete layers shared by multiple images until you delete all images that reference them.

You can delete images by name, short ID, or SHA. You can also delete multiple images with the same command.

The following command deletes three images — one by name, one by short ID, and one by SHA. I've trimmed the output for easier reading.

```
$ docker rmi redis:latest af111729d35a sha256:c5b1261d...f8e1ad6b
Untagged: redis:latest
Deleted: sha256:76d5908f5e19fcdd73daf956a38826f790336ee4707d9028f32b24ad9a
Untagged: nigelpoulton/tu-demo:v2
Deleted: sha256:af111729d35a09fd24c25607ec045184bb8d76e37714dfc2d9e55d13b3
Untagged: alpine:latest
Deleted: sha256:c5b1261d6d3e43071626931fc004f70149baeba2c8ec672bd4f27761f8
```

Docker will prevent the delete operation if the image is being used by a container or referenced by more than one tag. However, you can force the operation with the `-f` flag, but you should do so with caution, as forcing Docker to delete an image

in use by a container will untag the image and leave it on the system as a *dangling image.*

A handy way to **delete all images** is to pass a list of all local image IDs to the `docker rmi` command. You should use this command with caution, and if you're following along on Windows, it will only work in a PowerShell terminal.

```
$ 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
<Snip>

$ docker pull ubuntu
<Snip>

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

See how the `docker images -q` returns a list of local image IDs. 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 some of the commands we've used.

Images – The commands

- `docker pull` is the command to download images from
  remote registries. It defaults to Docker Hub but works with
  other registries. The following command will pull the image
  tagged as `latest` from the `alpine` repository on Docker
  Hub: `docker pull alpine:latest`.
- `docker images` lists all the images in your Docker host's
  local repository (image cache). You can add the `--digests`
  flag to see the SHA256 hashes.
- `docker inspect` gives you a wealth of image-related
  metadata in a nicely formatted view.
- `docker manifest inspect` lets you inspect the manifest list
  of images stored in registries. The following command will

show the manifest list for the `regctl` image on GitHub Container Registry (GHCR): `docker manifest inspect ghcr.io/regclient/regctl`.

- `docker buildx` is a Docker CLI plugin that works with Docker's latest build engine features. You saw how to use the `imagetools` sub-command to query manifest-related data from images.
- `docker scout` is a Docker CLI plugin that integrates with the Docker Scout backend to perform image vulnerability scanning. It scans images, provides reports on vulnerabilities, and even suggests remediation actions.
- `docker rmi` is the command to delete images. It deletes all layer data stored in the local filesystem, and you cannot delete images that are in use by containers.

Chapter summary

This chapter taught you the important theory and fundamentals of images.

You learned that images contain everything needed to run an application as a container. This includes just enough OS, source code, dependencies, and metadata.

You can start one or more containers from a single image.

Under the hood, Docker constructs images by stacking one or more read-only layers and presenting them as a unified object. Every image has a manifest that lists the layers that make up the image and how to stack them.

You learned that image names are also called tags, they're mutable, and they don't always pull the same image. For example, pulling the `alpine:latest` tag today will not pull the same image as it will a year from now. Fortunately, every image gets an immutable digest that you can use to guarantee you always pull the intended image.

Docker Hub has the notion of curated *official images* that should be safer to use than unofficial images. However, you should always exercise caution when downloading software from the internet, even official images from Docker Hub.

Images can share layers for efficiency, and Docker makes it easy to build and pull images for lots of different CPU architectures, such as ARM and AMD.

Docker Scout scans images for known vulnerabilities and provides remediation advice. It requires a Docker subscription and is integrated into the `docker` CLI, Docker Hub, and Docker Desktop.

In the next chapter, we'll take a similar tour of containers — the run-time sibling of images.

7: Working with containers

Docker implements the Open Container Initiative (OCI) specifications. This means some of the things you'll learn in this chapter will apply to other container runtimes and platforms that implement the OCI specifications.

I've divided the chapter into the following sections:

- Container – The TLDR
- Containers vs VMs
- Images and containers
- Check Docker is running
- Starting a container
- How containers start apps
- Connecting to a running container
- Inspecting container processes
- The `docker inspect` command
- Writing data to a container
- Stopping, restarting, and deleting a container
- Killing a container's main process
- Debugging slim images and containers with Docker Debug
- Self-healing containers with restart policies
- The commands

Containers – The TLDR

Containers are run-time instances of images, and you can start one or more containers from a single image.

Figure 7.1 shows multiple containers started from a single image. The shared image is read-only, but you can write to the containers.
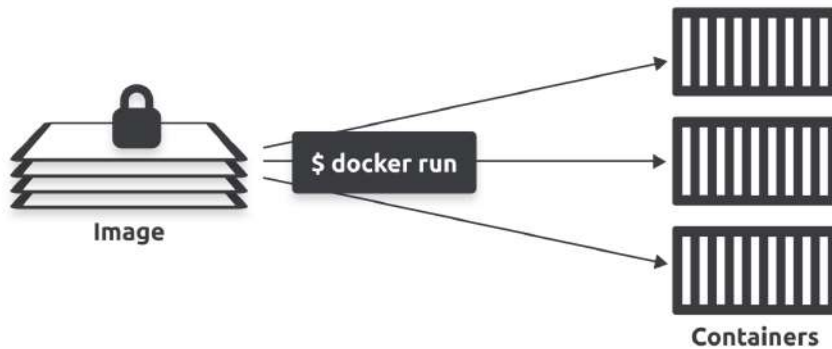
Figure 7.1

You can start, stop, restart, and delete containers just like you can with VMs. However, containers are smaller, faster, and more portable than VMs. They're also designed to be *stateless* and *ephemeral*, whereas VMs are designed to be long-running and can be migrated with their state and data.

Containers are also designed to be *immutable.* This means you shouldn't change them after you've deployed them — if a container fails, you replace it with a new one instead of connecting to it and making a live fix.

Containers should only run a single process and we use them to build microservices apps. For example, an application with four features, such as a web server, auth, catalog, and store, will have four containers — one running the web server, one running the auth service, one running the catalog, and another running the store.

Containers vs VMs

Containers and VMs are both virtualization technologies for running applications. They both work on your laptop, bare metal servers, in the cloud, and more. However, the ways they *virtualize* are very different:

- VMs virtualize hardware
- Containers virtualize operating systems

In the VM model, you power on a server and a hypervisor boots. When the hypervisor boots, it claims all hardware resources such as CPU, RAM, storage, and network adapters. To deploy an app, you ask the hypervisor to create a virtual

machine. It does this by carving up the hardware resources into virtual versions, such as virtual CPUs and Virtual RAM, and packaging them into a VM that looks exactly like a physical server. Once you have the VM, you install an OS and then an app.

In the container model, you power on the same server and an OS boots and claims all hardware resources. You then install a container runtime such as Docker. To deploy an app, you ask Docker to create a container. It does this by carving up OS resources such as process trees and filesystems into virtual versions and then packaging them as a container that looks exactly like a regular OS. You then tell Docker to run the app inside the container.

Figure 7.2 shows the two models side by side and attempts to demonstrate the more efficient nature of containers with the same server running 3x more containers than VMs.
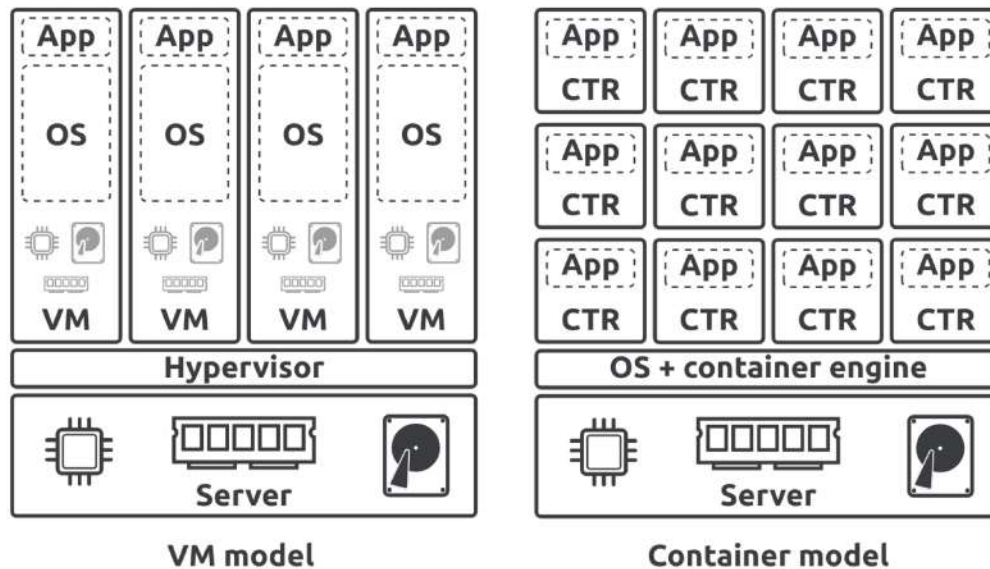
Figure 7.2

In summary, hypervisors perform *hardware virtualization* where they divide hardware resources into virtual versions and package them as VMs. Container runtimes perform *OS virtualization* where they divide OS resources into virtual versions and package them as *containers.* VMs look and feel exactly like physical servers. Containers look and feel exactly like regular operating systems.

The VM tax

One of the biggest problems with the virtual machine model is that you need to install an OS on every VM — every OS

consumes CPU, RAM, and storage and takes a relatively long time to boot.

Containers get around all of this by sharing a single OS on the host they're running on. This gives containers all of the following benefits over VMs:

- Containers are smaller and more portable
- You can run more containers on your infrastructure
- Containers start faster
- Containers reduce the number of operating systems you need to manage (patch, update, etc.)
- Containers present a smaller attack surface

Let's briefly expand on each point.

Containers are smaller than VMs because they only contain application code and a minimal set of OS-related constructs such as essential filesystem objects. Because of this, they're typically only a few megabytes in size. On the other hand, every VM needs a full OS, meaning they're usually hundreds or thousands of megabytes.

Because containers don't contain their own OS, you can run a lot more containers than VMs. For example, deploying 100 applications as VMs will require 100 operating systems, each

consuming CPU, memory, and storage, and each needing to be patched and managed. However, deploying the same 100 applications as containers requires no additional operating systems. This drastically reduces your OS management overhead and allows you to allocate more system resources to applications instead of operating systems.

Containers also start faster than VMs because they use the host's OS which is already booted. On the other hand, VMs need to go through a full OS bootstrapping process before starting the app.

One of the early concerns about containers centered around the shared kernel model where all containers on the same host share the host's kernel. While this offers performance and portability benefits, it's less secure than the VM model where every VM has its own dedicated kernel. For example, a rogue container that exploits a vulnerability in the host's kernel might be able to impact every other container on the same host. Fortunately, this is much less of a concern now that container platforms have matured and ship with class-leading tools that *can* make them more secure than non-container platforms. For example, most container engines and platforms implement *sensible defaults* for security-related technologies such as *SELinux, AppArmor, seccomp, capabilities,* and more. You can

even configure these to make containers more secure than VMs. Other technologies, such as image vulnerability scanning, give you more control over the security of your software than you ever had before.

At the time of writing, containers are the go-to solution for the vast majority of new applications.

Pre-reqs

You'll need a working Docker environment to follow along with the examples, and I recommend Docker Desktop. Other Docker setups should work, but you may have to manually install the Docker Debug plugin if you want to follow along with those examples.

Images and Containers

As previously mentioned, you can start multiple containers from a single image. The image is read-only in this relationship, but each container is read-write. As shown in Figure 7.3, Docker accomplishes this by creating a thin read-write layer for each container and placing it on top of the shared image.
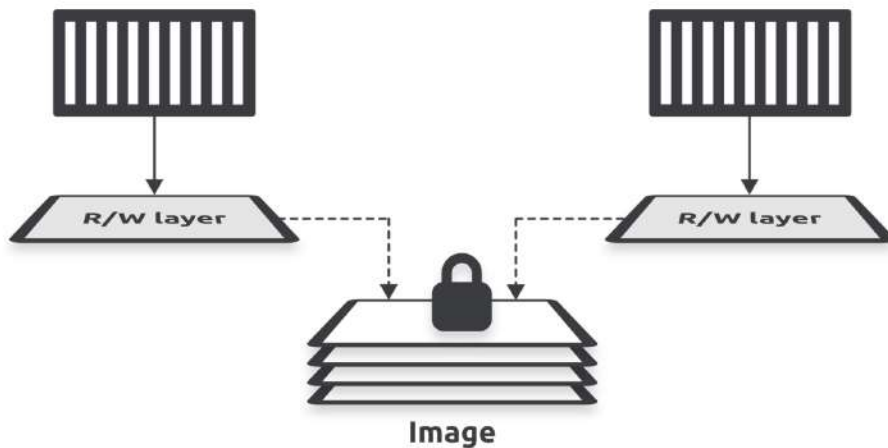
Figure 7.3 - Container R/W layers

In this example, each container has its own thin R/W layer but shares the same image. The containers can see and access the files and apps in the image *through* their own R/W layer, and if they make any changes, these get written to their R/W layer. When you stop a container, Docker keeps the R/W layer and restores it when you restart the container. However, when you delete a container, Docker deletes its R/W layer. This way, each container can make and keep its own changes without requiring write access to the shared image.

Check Docker is running

Run a `docker version` to check Docker is running. It's a good command because it checks the CLI and engine components.

```
$ docker version
Client:
 Version:          27.4.0
 API version:      1.47
 OS/Arch:          darwin/arm64
<Snip>
Server: Docker Desktop 4.30.0 (149282)
 Engine:
  Version:          27.4.0
  API version:      1.47 (minimum version 1.24)
  OS/Arch:          linux/arm64
<Snip>
```

As long as you get a response from the `Client` and `Server`, you're good to go and can skip to the next section.

If you get an error code in the `Server` section, this usually means your Docker daemon (server) isn't running or your user account doesn't have permission to access it. If you're running on Linux, you'll need to ensure your user account is a member of the local `docker` Unix group. If it isn't, you can add it by running `usermod -aG docker <username>` and restarting your shell. Alternatively, you can prefix all `docker` commands with `sudo`.

Your account needs to be a member of the **docker** group so it can access the API, which is exposed on a privileged local Unix socket at `/var/run/docker.sock`. It's also possible to expose the API over the network.

If your user account is already a member of the local `docker` group and you still get an error from the daemon, there's a good chance the Docker daemon isn't running. Run one of the following commands to check the status of the daemon.

Linux systems not using Systemd.

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

Linux systems using Systemd.

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

If the daemon isn't running, start it with the appropriate command for your system.

Starting a container

The `docker run` command is the simplest and most common way to start a new container.

Run the following command to start a new container called **webserver**.

```
$ docker run -d --name webserver -p 5005:8080 nigelpoulton/ddd-book:web0.1
Unable to find image 'nigelpoulton/ddd-book:web0.1' locally
web0.1: Pulling from nigelpoulton/ddd-book
4f4fb700ef54: Already exists
cf2a607f33f7: Download complete
0a1f0c111e9a: Download complete
c1af4b5db242: Download complete
Digest: sha256:3f5b281b914b1e39df8a1fbc189270a5672ff9e98bfac03193b42d1c02d
Status: Downloaded newer image for nigelpoulton/ddd-book:web0.1
b5594b3b8b3fdce544d2ca048e4340d176bce9f5dc430812a20f1852c395e96b
```

Let's take a closer look at the command and the output.

`docker run` tells Docker to run a new container

The `-d` flag tells Docker to run it in the background as a *daemon process* and detached from your local terminal

The `name` flag tells Docker to name this container **webserver**.

The `-p 5005:8080` flag maps port `5005` on your local system to port `8080` inside the container. This works because the container's web server is listening on port `8080`.

The `nigelpoulton/ddd-book:web0.1` argument tells Docker which image to use to start the container.

When you hit **Return**, the Docker client converted the command into an API request and posted it to the Docker API

exposed by the Docker daemon. The Docker daemon accepted the command and searched its local image repository for a copy of the `nigelpoulton/ddd-book:web0.1` image. It didn't find one, so it searched Docker Hub. In the example, it found one on Docker Hub and pulled a local copy.

Once it had a local copy of the image, the daemon made a request to containerd asking for a new container. containerd then instructed runc to create the container and start the app. It also performed the port mapping.

Run the following commands to verify Docker pulled the image and started the **webserver** container.

```
$ docker images
REPOSITORY               TAG      IMAGE ID       CREATED          SIZE
nigelpoulton/ddd-book    web0.1   3f5b281b914b   12 minutes ago   159MB

$ docker ps
CONTAINER ID   IMAGE          COMMAND           STATUS      PORTS
b5594b3b8b3f   nigelpoulton... "node ./app.js"   Up 2 mins   0.0.0.0:80
```

You can also test the app by connecting a browser to port `5005` on your Docker host. If you're using Docker Desktop, point your browser to `localhost:5005`. If you're not running Docker

Desktop, you may need to substitute `localhost` with the name or IP of the host Docker is running on.

Figure 7.4 - Web app running on container

Congratulations. Docker pulled a local copy of the image and started a container running the app shown in the image.

How containers start apps

In the previous section, you created a container running a web app. But how did the container know to start a web app?

There are three ways you can tell Docker how to start an app in a container:

1. An `Entrypoint` instruction in the image
2. A `Cmd` instruction in the image
3. A CLI argument

You'll learn more about these in the next chapter, but the *Entrypoint* and *Cmd* instructions are optional image metadata where you can store the command you want Docker to run to start the default app. Then, whenever you start a container from the image, Docker checks the Entrypoint or Cmd instruction and executes the stored command.

`Entrypoint` instructions cannot be overridden on the CLI, and anything you pass in via the CLI will be appended to the Entrypoint instruction as an argument.

`Cmd` instructions are overridden by CLI arguments.

Run the following command to see if the `nigelpoulton/ddd-book:web0.1` image has an Entrypoint instruction. The command searches the image metadata and returns any lines containing the word *"Entrypoint"* as well as the three lines immediately following it. Windows users will need to replace the `grep` command with `Select-String -Pattern 'Entrypoint' -Context 0,3`.

```
$ docker inspect nigelpoulton/ddd-book:web0.1 | grep Entrypoint -A 3
<Snip>
"Entrypoint": [
    "node",
    "./app.js"
],
```

This image has an Entrypoint instruction that translates into the
following command — `node ./app.js`. If you're not familiar
with Node.js, it's a simple command telling the Node.js runtime
to execute the code in the `app.js` file.

If an image doesn't have an Entrypoint instruction, you can
search for the presence of a *Cmd* instruction.

If an image doesn't have either, you'll need to pass an argument
on the CLI.

The format of the `docker run` command is:

```
docker run <arguments> <image> <command>
```

As mentioned, the `<command>` is optional; you don't need it if
the image has a Cmd or Entrypoint instruction. If you specify a
`<command>`, it will override a Cmd instruction but will be
appended to an Entrypoint instruction.

The following command starts a new background container based on the **Alpine** image and tells it to run the `sleep 60` command, causing it to run for 60 seconds and then exit. The `--rm` flag cleans up the exited container so you don't have to delete it manually.

```
$ docker run --rm -d alpine sleep 60
```

If you run a `docker ps` command before the 60-second sleep timer expires, you'll see the container in the output. If you run it after 60 seconds, the container will be gone. The `--rm` argument automatically cleans up the exited container.

Most production images will specify an Entrypoint or Cmd instruction.

Connecting to a running container

You can use the `docker exec` command to execute commands in running containers, and it has two modes:

- Interactive
- Remote execution

Interactive exec sessions connect your terminal to a shell process in the container and behave like remote SSH sessions.

Remote execution mode lets you send commands to a running container and prints the output to your local terminal.

Run the following command to start an interactive exec session by creating a new shell process (`sh`) inside the **webserver** container that is already running. The `-it` flag makes it an *interactive exec session*, and the `sh` argument starts a new `sh` process inside the container. `sh` is a minimal shell program installed in the container.

```
$ docker exec -it webserver sh
/src #
```

Notice how your shell prompt changed. This proves your terminal is connected to the shell process inside the container.

Try executing a few common Linux commands. Some will work, and some won't. This is because container images are usually 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.

The examples list the contents of your current directory and try to edit the `app.js` file with the `vim` editor.

```
/src # ls -l
total 100
-rw-r--r--     1 root      root              324 Feb 20 12:35 Dockerfile
-rw-r--r--     1 root      root              377 Feb 20 12:35 README.md
-rw-r--r--     1 root      root              341 Feb 20 12:35 app.js
drwxr-xr-x   183 root      root             4096 Feb 20 12:41 node_modules
-rw-r--r--     1 root      root            74342 Feb 20 12:41 package-lock.json
-rw-r--r--     1 root      root              404 Feb 20 12:38 package.json
drwxr-xr-x     2 root      root             4096 Feb 20 12:35 views
<Snip>

/src # vim app.js
sh: vim: not found
```

The `vim` command fails because it isn't installed in the container.

Inspecting container processes

Most containers only run a single process. This is the container's main app process and is always PID 1.

Run a `ps` command to see the processes running in your container. You'll need to be connected to the exec session from the previous section for these commands to work.

```
/src # ps
PID    USER       TIME   COMMAND
    1 root         0:00 node ./app.js
   13 root         0:00 sh
   22 root         0:00 ps
```

The output shows three processes:

- PID 1 is the main application process running the Node.js web app
- PID 13 is the shell process your interactive exec session is connected to
- PID 22 is the `ps` command you just ran

The `ps` process terminated as soon as it displayed the output, and the `sh` process will terminate when you exit the *exec session*. This means the only long-running process is PID 1 running the Node app.

If you kill the container's main process (PID 1), you'll also kill the container. This is because containers only run while their main process is executing — when that process is no longer running, there's no reason for the container to run. We'll demonstrate this later.

Type `exit` to quit the exec session and return to your local terminal.

Run another `docker exec` command without specifying the `-it` flags. This will remotely execute the command without creating an interactive session. The format of the command is

`docker exec <container> <command>`, and it will only work if the container has the command you're trying to execute.

```
$ docker exec webserver ps
PID   USER      TIME  COMMAND
   1 root       0:00 node ./app.js
  42 root       0:00 ps
```

This time, only two processes are running because you terminated the `sh` process when you typed `exit` to quit the previous interactive exec session.

The `docker inspect` command

You'll love the `docker inspect` command as it's a treasure trove of detailed information about images and containers.

The following command retrieves full details of the running **webserver** container, and I've snipped the output to highlight a few interesting things. However, I recommend running the command on your system and studying the output.

```
$ docker inspect webserver
<Snip>
"State": {
    "Status": "running"
},
"Name": "/webserver",
    "PortBindings": {
```

```
        "8080/tcp": [
            {
                "HostIp": "",
                "HostPort": "5005"
            }
        ]
    },
    "RestartPolicy": {
        "Name": "no",
        "MaximumRetryCount": 0
    "Image": "nigelpoulton/ddd-book:web0.1",
    "WorkingDir": "/src",
    "Entrypoint": [
        "node",
        "./app.js"
    ],
        }
<Snip>
```

The snipped output shows the container is *running*, is called *webserver*, is binding port `8080` in the container to `5005` on the host, has no restart policy, and is based on the `nigelpoulton/ddd-book:web0.1` image. The `Entrypoint` block lists the command that automatically runs every time the container starts.

We'll cover this in more detail later, but this container inherited its Entrypoint instruction from the image you started it from. You can verify this by running the following `docker inspect` command against the image. I've snipped the output to highlight the relevant section.

```
$ docker inspect nigelpoulton/ddd-book:web0.1
<Snip>
"Config": {
    "WorkingDir": "/src",
    "Entrypoint": [
        "node",
        "./app.js"
    ],
<Snip>
```

I recommend you take time to investigate the output of `docker inspect` commands. You'll learn a lot.

Writing data to a container

In this section, you'll exec onto the `webserver` container and edit the web server configuration to display a new message on the home page. In the next section, you'll stop and restart the container and verify your changes aren't lost.

> **WARNING:** *This section is for demonstration purposes only. In the real world, you shouldn't change live containers like this. Any time you need to change a live container, you should create and test a new container with the required changes and then replace the existing container with the new one.*

Open a new interactive exec session to the `webserver` container with the following command.

```
$ docker exec -it webserver sh
/src #
```

The container runs a simple Node.js web app that uses the `views/home.pug` file to build the app's home page.

Run the following command to open the `home.pug` file in the `vi` editor. Windows users can use Notepad or another editor.

```
/src # vi views/home.pug
```

If you know how to use `vi`, you can go ahead and change the text on line 8 after the `h1` tag to anything you like and save your changes.

Carefully follow these steps if you're not familiar with `vi`:

1. Press the `i` key to put `vi` into *insert mode*
2. Use the arrow keys to navigate to line 8
3. Use your `delete` key to delete the text **after** the `h1` tag on line 8
4. Type a new message of your choice

5. Press the `escape` key to exit *insert mode* and return to *command mode*

6. type `:wq` and press **enter** save your changes and exit (`:wq` is short for write and quit)

Once you've saved your changes, refresh your browser to see the updates.

Type `exit` to quit the exec session and return to your local terminal.

Congratulations, you've updated the web server config.

Stopping, restarting, and deleting a container

In this section, you'll execute the typical container lifecycle events and see how they impact the changes you've made to the container.

The following commands will only work if you've quit the interactive exec session.

Check your container is still running.

```
$ docker ps
CONTAINER ID   IMAGE            COMMAND           STATUS       PORTS
b5594b3b8b3f   nigelpoulton...  "node ./app.js"   Up 51 mins   0.0.0.0:86
```

Stop it with the `docker stop` command. It will take up to 10 seconds to gracefully stop.

```
$ docker stop webserver
webserver
```

Run another `docker ps` command.

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

The container no longer shows in the list of running containers. However, you can see it if you run the same command with the `-a` flag to show **all** containers, including stopped ones.

```
$ docker ps -a
CONTAINER ID    IMAGE          COMMAND          STATUS
b5594b3b8b3f    nigelpou...    "node ./app.js"    Exited (137) About a minute
```

As you can see in the output, it still exists but is in the `Exited` state. Restart it with the following command.

```
$ docker restart webserver
webserver
```

If you run another `docker ps`, you'll see it in the `Up` state.

Refresh your browser to see if Docker has saved your changes to the home page or reverted to the original.

Docker has saved your changes!

You can also run the following command to return the contents of the file directly from the container's filesystem.

```
$ docker exec webserver cat views/home.pug

html
  head
    title='Docker FTW'
    link(rel='stylesheet', href='https://stackpath.bootstrapcdn.com/....
  body
    div.container
      div.jumbotron
        h1 Everybody loves containers!    <<---- I changed this line
<Snip>
```

So far, you've seen that starting and stopping containers doesn't lose changes. You also saw that restarting them is very fast.

Run the following command to delete the container. The `-f` flag forces the operation and doesn't allow the app the usual 10-second grace period to flush buffers and gracefully quit. Be

careful forcing operations like this, as Docker doesn't ask you to confirm.

```
$ docker rm webserver -f
webserver
```

Run a `docker ps -a` to see if there's any sign of the container.

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

All signs of the container are gone and you cannot restart it. You can start a new instance by executing another `docker run` command and specifying the same image, but it won't have the changes you made.

> **WARNING:** *As previously mentioned, changing live containers like this is an anti-pattern and you shouldn't do it. We only showed it here to demonstrate how containers work and how changes to the container's filesystem (made to the container's own thin R/W layer) persist across restarts. An anti-pattern is something that works but isn't a good practice as it can have unintended consequences.*

Killing a container's main process

Earlier in the chapter, we learned that containers are designed to run a single process, and we said that killing this process also kills the container.

Let's test if that's true.

Run the following command to start a new interactive container called **ddd-ctr** based on the Ubuntu image and tell it to run a Bash shell as its main process.

```
$ docker run --name ddd-ctr -it ubuntu:24.04 bash
Unable to find image 'ubuntu:24.04' locally
24.04: Pulling from library/ubuntu
51ae9e2de052: Download complete
Digest: sha256:ff0b5139e774bb0dee9ca8b572b4d69eaec2795deb8dc47c8c829becd6
Status: Downloaded newer image for ubuntu:24.04
root@d3c892ad0eb3:/#
```

The command pulls the Ubuntu image and attaches your terminal to the container's Bash shell process.

Run a `ps` command to list all running processes.

```
root@d3c892ad0eb3:/# ps
  PID TTY          TIME CMD
    1 pts/0    00:00:00 bash
    9 pts/0    00:00:00 ps
```

PID 1 is the container's main process and is the Bash shell you told the container to run. The other one is the `ps` command and has already exited. This means the Bash process is the only process running in the container.

If you type `exit`, you'll terminate the Bash process **and** kill the container. This is because containers only run while their main process executes.

Test this by typing `exit` to return to your local terminal and then running a `docker ps -a` command to see if the container terminated.

```
root@d3c892ad0eb3:/# exit

$ docker ps -a
CONTAINER ID    IMAGE          COMMAND    STATUS                  NAMES
d3c892ad0eb3    ubuntu:24.04   "bash"     Exited (0) 3 secs ago   ddd-ctr
```

As expected, the container is in the *exited* state and not running. However, you can run the following two commands to restart it and attach your shell to its main process.

```
$ docker restart ddd-ctr
ddd-ctr
```

```
$ docker attach ddd-ctr
root@d3c892ad0eb3:/#
```

Your terminal is once again attached to the Bash shell in the container.

You can type `Ctrl PQ` to exit a container without killing the process you're attached to.

Type `Ctrl PQ` to exit the container and run another `docker ps` command to verify the container is still running this time.

```
root@d3c892ad0eb3:/# <Ctrl PQ>
read escape sequence

$ docker ps
CONTAINER ID    IMAGE          COMMAND     STATUS          NAMES
d3c892ad0eb3    ubuntu:24.04   "bash"      Up 27 seconds   ddd-ctr
```

The container is still up.

Now that you know how to exit containers without killing them, let's switch focus and see how to use *Docker Debug* to debug slim containers and images.

Debugging slim images and containers with Docker Debug

At the time of writing, Docker Debug is only included as part of Docker Desktop and requires a Pro, Team, or Business subscription.

It's a widely accepted good practice to deploy *slim images* that only contain app code and dependencies. This means no shell or debugging tools and is a big part of making images and containers small and secure. However, it also makes it difficult to debug them when things go wrong.

This is where **Docker Debug** comes to the rescue by allowing you to get shell access to images and containers that don't include a shell and seamlessly inject powerful debugging tools into them.

At a high level, Docker Debug works by attaching a shell to a container and mounting a toolbox loaded with debugging tools. This *toolbox* is mounted as a directory called `/nix` and is available during your debugging session but is never visible to the container. As soon as you exit the Docker Debug session, the `/nix` directory is removed. If you're debugging a running container, any changes you make are immediately visible to the container and persist across container restarts. For example, updating an `index.html` during a Docker Debug session will immediately update the running web app, and the changes will

persist if the container is stopped and restarted. If you're debugging an image or stopped container, the Docker Debug session creates a debug sandbox and adds it to the image as a R/W layer to make it feel like a running container. However, changes you make while debugging an image or stopped container are **not** persisted and are lost as soon as you quit the debug session.

If you've been following along, you'll have a running container called **ddd-ctr**. If you don't, you can start one by running `docker run --name ddd-ctr -it ubuntu:24.04 bash`.

Run the following commands to attach to the container and see if it has any debugging tools. The following `docker attach` command is similar to the `docker exec` commands you learned earlier but automatically connects to a container's main process. You don't need to run the `docker attach` command if you're already connected to the container.

```
$ docker attach ddd-ctr
root@d3c892ad0eb3:/#

root@d3c892ad0eb3:/# ping nigelpoulton.com
bash: ping: command not found

root@d3c892ad0eb3:/# nslookup nigelpoulton.com
bash: nslookup: command not found
```

```
root@d3c892ad0eb3:/# vim
bash: vim: command not found
```

The commands all failed because none of the tools are installed in this container. This would make debugging this container difficult without Docker Debug.

Type `Ctrl PQ` to gracefully disconnect from the container without killing the Bash process.

In the following steps, you'll use Docker Debug to get a shell session to the container and run commands that aren't installed in the container. You can even use Docker Debug to get shell access to containers and images that don't include a shell.

You need to log in to Docker to use Docker Debug, and it only works if you have a Pro, Team, or Business license.

```
$ docker login
Authenticating with existing credentials...
Login Succeeded
```

Run the following command to check if you have the Docker Debug CLI plugin. All modern versions of Docker Desktop include this by default. Other Docker installations may not have it, but you may be able to install it manually.

```
$ docker info
Client:
 Version:    26.1.1
 Context:    desktop-linux
 Debug Mode: false
 Plugins:
  debug: Get a shell into any image or container. (Docker Inc.)
    Version:  0.0.29
    Path:     /Users/nigelpoulton/.docker/cli-plugins/docker-debug
<Snip>
```

Once you're logged in and have the plugin installed, you're
ready to continue.

The format of the command is `docker debug <image>|`
`<container>` . We'll open a Docker Debug session to the
running container called **ddd-ctr**.

```
$ docker debug ddd-ctr
```



```
This is an attach shell, i.e.:
- Any changes to the container filesystem are visible to the container dir
- The /nix directory is invisible to the actual container.
                                                      Version: 0.0.37
```

```
root@d3c892ad0eb3 / [ddd-ctr]
docker >
```

You've successfully connected to the running container and got a new shell prompt (`docker >`). You also got some helpful info displaying the short ID and name of the container you're debugging, as well as a reminder that any changes you make will be visible to the container.

Try running the `ping, nslookup`, and `vim` commands that failed in the previous section. If you get stuck in the `vim` session, just type `:q` and press `Enter`.

```
docker > ping nigelpoulton.com
PING nigelpoulton.com (192.124.249.126) 56(84) bytes of data.
64 bytes from cloudproxy10126.sucuri.net (192.124.249.126): icmp_seq=1 ttl
64 bytes from cloudproxy10126.sucuri.net (192.124.249.126): icmp_seq=2 ttl
^C

docker > nslookup nigelpoulton.com
zsh: command not found: nslookup

docker > vim
~                    VIM  - Vi IMproved
~                     version 9.0.1441
~              by Bram Moolenaar et al.
~       Vim is open source and freely distributable
<Snip>
:q
```

The **ping** and **vim** commands worked, but the **nslookup** still failed. This is because the default Docker Debug *toolbox* includes **ping** and **vim** but doesn't include **nslookup**. Don't worry, though. You can use Docker Debug's built-in **install** command to add any package listed on `search.nixos.org`.

Run the following command to install the **bind** package (which includes the **nslookup** tool), and then run the **nslookup** command again.

```
docker > install bind
Tip: You can install any package available at: https://search.nixos.org/pa
installing 'bind-9.18.19'
<Snip>

docker > nslookup nigelpoulton.com
Server:    192.168.65.7
Address:   192.168.65.7#53

Non-authoritative answer:
Name:      nigelpoulton.com
Address:   192.124.249.126
```

The command worked, and **nslookup** is now installed in your *toolbox* and will be available in future Docker Debug sessions.

Congratulations, you've used Docker Debug to attach to a running container and run troubleshooting commands that

aren't part of the container. You've also seen how to install additional tools to your Docker Debug toolbox. Remember, any changes you make to running containers are immediately visible to the container and persist after you close the session.

Type `exit` to terminate the debug session and return to your local shell.

Run the following command to create a new Docker Debug session that debugs the `nigelpoulton/ddd-book:web0.1` image. Docker will automatically pull the image from Docker Hub if you don't have a local copy.

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



```
Note: This is a sandbox shell. All changes will not affect the actual imag
                                                Version: 0.0.37 (BET
root@3f5b281b914b /src [nigelpoulton/ddd-book:web0.1]
docker >
```

Notice the different message this time. Debugging images creates a *sandbox shell* and changes won't affect the actual image. This reminds you that debugging images and stopped containers behaves differently from debugging running containers:

- Changes made while debugging a live container are persisted
- Changes made while debugging images or stopped containers are deleted when you quit the debug session

Run an `nslookup` command to prove the tool is saved to your toolbox and available for use without re-installing.

```
docker > nslookup craigalanson.com
Server:    192.168.65.7
Address:   192.168.65.7#53

Non-authoritative answer:
Name:      craigalanson.com
Address:   198.185.159.144
<Snip>
```

Docker Debug has a built-in `entrypoint` command that lets you print, lint, and test an image or container's *Entrypoint* or *Cmd* command. These are the commands Docker executes to start the container's app.

Run the following `entrypoint` command to reveal the default command this container will run when it starts.

```
docker > entrypoint --print
node ./app.js
```

The `entrypoint` command is clever enough to look for Entrypoint **and** Cmd instructions.

Type `exit` to quit the debug session.

In summary, Docker Debug is a fantastic tool for debugging *slim* images and containers. It gets you shell access to containers and images that don't include a shell, and you can run troubleshooting tools that aren't available in the container or image. Any changes you make to *running containers* take immediate effect and persist across stop and restart operations. However, changes made while debugging *images and stopped containers* are lost when you close the session. In all cases, the tools you install and use are never part of the container or image.

Self-healing containers with restart policies

Container *restart policies* are a simple form of self-healing that allows the local Docker Engine to automatically restart failed

containers.

You apply *restart policies* per container, and Docker supports the following four policies:

- **no** (default)
- **on-failure**
- **always**
- **unless-stopped**

The following table shows how each policy reacts to different scenarios. A **Y** indicates the policy will attempt a container restart, whereas an **N** indicates it won't.

Restart policy	Non-zero exit code	Zero exit code	docker stop command	Restart when Daemon restarts
no	N	N	N	N
on-failure	Y	N	N	Y
always	Y	Y	N	Y
unless-stopped	Y	Y	N	N

Non-zero exit codes indicate a failure occurred. Zero exit codes indicate the container exited normally without an error.

We'll demo some examples, but you should also do your own testing.

Let's demonstrate the **always** policy by starting a new interactive container with the **--restart always** flag and telling it to run a shell process. We'll then type **exit** to kill the shell process **and** the container to see what happens.

Run the following command to start an interactive container called **neversaydie** with the **always** restart policy.

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

Your terminal will automatically connect to the shell process inside the container.

Type **exit** to kill the shell process and return to your local terminal. This will cause the container to exit with a zero exit code, indicating a normal exit without any failures.

According to the previous table, the **always** restart policy should automatically restart the container.

Run a **docker ps** command to see if this happened.

```
$ docker ps
CONTAINER ID    IMAGE     COMMAND    CREATED         STATUS         NAMES
1933623830bb    alpine    "sh"       35 seconds ago  Up 2 seconds   neversa
```

The container is running as expected. However, you can see it was created 35 seconds ago but has only been running for 2 seconds. This is because you forced it to exit when you killed the shell process, and then Docker automatically restarted it. It's also important to know that Docker restarted the same container and didn't create a new one. In fact, if you run a `docker inspect` against it, you'll see the `RestartCount` has been incremented to 1. Remember to replace `grep` with `Select-String -Pattern 'RestartCount'` if you're on Windows using PowerShell.

```
$ docker inspect neversaydie | grep RestartCount
        "RestartCount": 1,
```

An interesting feature of the `--restart always` policy is that if you stop a container with `docker stop` and then restart the Docker daemon, Docker will restart the container when the daemon comes up. To be clear:

1. You start a new container with the `--restart always` policy
2. You manually stop it with the `docker stop` command
3. You restart Docker (or an event causes Docker to restart)
4. When Docker comes back up, it starts the *stopped container*

If you don't want this behavior, you should try the `unless-stopped` policy.

If you are working with Docker Compose or Docker Stacks, you can apply restart policies to *services* as follows. We'll cover these in more detail in later chapters.

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

Clean up

You can run `docker images` and `docker ps -a` commands to see the images you pulled and the containers you created as part of this chapter. Your output will be similar to this.

```
$ docker images
REPOSITORY               TAG        IMAGE ID        CREATED         SIZE
nigelpoulton/ddd-book    web0.1     3f5b281b914b    4 days ago      159MB
ubuntu                   24.04      ff0b5139e774    13 days ago     138MB
alpine                   latest     c5b1261d6d3e    4 weeks ago     11.8MB

$ docker ps -a
CONTAINER ID    IMAGE           COMMAND     CREATED         STATUS
ac165419214f    alpine          "sh"        33 secs ago     Up 24 seconds
5bd3741185fa    ubuntu:24.04    "bash"      3 mins ago      Exited (0) ~1min ago
```

You can delete individual containers with the `docker rm <container> -f` command and images with the `docker rmi` command, and you should always delete containers before images.

You can also delete **all containers** and **all images** with the following two commands. Be warned though, Docker will not prompt you for confirmation.

```
$ docker rm $(docker ps -aq) -f
ac165419214f
5bd3741185fa

$ docker rmi $(docker images -q)
Untagged: nigelpoulton/ddd-book:web0.1
Deleted: sha256:3f5b281b914b1e39df8a1fbc189270a5672ff9e98bfac03193b42d1c02
Untagged: ubuntu:24.04
Deleted: sha256:ff0b5139e774bb0dee9ca8b572b4d69eaec2795deb8dc47c8c829becd6
Untagged: alpine:latest
Deleted: sha256:c5b1261d6d3e43071626931fc004f70149baeba2c8ec672bd4f27761f8
```

Both commands work by passing a list of **all** container/image IDs to the delete command.

Containers – The commands

- `docker run` is the command to start new containers. You give it the name of an image and it starts a container from it.

This example starts an interactive container from the Ubuntu image and tells it to run the Bash shell: `docker run -it ubuntu bash`.

- `Ctrl-PQ` is how you detach from a container without killing the process you're attached to. You'll use it frequently to detach from running containers without killing them.
- `docker ps` lists all running containers, and you can add the `-a` flag to also see containers in the stopped `(Exited)` state.
- `docker exec` allows you to run commands inside containers. The following command will start a new Bash shell inside a running container and connect your terminal to it: `docker exec -it <container-name> bash`. This next command runs a `ps` command inside a running container without opening an interactive shell session: `docker exec <container-name> ps`. For these to work, the container must include the Bash shell.
- `docker stop` stops a running container and puts it in the `Exited (137)` state. It issues a *SIGTERM* to the container's PID 1 process and allows the container 10 seconds to gracefully quit. If the process hasn't cleaned up and stopped within 10 seconds, it sends a *SIGKILL* to force the container to terminate immediately.
- `docker restart` restarts a stopped container.

- `docker rm` deletes a stopped container. You can add the `-f` flag to delete the container without having to stop it first.
- `docker inspect` shows you detailed configuration and run-time information about a container.
- `docker debug` attaches a debug shell to a container or image and lets you run commands that aren't available inside the container or image. It requires a Pro, Team, or Business Docker subscription.

Chapter summary

In this chapter, you learned some of the major differences between VMs and containers, including that containers are smaller, faster, and more portable.

You learned how to start, stop, and restart containers with the `docker` CLI, and you saw that changes to a container's filesystem persist across restarts.

You learned that containers run a single process and terminate if this process is killed. You also saw the three ways of telling a container which app to run and how to start it — via Entrypoint or Cmd instructions in the image metadata or via the `docker run` CLI.

You learned about Docker Debug and how it allows you to get a shell to slim containers and run troubleshooting commands that don't exist in the container.

Finally, you learned how to attach restart policies to containers and how the different restart policies work.

8: Containerizing an app

Docker makes it easy to package applications as images and run them as containers. We call this process *containerization*, and this chapter will walk you through the entire process.

I've divided the chapter as follows:

- Containerizing an app – The TLDR
- Containerize a single-container app
- Moving to production with multi-stage-builds
- Buildx, BuildKit, drivers, and Build Cloud
- Multi-architecture builds
- A few good practices

Containerizing an app – The TLDR

Docker aims to make it easy to *build, share,* and *run* applications. We call this *containerization* and the process looks like this:

1. Write your applications and create the list of dependencies
2. Create a *Dockerfile* that tells Docker how to build and run the app
3. Build the app into an image
4. Push the image to a registry (optional)

5. Run a container from the image

You can see these five steps in Figure 8.1.
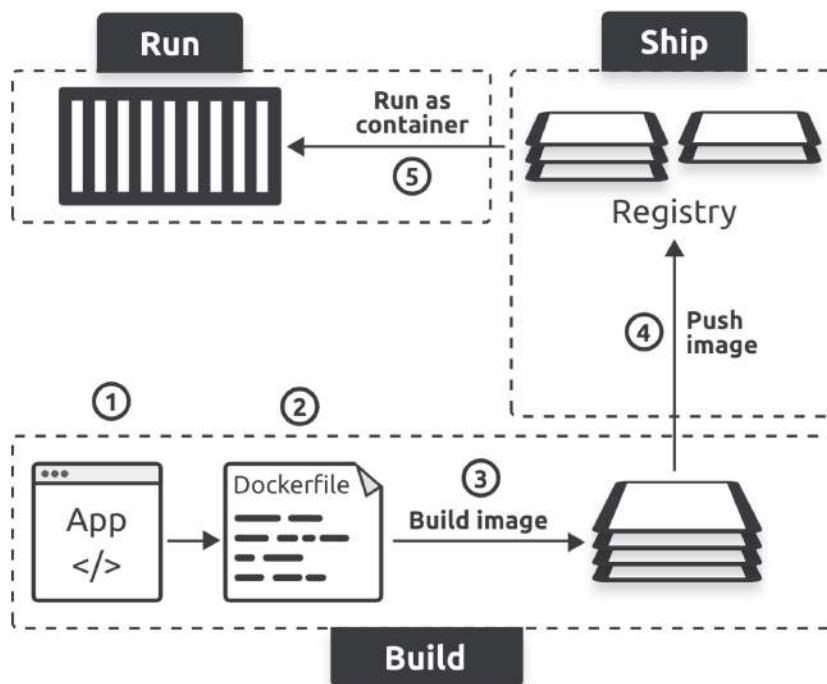
Figure 8.1 - Basic flow of containerizing an app

Containerize a single-container app

In this section, you'll complete the following steps to containerize a simple Node.js app:

- Get the application code from GitHub
- Create the Dockerfile

- Containerize the app
- Run the app
- Test the app
- Look a bit closer

I recommend you follow along with Docker Desktop. This is because we'll be using the new `docker init` command, which might not be installed on other versions of Docker. Don't worry if your Docker installation doesn't have `docker init`, we include instructions for you as well.

Get the application code

The application we'll use is a Node.js web app that serves a web page on port `8080`.

You'll need a copy of the book's GitHub repo containing the application code. If you don't already have it, run the following command to get it. You'll need `git` installed, and the command will create a new directory called `ddd-book`.

```
$ 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.
```

Change into the `ddd-book/node-app` directory and list its contents.

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

$ ls -l
total 98
-rw-r--r--@    1 nigelpoulton  staff    341 20 Feb 12:35 app.js
drwxr-xr-x  103 nigelpoulton  staff   3296 12 Mar 16:18 node_modules
-rw-r--r--     1 nigelpoulton  staff  39975 12 Mar 16:18 package-lock.json
-rw-r--r--@    1 nigelpoulton  staff    355  8 Mar 10:10 package.json
drwxr-xr-x    3 nigelpoulton  staff     96 20 Feb 12:35 views
```

This directory is your *build context* because it contains the application source code and the files listing dependencies.

For Docker to containerize it, it needs a *Dockerfile* with build instructions. Let's create it.

Create the Dockerfile

In the past, you had to create Dockerfiles manually. Fortunately, newer versions of Docker support the `docker init` command that reads your build context, analyzes your application, and automatically creates a Dockerfile implementing good practices.

Run the following command to create a Dockerfile for the app. If your Docker installation doesn't have the `docker init` plugin, you'll have to skip this step.

Feel free to accept a newer version of Node.js, but complete all other prompts as shown. You'll need to run it from the `node-app` directory.

```
$ docker init
Welcome to the Docker Init CLI!
<Snip>
? What application platform does your project use? Node
? What version of Node do you want to use? 23.3.0    <<---- Newer versions
? Which package manager do you want to use? npm
? What command do you want to use to start the app? node app.js
? What port does your server listen on? 8080

CREATED: .dockerignore
CREATED: Dockerfile
CREATED: compose.yaml
CREATED: README.Docker.md

✓ Your Docker files are ready!
```

The process created a new Dockerfile and placed it in your current directory. It looks like this.

```
1. ARG NODE_VERSION=20.8.0
2. FROM node:${NODE_VERSION}-alpine
3. ENV NODE_ENV production
```

```
4. WORKDIR /usr/src/app
5. RUN --mount=type=bind,source=package.json,target=package.json \
    --mount=type=bind,source=package-lock.json,target=package-lock.json \
    --mount=type=cache,target=/root/.npm \
    npm ci --omit=dev
6. USER node
7. COPY . .
8. EXPOSE 8080
9. CMD node app.js
```

Lines 1 and 2 tell Docker to pull the `node:23.3.0-alpine` image and use it as the base for the new image.

Line 3 tells Node to run in *production* mode. This is a Node.js optimization that increases performance while minimizing logging and other common development features.

Line 4 sets the working directory for the remaining steps. For example, the `RUN` and `COPY` instructions on lines 5 and 7 will run against the `WORKDIR` directory, as will the `node app.js` command on line 9.

Line 5 bind mounts the dependency files and installs them with the `npm ci --omit-dev` command.

Line 6 ensures Node.js runs the app as a non-root user.

Line 7 copies the application's source code from your build context (the first period) into the `WORKDIR` directory (the second period) inside the image.

Line 8 documents the application's network port.

Line 9 is the command Docker will execute whenever it starts a container from the image.

You now have everything Docker needs to build the application into a container image — source code, dependencies, and a Dockerfile.

Containerize the app

In this section, you'll build the application into a container image.

If your Docker installation doesn't have the `docker init` plugin and you didn't follow the previous step, you'll need to rename the `sample-Dockerfile` to `Dockerfile` before continuing.

Run the following command to build a new image called `ddd-book:ch8.node`. Be sure to include the trailing period (`.`) as this tells Docker to use your current working directory as the

build context. Remember, the *build context* is the directory where your app files live.

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

[+] Building 16.2s (12/12) FINISHED
 => [internal] load build definition from Dockerfile          0.0s
 => => transferring dockerfile: 1.21kB                        0.0s
 => => transferring context: 659B                             0.0s
 => [stage-0 1/4] FROM docker.io/library/node:20.8.0-alpine    3s   <<--
 => [stage-0 2/4] WORKDIR /usr/src/app                        0.2s   <<--
 => [stage-0 3/4] RUN --mount=type=bind,source=package...     1.1s   <<--
 => [stage-0 4/4] COPY . .                                    0.1s   <<--
 => exporting to image                                        0.2s
 => => exporting layers                                       0.2s
 => => writing image sha256:f282569b8bd0f0...016cc1adafc91    0.0s
 => => naming to docker.io/library/ddd-book:ch8.node
```

I've snipped the output, but you can see four numbered steps creating four image layers. These map to the instructions in the Dockerfile.

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

```
$ docker images
REPO        TAG        IMAGE ID      CREATED         SIZE
ddd-book    ch8.node   24dd040fa06b  18 minutes ago  242MB
```

Congratulations, you've containerized the app as an OCI image!

Run a **`docker inspect ddd-book:ch8.node`** command to verify the image and see the settings from the Dockerfile. You should be able to see the image layers and metadata such as the **`Exposed Ports`**, **`WorkingDir`**, and **`Entrypoint`** values.

```
$ docker inspect ddd-book:ch8.node
[
    {
        "Id": "sha256:24dd040fa06baf6e40144c5a59f99a749159a932ecebb737751f
        "RepoTags": [
            "ddd-book:ch8.node"
            <Snip>
            "ExposedPorts": {
                "8080/tcp": {}
            "WorkingDir": "/usr/src/app",
            "Cmd": [
                "/bin/sh",
                "-c",
                "node app.js"
            ],
            <Snip>
            "Layers": [
                "sha256:5f4d9fc4d98de91820d2a9c81e501c8cc6429bc8758b43fcb2
                "sha256:6b20c4e93dbab9786f96268bbe32c208d385f2c4490a278ad3
                "sha256:012c308a78ec993a47fdb7c4c6d17b53d8ce2649a463be28ae
                "sha256:35a839ac7cc922afd896a0297e692141c77ed6e03eff6a70db
                "sha256:918caa8070410ccfb2c5b3b4d62ca66742c46bf21fe0bd4337
                "sha256:a48b3b3d0c5a693840e7e4abd7971f130b4447573483628bcb
                "sha256:ea2d4594dbbef4009441a33dd1dd4c5076d7fe09a171381a6b
            ]
    <Snip>
```

You might wonder why the image has seven layers when only four Dockerfile instructions created layers. This is because the `node:20.8.0-alpine` base image already had four layers. Therefore, the `FROM` instruction pulled a base image with four layers, and then the `WORKDIR, RUN` and `COPY` instructions added three more layers. You can see this in Figure 8.2.
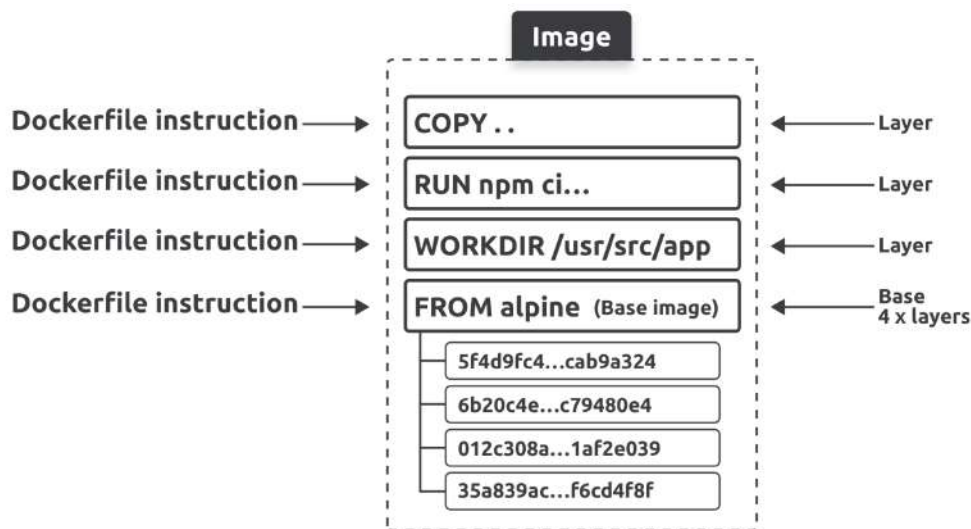
Figure 8.2 - Dockerfile and image layers

Push the image to Docker Hub

This is an optional section, and you'll need a Docker Hub account to follow along. Go to `hub.docker.com` and sign up for a free one now.

You'll complete the following steps:

1. Login to Docker Hub
2. Re-tag the image
3. Push the image

After creating images, you'll normally push them to a registry where you can keep them safe and make them accessible to teammates and clients. Lots of registries exist, but Docker Hub is the most common public registry and is where Docker pushes images by default.

Log in to Docker Hub.

```
$ docker login
USING WEB-BASED LOGIN
To sign in with credentials on the command line, use 'docker login -u <use

Your one-time device confirmation code is: PNXK-SGJG
Press ENTER to open your browser or submit your device code here: https://

Login Succeeded
```

Once logged in, you need to re-tag the image. This is because Docker uses the image tag to determine which registry and repository to push it to.

If you run a `docker images` command, you'll see an image tagged as `ddd-book:ch8.node` . If you push this image, Docker will try to push it to a repository called `ddd-book` on Docker Hub. However, no such repository exists, and the command will fail.

Run the following command to re-tag the image to include your Docker ID. The format of the command is `docker tag <current-tag> <new-tag>` , and it creates an additional tag for the same image.

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

Run another `docker images` command to see the image with both tags. Notice how everything is identical except the `REPO` column. This is because it's the same image with different names.

```
$ docker images
REPO                   TAG        IMAGE ID      CREATED         SIZE
nigelpoulton/ddd-book  ch8.node   24dd040fa06b  38 minutes ago  268M
ddd-book               ch8.node   24dd040fa06b  38 minutes ago  268M
```

Push it to Docker Hub. You'll need to be logged in with your Docker ID for this to work, and you'll need to use your Docker

ID instead of mine.

```
$ docker push nigelpoulton/ddd-book:ch8.node
The push refers to repository [docker.io/nigelpoulton/ddd-book]
e4ef261755c8: Pushed
d25f74b85615: Pushed
7e1aebde141d: Pushed
7b3f8039e3c4: Pushed
2a2799ae89a2: Mounted from library/node
4927cb899c33: Mounted from library/node
579b34f0a95b: Pushed
ced319b3ffb5: Pushed
ch8.node: digest: sha256:24dd040fa06baf...1f7f862963527a size: 856
```

Figure 8.3 shows how Docker figured out where to push the image.
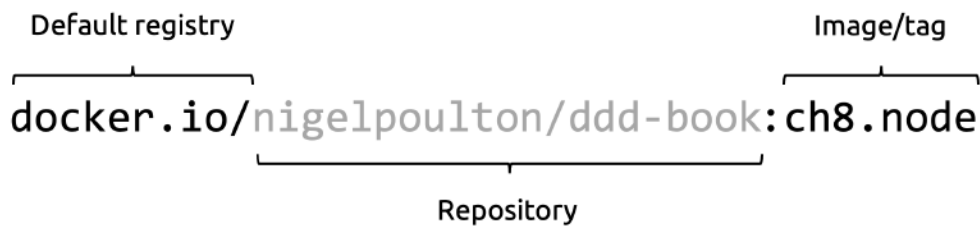
Figure 8.3

Now that you've pushed the image 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

As previously mentioned, the application is a web server listening on port `8080` .

Run the following command to start it as a container. You'll have to delete the `nigelpoulton` image prefix or replace it with your ID.

```
$ docker run -d --name c1 \
  -p 5005:8080 \
  nigelpoulton/ddd-book:ch8.node
```

The `-d` flag runs the container in the background, and the `--name` flag calls it **c1**. The `-p 5005:8080` maps port `5005` on your Docker host to port `8080` inside the container, which means you'll be able to point a browser to port `5005` and reach the app. The last line tells Docker to base the container on the `nigelpoulton/ddd-book:ch8.node` image you just built.

Docker will use the local copy of the image from the previous steps. It only pulls a copy from Docker Hub if it doesn't have a local copy.

Check the container is running and verify the port mapping.

```
$ docker ps

ID     IMAGE              COMMAND            STATUS      PORTS
49..   ddd-book:ch8.node  "node ./app.js"    UP 6 secs   0.0.0.0:5005->8080/
```

I've snipped the output for readability, but the container is running, and port `5005` on the Docker host maps to port `8080` in the container.

Test the app

Open a web browser and point it to the DNS name or IP address of your Docker host on port `5005` . If you're using Docker Desktop or a similar local environment, you can connect to `localhost:5005` . Otherwise, use the IP or DNS of the Docker host on port `5005` .

You should see the app as shown in Figure 8.4.

Figure 8.4

You can try the following if it doesn't work:

1. Run a `docker ps` command to ensure the **c1** container is running
2. Check port mapping is correct — `0.0.0.0:5005->8080/tcp`
3. Check that firewall and other network security settings aren't blocking traffic to your Docker host on port `5005`

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

Looking a bit closer

Now that you've containerized the application 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.

You can insert comments by starting a line with the `#` character, and the builder will ignore them.

All non-comment lines are called *instructions* or *steps* and take the format `<INSTRUCTION> <arguments>`. Instruction names are not case-sensitive, but it's common to write them in UPPERCASE to make the file easier to read.

Some instructions create new layers, whereas others add metadata.

Examples of instructions that create new layers are `FROM`, `RUN`, `COPY` and `WORKDIR`. Examples that create metadata include `EXPOSE`, `ENV`, `CMD`, and `ENTRYPOINT`. The premise is this:

- Instructions that add *content*, such as files and programs, create new layers
- Instructions that don't add content don't add layers and only create metadata

You can run a `docker history` command against any image to see the instructions that created it.

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

IMAGE          CREATED BY                                       SIZE      COMM
24dd...a06b    CMD ["/bin/sh" "-c" "node app.js"]               0B        buil
<missing>      EXPOSE map[8080/tcp:{}]                          0B        buil
<missing>      COPY . . # buildkit                              98kB      buil
<missing>      USER node                                        0B        buil
<missing>      RUN /bin/sh -c npm ci --omit=dev # buildkit      13.6MB    buil
<missing>      WORKDIR /usr/src/app                             16.4kB    buil
<missing>      ENV NODE_ENV=production                          0B        buil
<Snip>
<missing>      ADD alpine-minirootfs-3.21.0-aarch64.tar.gz      8.84MB    8.35
```

A few things are worth noting from the output.

The bottom few lines that I've snipped from the book related to the history of the **node:23.3.0-alpine** base image that was pulled by the **FROM** instruction.

All lines ending with **buildkit.dockerfile.v0** relate to instructions from the Dockerfile used to build the image.

The **CREATED BY** column lists the exact Dockerfile instruction that created the layer or metadata.

Lines with a non-zero value in the **SIZE** column created new layers, whereas the lines with **0B** only added metadata. In this example, three lines/instructions created layers.

Run a **docker inspect** to see the list of image layers.

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

<Snip>
},
"RootFS": {
    "Type": "layers",
    "Layers": [
                "sha256:5f4d9fc4d98de91820d2a9c81e501c8cc6429bc8758b43fcb2
                "sha256:6b20c4e93dbab9786f96268bbe32c208d385f2c4490a278ad3
                "sha256:012c308a78ec993a47fdb7c4c6d17b53d8ce2649a463be28ae
                "sha256:35a839ac7cc922afd896a0297e692141c77ed6e03eff6a70db
                "sha256:918caa8070410ccfb2c5b3b4d62ca66742c46bf21fe0bd4337
                "sha256:a48b3b3d0c5a693840e7e4abd7971f130b4447573483628bcb
                "sha256:ea2d4594dbbef4009441a33dd1dd4c5076d7fe09a171381a6b
    ]
},
```

As previously mentioned, the output shows seven layers because the base image had four layers, and the Dockerfile added three more.

Figure 8.5 maps the Dockerfile instructions to image layers. The bold instructions with arrows create layers; the others create metadata. The layer IDs will be different in your environment.
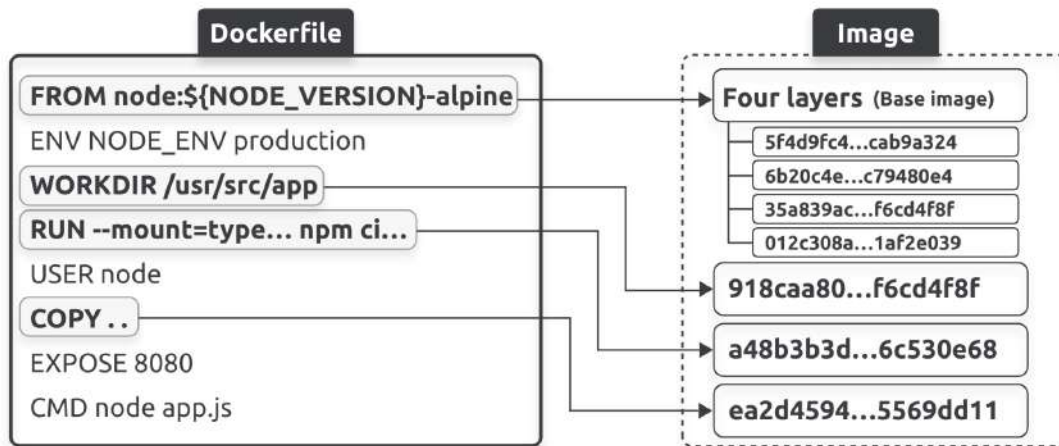
Figure 8.5

Note: *Older builders didn't create a layer for* `WORKDIR` *instructions. However, the instruction modifies filesystem permissions and the current builder creates a very small layer. This behavior may change in the future.*

It's generally considered a good practice to use *Docker Official Images* and *Verified Publisher* images as the *base layer* for new images you create. This is because they maintain a high standard and quickly implement fixes for known vulnerabilities.

Moving to production with multi-stage builds

When it comes to container images... *big is bad!* For example:

- Big means slow

- Big means more potential vulnerabilities
- Big means a larger attack surface

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

This is where *multi-stage builds* come into play.

At a high level, multi-stage builds use a single Dockerfile with multiple `FROM` instructions — each `FROM` instruction represents a new *build stage*. This allows you to have a *stage* where you do the heavy lifting of building the app inside a large image with compilers and other build tools, but then you have another stage where you copy the compiled app into a *slim image* for production. The builder can even run different stages in parallel for faster builds.

> **Note:** *A slim image is a very small image intended for production use that only contains files and apps that are absolutely necessary to run the application. They do not include shells, package managers, or troubleshooting tools.*

Figure 8.6 shows a high-level workflow. Stage 1 builds an image with all the required build and compilation tools. Stage 2 copies the app code into the image and builds it. Stage 3 creates a small

production-ready image containing only the compiled app and anything needed to run it.
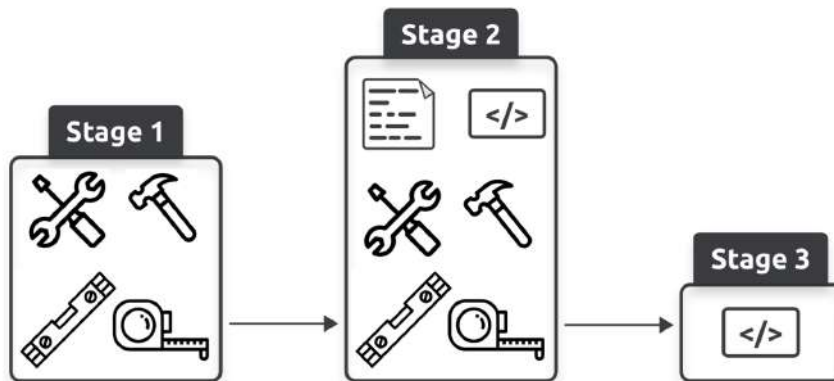
Figure 8.6

Let's look at an example!

We'll work with the code in the `multi-stage` folder of the book's GitHub repo. It's a simple Go app with a client and server borrowed from the *Docker samples buildme* repo on GitHub. Don't worry if you're not a Go programmer; you don't need to be. You only need to know that it compiles the *client* and *server* apps into executable files that **do not** need the Go language or any other tools or runtimes to execute.

Here's the Dockerfile:

```
FROM golang:1.23.4-alpine AS base          <<---- Stage 0
WORKDIR /src
COPY go.mod go.sum .
RUN go mod download
COPY . .

FROM base AS build-client                  <<---- Stage 1
RUN go build -o /bin/client ./cmd/client

FROM base AS build-server                  <<---- Stage 2
RUN go build -o /bin/server ./cmd/server

FROM scratch AS prod                       <<---- Stage 3
COPY --from=build-client /bin/client /bin/
COPY --from=build-server /bin/server /bin/
ENTRYPOINT [ "/bin/server" ]
```

The first thing to note is that there are four `FROM` instructions. Each of these is a distinct *build stage*, and Docker numbers them starting from 0. However, we've given each stage a friendly name:

- Stage 0 is called `base` and builds an image with compilation tools, etc
- Stage 1 is called `build-client` and compiles the client executable
- Stage 2 is called `build-server` and compiles the server executable
- Stage 3 is called `prod` and copies the client and server executables into a slim image

Each stage outputs an intermediate image that later stages can use. However, Docker deletes them when the final stage completes.

The goal of the `base` stage is to create a reusable build image with all the tools stages 1 and 2 need to build the client and server applications. The image created by this stage is only used to compile the executables and not for production. It pulls the `golang:1.23.4-alpine` image, which is over 350MB when uncompressed. It sets the working directory to `/src` and copies in the `go.mod` and `go.sum` files from your working directory. These files list the application dependencies and hashes. After that, it uses the `RUN` instruction to install the dependencies and then the `COPY` instruction to copy the application source code into the image. All of this creates a large image with three layers containing a lot of build stuff but not much app stuff. When this build stage completes, it outputs a large image that later stages can use.

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 issues a `RUN` instruction to compile the client app into a binary executable. The goal of this stage is to create an image with the compiled *client* binary that later stages can reference.

The `build-server` stage does the same for the *server* component and outputs a similar image for use by later stages.

The `prod` stage pulls the minimal `scratch` image. It then runs two `COPY --from` instructions to copy the compiled *client app* from the `build-client` stage and the compiled *server app* from the `build-server` stage. It then tells Docker to run the server app when it's started as a container. This stage outputs the final production image containing just the client and server binaries inside a tiny scratch image and the metadata telling Docker how to start the app.

The builder will run the `base` stage first, then run the `build-client` and `build-server` stages in parallel, and finally run the `prod` stage.

It will always attempt to run stages in parallel, but it can only do this when no dependencies exist. For example, the `build-client` and `build-server` stages both start with `FROM base...`, meaning they depend on the `base` stage and cannot run until that stage is built. However, the `build-client` and `build-server` can run in parallel because they don't depend on each other. To work out if build stages can run in parallel, start reading the Dockerfile from the top and check if the `FROM`

instructions reference other `FROM` instructions immediately before or after — if they do, they can't run in parallel.

Let's see it in action.

Change into the `multi-stage` directory and verify the Dockerfile and associated app files exist.

```
$ ls -l

total 28
-rw-rw-r-- 1 ubuntu ubuntu  368 Mar 25 10:09 Dockerfile
-rw-rw-r-- 1 ubuntu ubuntu  433 Mar 25 10:09 Dockerfile-final
-rw-rw-r-- 1 ubuntu ubuntu  305 Mar 25 10:09 README.md
drwxrwxr-x 4 ubuntu ubuntu 4096 Mar 25 10:09 cmd
-rw-rw-r-- 1 ubuntu ubuntu 1013 Mar 25 10:09 go.mod
-rw-rw-r-- 1 ubuntu ubuntu 5631 Mar 25 10:09 go.sum
```

Build the image and watch the **build-client** and **build-server** stages execute in parallel. This can significantly improve the performance of large builds.

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

[+] Building 14.6s (15/15) FINISHED
 => [internal] load build definition from Dockerfile                    0.0s
 => => transferring dockerfile: 736B                                    0.0s
 <Snip>
 => [build-client 1/1] RUN go build -o /bin/client ./cmd/client         5.1s
 => [build-server 1/1] RUN go build -o /bin/server ./cmd/server         5.1s
 <Snip>
```

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

```
$ docker images

REPO    TAG      IMAGE ID        CREATED        SIZE
multi   full     a7a01440f2b5    5 seconds ago  26.7MB
```

The final production image is only 26MB, much smaller than the 350MB+ base image pulled by the `base` stage to build and compile the app. This is because the final `prod` stage extracted the compiled client and server binaries and placed them in a tiny new scratch image.

Run a `docker history` to see the final production image. It only has two layers — one created by copying in the client binary and the other by copying in the server binary. None of the previous build stages are included in the final production image.

```
$ docker history multi:full
IMAGE           CREATED        CREATED BY                          SIZE
a7a01440f2b5    2 minutes ago  ENTRYPOINT ["/bin/server"]          0B
<missing>       2 minutes ago  COPY /bin/server /bin/ # buildkit   8.64MB
<missing>       2 minutes ago  COPY /bin/client /bin/ # buildkit   8.53MB
```

Multi-stage builds and build targets

You can also build multiple images from a single Dockerfile.

The previous example compiled client and server apps and copied both into the same image. However, Docker makes it easy to create a separate image for each by splitting the final `prod` stage into two stages as follows:

```
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               <<---- New stage
COPY --from=build-client /bin/client /bin/
ENTRYPOINT [ "/bin/client" ]

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

I've pre-created the Dockerfile and called it `Dockerfile-final` in the `multi-stage` folder, but you can see the only change is splitting the final `prod` stage into two stages — one for the

client build and the other for the server build. With a Dockerfile like this, you tell a `docker build` command which of the two final stages to target for the build.

Let's do it.

Run the following two commands to create two different images from the same `Dockerfile-final` file. Both commands use the `-f` flag to tell Docker to use the `Dockerfile-final` file. They also use the `--target` flag to tell the builder which stage to build from.

```
$ 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 image sizes.

```
$ docker images
REPOSITORY      TAG       IMAGE ID        CREATED          SIZE
multi           full      a7a01440f2b5    4 minutes ago    26.7MB
multi           server    a75778df1b9c    4 seconds ago    11.7MB
multi           client    02b621e9415f    12 seconds ago   11.9MB
```

You now have three images, and the `client` and `server` images are each about half the size of the `full` image. This makes sense because the `full` image contains the client and server binaries, whereas the others only include one.

Buildx, BuildKit, drivers, and Build Cloud

This section takes a quick look at the major build components.

Behind the scenes, Docker's build system has a client and server:

- **Client:** Buildx
- **Server:** BuildKit

Buildx is Docker's latest and greatest build client. It's implemented as a CLI plugin and supports all the latest features of BuildKit, such as multi-stage builds, multi-architecture images, advanced caching, and more. It's been the default build client since Docker v23.0 and Docker Desktop v4.19. This means every time you run a `docker build` command, you're automatically using the Buildx builder.

You can configure Buildx to talk to multiple BuildKit instances, and we call each instance of BuildKit a *builder*. Builders can run

on your local machine, in your cloud or datacenter, or Docker's
Build Cloud.

If you point buildx at a local builder, image builds will be done
on your local machine. If you point it at a remote builder, such
as *Docker Build Cloud*, builds will be done on remote
infrastructure.

Figure 8.7 shows a Docker environment configured to talk to a
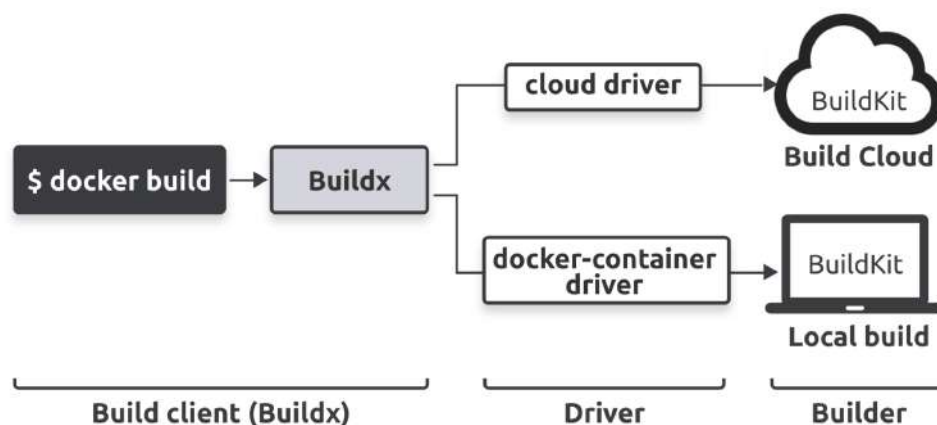local and a remote builder.

Figure 8.7 - Docker build architecture

In the diagram, the local builder uses the `docker-container`
driver to create a local BuildKit instance inside a dedicated
container. All builds using this driver will run in the dedicated
container. The other option uses the cloud driver to send builds

to Docker's Build Cloud service. Build Cloud offers fast builds and a shared cache but requires a paid subscription.

When you run a `docker build` command, buildx interprets the command and sends the build request to the selected *builder*. This includes the Dockerfile, command line arguments, caching options, export options, and the build context (app and dependency list). The *builder* performs the build and exports the image. The Buildx client monitors the build and reports on progress.

Run the following command to see the builders you have configured on your system. I've trimmed the output in the book, but you can see a local and a remote builder.

```
$ docker buildx ls
NAME/NODE                DRIVER/ENDPOINT        PLATFORMS
builder *                docker-container
  builder0               desktop-linux          linux/arm64, linux/amd6
                                                 linux/riscv64, linux/pp
                                                 linux/386, linux/mips64
                                                 linux/arm/v7, linux/arm
cloud-nigelpoulton-ddd   cloud
  linux-arm64            cloud://nigel...arm64   linux/arm64*
  linux-amd64            cloud://nigel...amd64   linux/amd64*, linux/amd
                                                 linux/amd64/v3,linux/am
<Snip>
```

Notice how the first builder supports more platforms than the cloud builder. This is because the **docker-container** driver utilizes QEMU to emulate target hardware. It usually works but can be slow.

The second builder is Docker's Build Cloud, which only supports AMD and ARM builds. Builds running in Build Cloud run on native hardware and offer a shared cache so that teammates can share a common cache for even faster builds. Complex builds can be much quicker when executed on native hardware such as Build Cloud.

Run a **docker buildx inspect** command against one of your builders.

```
$ docker buildx inspect cloud-nigelpoulton-ddd

Name:          cloud-nigelpoulton-ddd
Driver:        cloud
Nodes:
Name:      linux-arm64
Endpoint:  cloud://nigelpoulton/ddd_linux-arm64
Status:    running
Buildkit:  v0.16.0
Platforms: linux/arm64*, linux/arm/v6, linux/arm/v7
Labels:
 org.mobyproject.buildkit.worker.executor:        oci
 org.mobyproject.buildkit.worker.hostname:        nigelpoulton_ddd-cloud_
 org.mobyproject.buildkit.worker.network:         host
 org.mobyproject.buildkit.worker.oci.process-mode: sandbox
 org.mobyproject.buildkit.worker.selinux.enabled:  false
```

```
   org.mobyproject.buildkit.worker.snapshotter:       overlayfs
GC Policy rule#0:
 All:        true
 Keep Bytes: 25GiB
<Snip>
```

Let's see how to perform multi-architecture builds.

Multi-architecture builds

You can use the `docker build` command to build images for
multiple platforms and CPU architectures, including ones
different from your local machine. For example:

- Docker on an AMD machine can build ARM images
- Docker on an ARM machine can build AMD images

You also have the option to perform builds locally or in the
cloud. Both work with the standard `docker build` command
and only require minimal backend configuration.

Run the following command to list your current builders.
Remember, a *builder* is an instance of BuildKit that will perform
builds.

```
$ docker buildx ls
NAME/NODE                      DRIVER/ENDPOINT         PLATFORMS
builder *                      docker-container
```

```
    builder0                    desktop-linux               linux/arm64, linux/amd6
                                                            linux/riscv64, linux/pp
                                                            linux/386, linux/mips64
                                                            linux/arm/v7, linux/arm
  cloud-nigelpoulton-ddd        cloud
    linux-arm64                 cloud://nigel...arm64   linux/arm64*
    linux-amd64                 cloud://nigel...amd64   linux/amd64*, linux/amd
                                                        linux/amd64/v4

  <Snip>
```

The book's output shows two builders; the one with the asterisk (*) is the default builder. In this example, the default builder is called **builder** and uses the **docker-container** driver to perform builds inside a local build container. Unless you specify a different builder, all builds will run inside this build container. It supports multiple architectures, including AMD, ARM, RISC-V, s390x, and more.

If you don't already have one, create a new builder called *container* that uses the **docker-container** driver with the following command.

```
  $ docker buildx create --driver=docker-container --name=container
```

Run another **docker buildx ls** to show the new builder. Don't worry if it shows as present but inactive.

Make it the default builder.

```
$ docker buildx use container
```

Change into the `web-app` directory and run the following command to build the app into AMD and ARM images and export them directly to Docker Hub.

Be sure to substitute your Docker ID as the command pushes directly to Docker Hub and will fail if you try to push it to my repositories. If you don't have a Docker Hub account or don't want to push the images, you can replace the `--push` with `--load`.

```
$ docker buildx build --builder=container \
  --platform=linux/amd64,linux/arm64 \
  -t nigelpoulton/ddd-book:ch8.1 --push .

[+] Building 79.3s (26/26) FINISHED
 <Snip>
 => [linux/arm64 2/5] RUN apk add --update nodejs npm curl
 => [linux/amd64 2/5] RUN apk add --update nodejs npm curl
 => [linux/amd64 3/5] COPY . /src
 => [linux/amd64 4/5] WORKDIR /src
 => [linux/amd64 5/5] RUN  npm install
 => [linux/arm64 3/5] COPY . /src
 => [linux/arm64 4/5] WORKDIR /src
 => [linux/arm64 5/5] RUN  npm install
 => exporting to image
 <Snip>
```

```
   => => pushing layers
   => => pushing manifest for docker.io/nigelpoulton/ddd-book:web0.2@sha256:
   => [auth] nigelpoulton/ddd-book:pull,push token for registry-1.docker.io
```

I've snipped the output, but you can still see two important
things:

- Each Dockerfile instruction executed twice — once for AMD
  and once for ARM
- The last few lines show the image layers being pushed
  directly to Docker Hub

Now that you've performed a build, the builder will show as
active and list the architectures it supports.

Figure 8.8 shows how the images for both architectures appear
on Docker Hub under the same repository and tag.

Figure 8.8 - Multi-platform image

You can also perform the builds using Docker Build Cloud. This is a cloud-based service that offers fast builds and lets you share your build cache with teammates. It requires a paid subscription.

If you have a Docker subscription that grants you access to Build Cloud, you can go to `build.docker.com` and configure your first cloud builder. You can also create cloud builders from the CLI as follows. If you're following along, you'll need to give yours a different name.

```
$ docker buildx create --driver cloud nigelpoulton/ddd
```

Once you have a cloud builder, you can either make it your default builder with a `docker buildx use <builder>` command, or you can specify it when performing individual builds.

The following command uses the `--builder` flag to use the **cloud-nigelpoulton-ddd** cloud builder to build the same images as in the previous steps. Remember to use your own cloud builder if you're following along.

```
$ docker buildx build \
  --builder=cloud-nigelpoulton-ddd \
  --platform=linux/amd64,linux/arm64 \
```

```
    -t nigelpoulton/ddd-book:ch8.1 --push .

 => [internal] connected to docker build cloud service
 <Snip>
```

At the time of writing, Build Cloud supports various AMD and
ARM architectures, whereas the `docker-container` driver
supports more but is slower and less reliable.

A few good practices

Let's finish the chapter with a few best practices. This isn't a full
list, and the advice applies to local builds and cloud builds.

Leverage the build cache

BuildKit uses a cache to speed up builds. The best way to see the
impact is to build a new image on a clean Docker host and then
repeat the same build immediately after. The first build will pull
images and take time to build layers. The second build will
instantly complete because the layers and other artifacts from
the first build are cached and leveraged by later builds.

If you use a local builder, the cache is only available to other
builds you execute on the same system. However, your entire
team can share the cache on Docker Build Cloud.

For each build, the builder iterates through the Dockerfile one line at a time, starting from the top. For each line, it checks if it already has the layer in its cache. If it does, a *cache hit* occurs, and it uses the cached layer. If it doesn't, a *cache miss* occurs, and it builds a new layer from the instruction. Cache hits are one of the best ways to make builds faster.

Let's take a closer look.

Assume the following Dockerfile:

```
FROM alpine
RUN apk add --update 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 you already have a copy of this image, the builder moves on to the next instruction. If you don't have a copy, it pulls it from Docker Hub.

The next instruction (`RUN apk...`) runs a command to update package lists and install `nodejs` and `npm`. Before executing the instruction, Docker checks the build cache for a layer built from

the same base image using the same instruction. In this case, it's looking for a layer built by executing the `RUN apk add --update nodejs npm` instruction directly on top of the `alpine:latest` image.

If it finds a matching layer, it links to that layer and continues the build with the cache intact. If it does **not** find a matching layer, it invalidates the cache and builds the layer. Invalidating the cache means the builder **must** execute all remaining Dockerfile instructions in full and cannot use the cache.

Let's assume Docker had a cached layer for the `RUN` instruction and that the layer's ID is **AAA**.

The next instruction runs a `COPY . /src` command to copy the app code into the image. The previous instruction scored a cache hit, meaning Docker can check if it has a cached layer built by running a `COPY . /src` against the **AAA** layer. If it has a cached layer for this, it links to the layer and proceeds to the next instruction. If it doesn't have a cached layer, it builds it 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 things.

Any time an instruction results in a cache miss, the cache is invalidated and no longer checked for the rest of the build. This means you should write your Dockerfiles so that instructions most likely to invalidate the cache go near the end of the Dockerfile. This allows builds to benefit from the cache for as long as possible.

You can force a build to ignore the cache by running `docker build` with the `--no-cache` option.

It's also important to understand that `COPY` and `ADD` instructions include logic to ensure the *content* you're copying into the image hasn't changed since the last build. For example, you might have a cached layer that Docker built by running a `COPY . /src` against the **AAA** image. However, if the *files* that the `COPY . /src` instruction copies into the layer have changed since the cached layer was built, you cannot use the cached layer as you'd get old versions of the files. To protect against this, Docker performs checksums against each file it copies. If the checksums don't match, the cache is invalidated, and Docker builds a new layer.

Only install essential packages

We often joke that we *install the entire internet* when we build apps. As a quick example, the simple Node.js app used earlier in

the chapter depends on two packages:

- Express
- Pug

However, these packages depend on other packages, which in turn depend on others. At the time of writing, building this simple application with two dependencies actually downloads **over 110 packages!**

Fortunately, some package managers provide a way for you to only download and install essential packages instead of *the entire internet*. One example is the `apt` package manager that lets you specify the `no-install-recommends` flag so that it only installs packages in the *depends* field and not every *recommended* and *suggested* package. Each package manager does this differently, but it's worth investigating as it can massively impact the size of your images.

Clean up

If you've followed along, you'll have one running container and several images in your local image repository. You should delete the running container and, optionally, the local images.

Run the following command to delete the container.

```
$ docker rm c1 -f
```

Optionally delete the local images with the following command. Be sure to use the names of the images in your environment.

```
$ docker rmi \
  multi:full multi:client multi:server ddd-book:ch8.node nigelpoulton/ddd-
```

Containerizing an app – The commands

- `docker build` containerizes applications. It reads a Dockerfile and follows the instructions to create an OCI image. The `-t` flag tags the image, and the `-f` flag lets you specify the name and location of the Dockerfile. 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. It's usually the first instruction in a Dockerfile, and it's considered a good practice to build from *Docker Official Images* or images from *Verified Publishers*. `FROM` is also used to identify new build stages in multi-stage builds.
- The Dockerfile `RUN` instruction lets you run commands during a build. It's commonly used to update packages and

install dependencies. Every `RUN` instruction creates a new image layer.

- The Dockerfile `COPY` instruction adds files to images, and you'll regularly use it to copy your application code into a new image. Every `COPY` instruction creates an image layer.
- The Dockerfile `EXPOSE` instruction documents an application's network port.
- The Dockerfile `ENTRYPOINT` and `CMD` instructions tell Docker how to run the app when starting a new container.
- Some other Dockerfile instructions include `LABEL`, `ENV`, `ONBUILD`, `HEALTHCHECK` and more.

Chapter summary

This chapter taught you how to containerize an application. This is the process of building an app into a container image and running it as a container.

You pulled some application source code from GitHub and used the `docker init` command to auto-generate a Dockerfile with instructions telling Docker how to build the app into a container image. You then used `docker build` to create the image, `docker push` to push it to Docker Hub, and `docker run` to run it as a container.

Along the way, you learned that some Dockerfile instructions add content to an image and therefore create new layers. Instructions that don't add content only create metadata.

After that, you learned how multi-stage builds allow you to create small and efficient production images without the bloat carried over from compiling the app.

After that, you learned that buildx is the default build client that integrates with the latest features of the BuildKit build engine. You learned how to create local and remote builders (BuildKit instances) and how to use them to perform multi-architecture builds.

You also learned the importance of the build cache for speeding up builds and how to optimize Dockerfiles to leverage the build cache.

9: Multi-container apps with Compose

In this chapter, you'll deploy and manage a multi-container AI chatbot using Docker Compose. Along the way, we'll look at how the chatbot app works.

The app is a multi-container app with three services — a web interface, a backend API, and a model server. The model server runs [Ollama](#) to execute a local instance of a large language model (LLM) that's pre-trained to answer developer-related questions and keep answers short. LLMs require a lot of memory and run best GPUs. However, not everyone has access to GPUs, so I've configured the app to work on systems with CPUs and a minimum of 8GB of RAM. I'll show you how to configure the app for GPUs if your Docker host has NVIDIA GPUs.

When referring to Docker Compose, we usually shorten it to *Compose* and always write it with a capital "C".

I've organized the chapter as follows:

- Compose – The TLDR
- Compose background
- Install Compose
- The AI chatbot app

- Compose files
- Deploy the app
- Use the app
- Inspect the app
- Manage the app
- Inspect the Ollama configuration

Compose – The TLDR

We create modern cloud-native applications by writing small specialized services and combining them into a meaningful app. We call them *microservices applications*, and they bring a lot of benefits, such as self-healing, autoscaling, and rolling updates. However, they can be complex.

For example, you might have a microservices app with the following seven services:

- Web frontend
- Ordering
- Catalog
- Backend datastore
- Logging
- Authentication
- Authorization

Instead of hacking these services together with complex scripts and long `docker` commands, Compose lets you describe them in a simple YAML file called *Compose file*. You can then deploy the entire app by running a `docker compose up` command that sends the Compose file to Docker, and Docker takes care of the rest. Behind the scenes, Docker reads the file and runs the commands to deploy everything defined in the file.

This means that Compose files are as important as your code files, and you should manage them in a version control system such as Git.

That's the basics. Let's dig deeper.

Compose background

When Docker was new, it had no way to deploy and manage multi-container apps. To fill this gap, a company called *Orchard Labs* built a tool called *Fig* that ran on top of Docker and let you define complex multi-container apps in a simple YAML file. You could even use the `fig` command-line tool to manage the entire application lifecycle.

Behind the scenes, Fig would read the YAML file and call the appropriate `docker` commands.

Fig was so good that Docker, Inc. acquired Orchard Labs and rebranded Fig as *Docker Compose*. They renamed the command-line tool from `fig` to `docker-compose`, and then, more recently, they folded it into the main `docker` CLI with its own `compose` sub-command. Later in the chapter, you'll run simple `docker compose` commands to deploy and manage the multi-container chatbot app.

There is also a [Compose Specification](#) driving Compose as an open standard for defining multi-container microservices apps. It's a community-led specification and kept separate from the Docker implementation to maintain better governance and clearer demarcation. However, *Docker Compose* is the *reference implementation*, and you should expect Docker to implement the full spec.

Reading the specification documentation is a great way to learn the details.

Installing Compose

All modern versions of Docker come with Docker Compose pre-installed, and you no longer need to install it as a separate application.

Test it with the following command. Be sure to use the `docker compose` command instead of the older `docker-compose`.

```
$ docker compose version
Docker Compose version v2.31.0
```

The AI chatbot app

We'll deploy the multi-container AI chatbot app shown in Figure 9.1. It's a clone of an [official Docker app](#) available on Docker Hub. I've cloned it here so that I control the version you'll use and ensure it stays in sync with the instructions and workflow of the book. I've also provided the app code and Dockerfiles, and I've pre-built multi-architecture images — none of which the official Docker app provided at the time of writing.
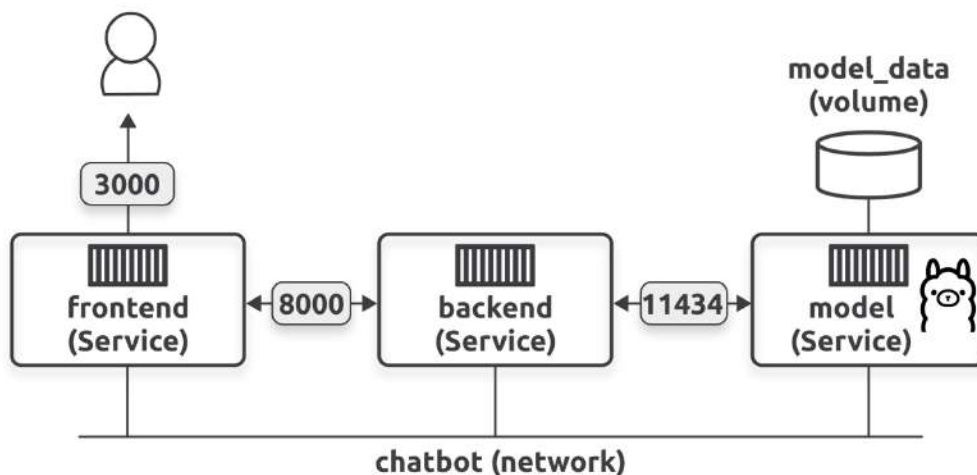

Figure9.1 - Sample app

The app has a network, a volume, and three services. Each service performs a specific task, and as shown in the diagram, is deployed in its own container.

The **frontend** service runs a web server that exposes the chat interface on port `3000` and talks to the backend API service on port `8000`. The **backend** service runs a FastAPI server that listens on port `8000` and forwards prompts (questions) from the frontend service to the model server on port `11434` and streams responses. The **model** server uses the Ollama runtime to execute a local instance of a Mistral *instruct* LLM optimized as a coding assistant. The model service mounts the **model_data** volume to store the LLM. All three services connect to the **chatbot** network.

This is all defined in the `compose.yaml` file in the `ai-compose` folder of the book's GitHub repo.

In this example, all three *services* define and deploy a single container. However, it's possible for a service to deploy multiple identical containers to help with performance and load-balancing. You'll do this in the Swarm chapter later in the book.

If you haven't already done so, clone the repo on the machine you're running your Docker commands from. You'll need `git`

installed, and the command will create a new directory called
`ddd-book` .

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

Change into the `ddd-book/ai-compose` directory and list its
contents.

```
$ cd ddd-book/ai-compose/
$ ls -l
total 24
drwxr-xr-x@ 5 nigelpoulton  staff   160 18 Dec 12:25 backend
drwxr-xr-x@ 8 nigelpoulton  staff   256 18 Dec 14:32 frontend
drwxr-xr-x@ 4 nigelpoulton  staff   128 18 Dec 12:25 model
-rw-r--r--@ 1 nigelpoulton  staff  1338 18 Dec 14:28 compose.yaml
```

This directory is your *build context*, containing all the app code
and configuration files you need to deploy and manage the app.

- The `backend` folder contains the application code and
  Dockerfile for the **backend** service
- The `frontend` folder contains the application code and
  Dockerfile for the **frontend** service
- The `model` folder contains the application code and
  Dockerfile for the **model** service
- The `compose.yaml` file is the Compose file that describes the
  app (three services, one network, and one volume)

In a moment, you'll run a `docker compose` command to deploy the app. That sends the `compose.yaml` file to Docker, which will read it and create the **chatbot** network, the **model_data** volume, and a single container for each service (it's possible to deploy multiple containers per service and you'll do this in Chapter 12, Deploying apps to Swarm). It will mount the **model_data** volume into the **model** container and connect all three containers to the **chatbot** network.

Let's take a closer look at the Compose file.

Compose files

Compose defines applications in YAML files called *Compose files* that are usually called `compose.yaml` or `compose.yml`. However, you can use the `-f` flag to specify a different file name at run time.

Here is the Compose file we'll be using.

```
volumes:                              <<---- Volumes are defined in this
  model_data:
networks:                             <<---- Networks are defined in this
  chatbot:
services:                             <<---- Services are defined in this
  frontend:                           <<---- This block defines the front
    image: nigelpoulton/ddd-book:ai-fe
    command: npm run start
    networks:
```
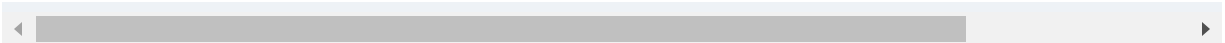
```yaml
      - chatbot
    ports:
      - "3000:3000"
    environment:
      - PORT=3000
      - HOST=0.0.0.0
    depends_on:
      - backend
  backend:                              <<---- This block defines the backe
    image: nigelpoulton/ddd-book:ai-be
    networks:
      - chatbot
    ports:
      - "8000:8000"
    environment:
      - MODEL_HOST=http://model:11434
    depends_on:
      model:
        condition: service_healthy
  model:                                <<---- This block defines the model
    build: ./model
    networks:
      - chatbot
    ports:
      - "11434:11434"
    volumes:
      - model_data:/root/.ollama
    environment:
      - MODEL=${MODEL:-mistral:latest}
    healthcheck:
      test: ["CMD-SHELL", "curl -s http://localhost:11434/api/tags | jq -e
      interval: 10s
      timeout: 5s
      retries: 50
      start_period: 900s
    deploy:
      resources:
        limits:
          memory: 8G
```

The first thing to note is that the file has three top-level keys with a block of code beneath each:

- volumes
- networks
- services

The `volumes` block defines a single volume, the `networks` block defines a single network, and the `services` block defines three services.

Let's look closer at each, starting with the `volumes` and `networks` blocks.

```
volumes:
  model_data:
networks:
  chatbot:
```

The **volumes** block defines a single volume called **model_data** that Docker will create on the host's filesystem. Later in the file, the **model** service will mount it and use it to store the LLM. The **networks** block defines a single network called **chatbot** that all three services will connect to.

The top-level `services` key is mandatory and is where you define application microservices. This file defines three microservices called **frontend**, **backend**, and **model**, and deploys a single container for each. You'll learn how to scale services with multiple identical containers in the chapter on deploying apps to Swarm.

Let's look at each in detail, starting with the `frontend` service.

```
services:
  frontend:                           <<---- Service name (containers w
    image: nigelpoulton/ddd-book:ai-fe   <<---- Base containers on this im
    command: npm run start            <<---- Run this command to start
    environment:
      - PORT=3000                     <<---- Variable telling the app to use
      - HOST=0.0.0.0                  <<---- Variable telling the app to use
    networks:
      - chatbot                       <<---- Connect container to the "chatbo
    ports:
      - "3000:3000"                   <<---- Map port 3000 on the host to 300
    depends_on:
      - backend                       <<---- Wait for backend to be running b
```

Let's explain the fields.

- `frontend:` The service's name, and all containers Docker creates as part of this service will have "frontend" in their names.

- `image:` Tells Docker which image to use when creating containers for this service.
- `command:` Specifies the command Docker will execute to start the app in every container it creates for this service.
- `environment:` Defines two environment variables the app will use to know which port and network interfaces to serve on.
- `networks:` Tells Docker to connect the service's containers to the `chatbot` network. The network should already exist or be defined in the `networks` top-level key.
- `ports:` Tells Docker to map port `3000` on the Docker host to port `3000` inside the container.
- `depends_on:` Tells Docker to wait for the **backend** service to be running before starting this one.

Stepping through the block, Compose will instruct Docker to create a container called **frontend** based on the `nigelpoulton/ddd-book:ai-fe` image. It will run an `npm run start` command to start the app, use a couple of environment variables to bind the app to port `3000` on all interfaces, connect the container to the **chatbot** network, and create a mapping so that traffic hitting the Docker host on port `3000` will reach the container on `3000`. Finally, it creates a dependency on the **backend** service, ensuring this service will not start until the **backend** service is running.

Specifying a **command** in a Compose file overrides **CMD** commands specified in images and Dockerfiles. If the image or Dockerfile uses an **ENTRYPOINT** instruction, commands specified in Compose files are appended as arguments.

The **backend** service is similar.

```
backend:                             <<---- Service name (containers will
  image: nigelpoulton/ddd-book:ai-be   <<---- Base containers on this in
  environment:
    - MODEL_HOST=http://model:11434  <<---- Variable telling app where to
  networks:
    - chatbot                        <<---- Connect container to the "chat
  ports:
    - "8000:8000"                    <<---- Map port 8000 on the host to &
  depends_on:
    model:
      condition: service_healthy     <<---- Wait for model server to be HE
```

Let's step through this one.

- `backend:` The service's name, and all containers created for this service will inherit this in their names.
- `environment:` This service doesn't specify a `command`, meaning it will execute the command specified in the Dockerfile that created the image. However, it uses an environment variable to tell the app where to find the model server.

- `network:` Tells Docker to connect **backend** service containers to the **chatbot** network.
- `ports:` Maps port `8000` on the Docker host to port `8000` inside the container.
- `depends_on.model.condition: service_healthy` ensures this service will only go live after the **model** service is running and passed a **healthcheck**.

Compose will read this block and start a container called **backend** and base it on the `nigelpoulton/ddd-book:ai-be` image. It will execute the image's **CMD** or **ENTRYPOINT** command to start the app and inject the `MODEL_HOST` environment variable into the app so it knows where to find the model server. It will connect the container to the **chatbot** network and map port `8000` on the Docker host to `8000` on the container. Finally, it creates a dependency requiring the model service to pass a healthcheck before this service accepts requests.

Finally, let's inspect the **model** service definition.

```
model:                      <<---- Service name (containers will inherit
  build: ./model            <<---- Build image from app and Dockerfile i
  environment:              <<---- Variable telling model server which b
    - MODEL=${MODEL:-mistral:latest}
  volumes:
    - model_data:/root/.ollama    <<---- Mount model_data volume to /root/
```

```
    networks:
      - chatbot                 <<---- Attach container to chatbot network
    ports:
      - "11434:11434"           <<---- Map port 11434 on Docker host to 1143
    deploy:
      resources:
        limits:
          memory: 8G            <<---- Allocate 8GB of memory to this servi
    healthcheck:                <<---- Defines healthcheck command and instr
      test: ["CMD-SHELL"...     <<---- Run this command to test the model he
      interval: 10s             <<---- Wait this long after the container fi
      timeout: 5s               <<---- Don't let checks run longer than 5 se
      retries: 50               <<---- Retry the healthcheck up to 50 times
      start_period: 600s        <<---- Keep checking for up to 10 mins
```

- `model:` Name of the service and its container will inherit this in its name.
- `build: ./model:` Tells Docker to build the image for this service from the application files and Dockerfile in the `./model` directory.
- `environment:` Tells Ollama to use the model stored in the **MODEL** variable.
- `volumes:` Mount the **model_data** volume into the container at `/root/.ollama`
- `networks:` Connect to the **chatbot** network.
- `ports:` Mapp port `11434` on the docker host to `11434` on the container.
- `deploy.resources.limits.memory:` Allocate 8GB of memory to this service's container

- `healthcheck:` Run the command in the `test` field to determine if the application is up and serving the model. The actual command is much longer and I've trimmed it for the book. The other values tell Docker to repeat the test for up to 50 times or 10 minutes.

Stepping through the block, this service is called **model** and Compose will instruct Docker to build its image from the Dockerfile and application files in the `./model` directory. This service doesn't define a `command`, so it will start the app using the command specified in the Dockerfile's **CMD** or **ENTRYPOINT** property and use the **MODEL** environment variable to know which LLM to serve. If the LLM doesn't exist, Ollama will pull it from the Ollama library and store it in `/root/.ollama`, which is backed by the **model_data** volume. It will attach the container to the **chatbot** network and map port `11434` on the Docker host to `11434` on the container. Finally, it will allocate 8GB of RAM to the container and won't mark itself as healthy until the healthcheck command completes successfully. According to the healthcheck settings, it will wait 10 seconds after the container starts before attempting the first check. It will allow each check to run for five seconds and will keep retrying for up to 10 minutes or 50 more times.

If your Docker host has access to NVIDIA GPUs, you'll need to remove the comment characters (#) from the following five lines to ensure your model server gets access to your GPUs. You'll find the lines below the **model** service's `deploy` block.

```
deploy
  ...
    ...
      ...
    reservations:
      devices:
        - driver: nvidia
          count: all              <<---- Can set to an integer. E.g. 1, 2, 3
          capabilities: [gpu]
```

This configuration allocates `all` available GPUs to the **model** service. However, you can be more granular by setting the number of GPUs to 1, 2, 3, or however many GPUs your host has.

Finally, the app file for the model server is a shell script called `start.sh`. It starts Ollama as a background service, checks for a local copy of the LLM specified in the **MODEL** variable, and pulls it from the Ollama library if it doesn't have a local copy.

```
#!/bin/bash
set -e

/bin/ollama serve &           <<---- Start Ollama in the background and
OLLAMA_PID=$!                 <<---- acquire the Ollama PID
```

```
echo "Waiting for Ollama to start..."                     ------┐
for i in $(seq 1 30); do                                        |
    if curl -s http://localhost:11434/api/tags >/dev/null; then |
        break                                                   | Wait
    fi                                                          |
sleep 1                                                         |
done                                                       ------┘

if ! curl -s http://localhost:11434/api/tags | \          ------┐
    jq -e ".models[] | \                                        |
    select(.name == \"${MODEL}\")" > /dev/null; then            |
    echo "${MODEL} model not found. Pulling..."                 | Check
    /bin/ollama pull ${MODEL}                                   | pull
else                                                            |
    echo "${MODEL} model already exists"                        |
fi                                                         ------┘

wait $OLLAMA_PID                <<---- Wait for Ollama to start
```

Now that you understand how the Compose file works, let's
deploy the app.

Deploy the app

In this section, you'll use the Compose file to deploy the app.
You'll need a local copy of the book's GitHub repo, and you'll
need to run all commands from the `ai-compose` folder.

Run the following command to deploy the app. It may take a
few minutes if your internet connection is slow, as it pulls the

Ollama image, which is ~1.5GB, and then it pulls the Mistral model which is ~4GB.

```
$ docker compose up

[+] Running 7/15
 - backend Pulled
   <Snip>
 - frontend Pulled
   <Snip>
 - Service model                      Built
 - Network ai-compose_chatbot         Created
 - Volume "ai-compose_model_data"     Created
 - Container ai-compose-model-1       Created
 - Container ai-compose-backend-1     Created
 - Container ai-compose-frontend-1    Created
Attaching to backend-1, frontend-1, model-1
model-1     | mistral:latest model not found. Pulling...
pulling manifest
pulling manifest         <<---- This bit can take a long time on slow inte
<Snip>
backend-1   | INFO:     Application startup complete.
frontend-1  | [remix-serve] http://localhost:3000 (http://0.0.0.0:3000)
```

I've trimmed the output so it only shows the major steps. These include pulling the **backend** and **frontend** images, building the **model** image, creating the network volume and containers, pulling the Mistral LLM, and starting the app.

If you have a slow internet connection, the model server's healthcheck may fail with a **dependency failed to start:**

container ai-compose-model-1 is unhealthy error message. This usually means pulling the model took longer than the value specified in `healthcheck.start_period`. You can run the `docker compose up` command again, and if it fails again, you can increase the healthcheck start period in the Compose file so the healthcheck has longer to keep trying.

We'll look at the app in a minute, but let's talk about the `docker compose` command first.

Running `docker compose up` is the most common way to deploy a Compose app. It sends the Compose file in the local directory to Docker, and Docker runs the commands to deploy everything defined in the file. This includes building and pulling all the necessary images, creating all required networks and volumes, and starting all containers.

The command you executed didn't specify the name or location of the Compose file, so Docker assumed it was called `compose.yaml` in the local directory. However, you can use the `-f` flag to point to a Compose file with a different name in a different directory. For example, you'd run the following command if your Compose file was called `chatbot.yaml` in the `llm` subfolder.

```
$ docker compose -f ./llm/chatbot.yaml up
```

It's also common to use the **--detach** flag to bring the app up
in the background. We didn't use it this time because I wanted
you to see the log messages showing Docker pulling and
building images and Ollama pulling the model.

Use the app

The app is up and running, and you can start using it.

Point your browser to your Docker host on port `3000` and ask
your chatbot some questions. You can connect to
`localhost:3000` if you're running Docker Desktop.
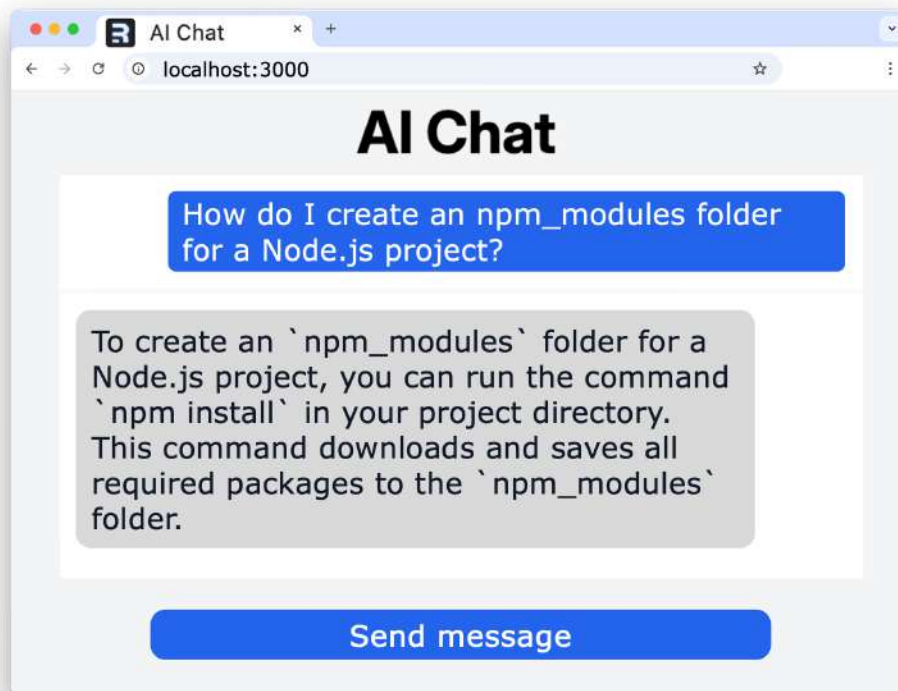
If you followed along and brought the application up in the foreground, you'll see the **frontend**, **backend**, and **model** servers exchanging messages in your other terminal window each time you issue a prompt. We'll see more of this later, but if you look closely, you'll see the **backend** posting requests to the **model**. I've formatted the JSON so it's easier to read.

```
backend-1    | INFO: 172.18.0.4:40688 - "POST /api/v1/chat/stream HTTP/1.1'
backend-1    | INFO: app.routes.generation:Sending request to Ollama:
{
    "model":"mistral",
    "messages":[
```

```
    {
        "role":"system",
        "content":"You are a coding assistant. Keep your responses focuse
        development, and technical topics. Be concise and provide practic
        when relevant. Limit explanations to 2-3 sentences unless code ex
    },
    {
        "role":"user",
        "content":"How do I create an npm_modules folder for a Node.js pr
    }
],
"stream":true,
"temperature":0.7
}
```

Congratulations. You've successfully deployed a multi-container
AI chatbot using Docker Compose!

Inspect the app

Now that you've deployed the app, you can run regular `docker`
commands to see the images, containers, networks, and
volumes — remember, Compose uses regular Docker
commands and API calls to build and manage the app.

Open a new terminal window and run the following command
to see the images Docker pulled and built for the app.

```
$ docker images
REPOSITORY                     TAG        IMAGE ID        CREATED         SIZ
ai-compose-model               latest     627b3f7223ed    1 min ago       7.2
```

```
nigelpoulton/ddd-book       ai-be      828623f6f219      2 mins ago      126
nigelpoulton/ddd-book       ai-fe      47e007c828dc      2 mins ago      63:
```

As per the instructions in the Compose file, Docker <u>pulled</u> the
frontend and **backend** images and used the Dockerfile in the
`model` directory to <u>build</u> the **model** image.

If you look at the **model** service's Dockerfile, you'll see it pulls
an Ollama Docker image, updates and installs packages, sets a
couple of environment variables, copies the application's
startup script and marks it as executable, sets some NVIDIA-
related variables for potential GPU use, documents the network
port, and lists the command to execute when the container
starts.

```
FROM ollama/ollama:0.5.4                          <<---- Base image
RUN apt-get update && apt-get install -y...        <<---- Update & install
ENV OLLAMA_HOST=0.0.0.0                            <<---- Ollama to listen
ENV PATH=/usr/local/sbin:/usr/local/bin:/usr...    <<---- Update system pa
COPY start.sh /usr/local/bin/start.sh              <<---- Copy app startup
RUN chmod +x /usr/local/bin/start.sh               <<---- Make startup scr
ENV LD_LIBRARY_PATH=/usr/local/nvidia/lib:/usr/local/nvidia/lib64   <<----
ENV NVIDIA_DRIVER_CAPABILITIES=compute,utility                      <<----
ENV NVIDIA_VISIBLE_DEVICES=all                                      <<----
EXPOSE 11434                                       <<---- Document network
ENTRYPOINT ["/bin/bash", "-c", "/usr/local/bin/start.sh"]      <<---- How
```

Run the following commands to list the containers, network, and volume.

```
$ docker ps
IMAGE                          COMMAND                   STATUS
nigelpoulton/ddd-book:ai-fe    "docker-entrypoint…"      Up 5 mins            a
nigelpoulton/ddd-book:ai-be    "uvicorn app.main:…"      Up 5 mins            a
ai-compose-model               "/bin/bash -c /usr…"      Up 5 mins (healthy)  a

$ docker network ls
NETWORK ID     NAME                    DRIVER    SCOPE
fe31e60beafa   ai-compose_chatbot      bridge    local
<Snip>

$ docker volume ls
DRIVER     VOLUME NAME
local      ai-compose_model_data
```

Docker created all of the app resources defined in the Compose file — three images, three containers, one network, and one volume. It also added the project name to the start of every resource name. The *project name* is the name of the build context directory, which in our example is `ai-compose`. Doing this makes it easy to locate and group resources that are part of the same Compose apps.

Compose also adds a number to the end of each container name in case a service has multiple replicas. For example, if the **frontend** service has three replicas, Compose will call them **ai-**

compose-frontend-1, **ai-compose-frontend-2**, and **ai-compose-frontend-3**.

At this point, the application is up and working.

Manage the app

In this section, you'll see how to stop, restart, delete, and get the status of Compose apps.

Check the status of the app.

```
$ docker compose ls
NAME                    STATUS              CONFIG FILES
ai-compose              running(3)          /Users/nigelpoulton/ddd-book/ai-co
```

The app is up and all three services are running.

Shut it down with the following command. If you started it in the foreground, you'll see messages requesting the app to gracefully quit.

```
$ docker compose down
  frontend-1 exited with code 0
  backend-1   | INFO:     Shutting down
  backend-1   | INFO:     Waiting for application shutdown.
  backend-1   | INFO:     Application shutdown complete.
  backend-1   | INFO:     Finished server process [7]
```

```
  backend-1   | INFO:    Stopping reloader process [1]
  backend-1 exited with code 0
  model-1     | [GIN] 2025/01/01 - 12:50:52 | 200 | 1.11225ms | 127.0.0.1
  model-1 exited with code 137
  <Snip>
  - Container ai-compose-frontend-1  Removed
  - Container ai-compose-backend-1   Removed
  - Container ai-compose-model-1     Removed
  - Network ai-compose_chatbot       Removed
```

The output shows Docker removing all three containers and the network. However, it doesn't mention removing the volume or images.

Run a `docker volumes ls` command to see if the volume still exists.

```
$ docker volume ls
DRIVER    VOLUME NAME
local     ai-compose_model_data
```

The volume still exists and will still have the Mistral LLM files. This is because Docker expects us to store important information on volumes, so it doesn't delete them when it deletes other application resources such as networks and containers.

Run a `docker images` command to see if the images still exist.

```
$ docker images
REPOSITORY              TAG        IMAGE ID        CREATED         SIZE
ai-compose-model        latest     627b3f7223ed    1 min ago       7.29
nigelpoulton/ddd-book   ai-be      828623f6f219    2 mins ago      126M
nigelpoulton/ddd-book   ai-fe      47e007c828dc    2 mins ago      633M
```

Keeping the images and the volume can make redeploying the app much quicker.

You're about to redeploy the app. However, this time, you'll configure Compose to *build* the frontend and backend images instead of pulling them or using the local copies already pulled.

Delete the frontend and backend images.

```
$ docker rmi nigelpoulton/ddd-book:ai-fe nigelpoulton/ddd-book:ai-be
Untagged: nigelpoulton/ddd-book:ai-fe
Deleted: sha256:47e007c8...fdce0962
Untagged: nigelpoulton/ddd-book:ai-be
Deleted: sha256:828623f6...4e30c2f6
```

Now edit the Compose file to build both images instead of pulling them. Adding a `#` to the start of a line in a Compose file tells Docker to ignore the line.

```
services:
    frontend:
#       image: nigelpoulton/ddd-book:ai-fe     <<---- Add # to start of this
```

```
        build: ./frontend                    <<---- Remove # from start of
        command: npm run start
        develop:
        <Snip>
      backend:
  #       image: nigelpoulton/ddd-book:ai-be  <<---- Add # to start of this
        build: ./backend                      <<---- Remove # from start of
        networks:
        <Snip>
```

Save your changes and run the following command to redeploy
the app with the `--detach` flag to start it in the background.

```
$ docker compose up --detach
<Snip>
[+] Running 7/7
 - Service backend                  Built
 - Service frontend                 Built
 - Network ai-compose_chatbot       Created
 - Synchronized File Shares
 - Container ai-compose-model-1     Healthy
 - Container ai-compose-backend-1   Started
 - Container ai-compose-frontend-1  Started
```

You can see that Docker followed your instructions and built
the backend and frontend images. It also shows the **model**
service showing as `Healthy` because it's configured with a
healthcheck in the Compose file. The other services show as
`Started` because they don't have healthchecks.

Check the current state of the app with the `docker compose ps` command.

```
$ docker compose ps
NAME                   COMMAND                 SERVICE   STATUS     PORT
ai-compose-backend-1   "uvicorn app.main:ap…"  backend   Up 2 mins  0.0.
ai-compose-frontend-1  "docker-entrypoint.s…"  frontend  Up 2 mins  0.0.
ai-compose-model-1     "/bin/bash -c /usr/l…"  model     Up 2 mins  0.0.
```

The output shows the containers, the commands they're executing, their current state, and the network ports they're listening on.

Run a `docker compose top` to list the processes inside each container. You should recognize some of the processes from the Compose file and Dockerfiles.

```
$ docker compose top
ai-compose-backend-1
UID    PID      PPID    ... CMD
1000   13296    13278   ... /usr/local/bin/python3.12 /usr/local/bin/uvicor
1000   13405    13296   ... /usr/local/bin/python3.12 -c from multiprocessi
1000   13406    13296   ... /usr/local/bin/python3.12 -c from multiprocessi

ai-compose-frontend-1
UID    PID      PPID    ... CMD
1000   13364    13345   ... npm run start
1000   13411    13364   ... node /app/node_modules/.bin/remix-serve ./build

ai-compose-model-1
```

```
UID    PID     PPID   ...  CMD
root   13196   13178  ...  /bin/bash /usr/local/bin/start.sh
root   13230   13196  ...  /bin/ollama serve
```

The PID numbers returned are the PID numbers as seen from the Docker host (not from within the containers).

Run the following commands to stop the app and recheck its status.

```
$ docker compose stop
[+] Stopping 3/3
 - Container ai-compose-frontend-1  Stopped
 - Container ai-compose-backend-1   Stopped
 - Container ai-compose-model-1     Stopped

$ docker compose ls
NAME                STATUS              CONFIG FILES

$ docker compose ps
NAME    COMMAND    SERVICE    STATUS    PORTS
```

The app is down, but Docker hasn't deleted any of its resources. Verify this by checking the containers still exist.

```
$ docker compose ps -a
NAME                    IMAGE                        SERVICE    STATUS
ai-compose-backend-1    nigelpoulton/ddd-book:ai-be  backend    Exited (0)
ai-compose-frontend-1   nigelpoulton/ddd-book:ai-fe  frontend   Exited (0)
ai-compose-model-1      ai-compose-model             model      Exited (1:
```

All three containers still exist in the `Exited` state. The **ai-compose-model-1** container shows a non-zero exit code because Docker only allowed it 10 seconds to shut down. It must not have completed a graceful shutdown during those ten seconds, forcing Docker to kill it.

Refresh your browser to confirm the chatbot is not running. The connection will be refused because the app is down and nothing is listening on port `3000`.

Restart the app with the `docker compose restart` command. It will restart quickly because the resources already exist.

```
$ docker compose restart
[+] Restarting 3/3
 - Container ai-compose-backend-1   Started
 - Container ai-compose-frontend-1  Started
 - Container ai-compose-model-1     Started
```

Verify the operation worked.

```
$ docker compose ls
NAME              STATUS        CONFIG FILES
ai-compose        running(3)    /Users/nigelpoulton/temp/ddd-book/ai-com
```

Go back to your browser and refresh the page again. It should be working.

Congratulations. You've deployed a multi-container AI chatbot app and used Docker Compose to perform some common management tasks.

Let's inspect the Ollama configuration before cleaning up.

Inspect the Ollama configuration

This section has nothing to do with Compose but is useful if you're interested in Ollama and running LLMs locally.

As per Figure 9.3 (which you've already seen), the application is a three-tier AI chatbot. The **frontend** service hosts a Remix web server that connects to a FastAPI server running on the **backend** service. The backend server integrates with the Ollama instance running on the **model** service.
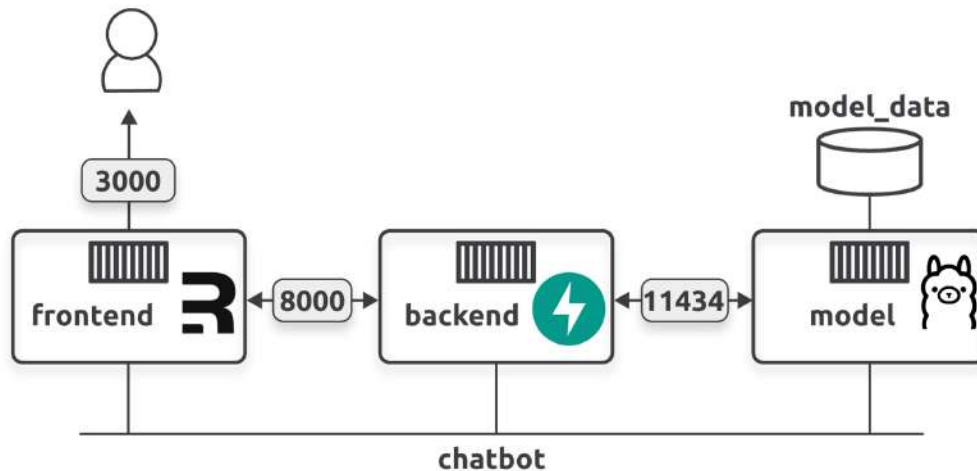
Figure9.3 - Sample app

Ollama is a popular open-source LLM runtime that lets you download and execute LLMs locally. It might be helpful to think of Ollama as *Docker for LLMs* as it lets you pull and push LLMs and run them locally so that nothing leaves your local network.

The **model** service's Dockerfile and startup script are in the `model` folder and you can inspect them to see how the image is built and how the app works. However, the high-level process involves pulling an Ollama container image from Docker Hub, copying in the startup script, and configuring the container to execute the startup script when it starts. The startup script starts Ollama and pulls the Mistral LLM. Once up and running, the model server exposes the Ollama API on port `11434`.

Open a shell session to the **model** server container. Your shell prompt will change to a "#" to indicate you're connected to the model server.

```
$ docker exec -it ai-compose-model-1 sh
#
```

Check that Ollama is installed.

```
# ollama --version
ollama version is 0.5.4-0-g2ddc32d-dirty
```

Check for an Ollama server process.

```
$ ps -ef
ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 13:26 ?        00:00:00 /bin/bash /usr/local/bin/s
root         6     1  0 13:26 ?        00:00:00 /bin/ollama serve
<Snip>
```

PID 6 shows an active Ollama server process. This means Ollama is running.

PID 1 is the startup script that started Ollama. It's the `start.sh` script from the `model` folder that was copied into the image and

set as the container's main app by the Dockerfile. Here's a snipped copy of the Dockerfile for reference.

```
FROM ollama/ollama:0.5.4
<Snip>
COPY start.sh /usr/local/bin/start.sh
<Snip>
ENTRYPOINT ["/bin/bash", "-c", "/usr/local/bin/start.sh"]
```

See if there are any local LLMs.

```
# ollama ls
NAME              ID            SIZE      MODIFIED
mistral:latest    f974a74358d6  4.1 GB    21 hours ago
```

The output shows a local copy of a Mistral LLM. This matches the `MODEL=${MODEL:-mistral:latest}` environment variable specified in the Compose file. The LLM is ~4GB and stored on the **model_data** volume mounted at `/root/.ollama` in the model server container.

Inspect the Mistral LLM's manifest file.

```
$ cat /root/.ollama/models/manifests/registry.ollama.ai/library/mistral/la
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
  "config": {
    "mediaType": "application/vnd.docker.container.image.v1+json",
```

```
      "digest": "sha256:42347cd80dc868877d2807869c0e9c90034392b2f1f001cae156
      "size": 485
    },
    "layers": [
      {
        "mediaType": "application/vnd.ollama.image.model",
        "digest": "sha256:ff82381e2bea77d91c1b824c7afb83f6fb73e9f7de9dda631b
        "size": 4113289152
      },
      <Snip>
    ]
  }
```

I've snipped the output to save space in the book. The full output lists all the layers that make up the model, including the image metadata, the model template, parameters, the license, and the model file.

You can see these layers by listing the contents of the `/root/.ollama/models/blobs` directory. The file names match the SHA digests from the LLM's manifest file.

```
$ ls -hl /root/.ollama/models/blobs
total 3.9G
-rw-r--r-- 1 root root  485 Dec 19 16:28 sha256-42347cd8...021e2e19   <<--
-rw-r--r-- 1 root root  12K Dec 19 16:28 sha256-43070e2d...5e1379c1   <<--
-rw-r--r-- 1 root root  801 Dec 19 16:28 sha256-491dfa50...71045621   <<--
-rw-r--r-- 1 root root   30 Dec 19 16:28 sha256-ed11eda7...4003d337   <<--
-rw-r--r-- 1 root root   4G Dec 19 16:28 sha256-ff82381e...64aa5435   <<--
```

Let's inspect the model's image metadata file to learn about the model. Feel free to inspect the license, template, and parameters files. Your SHAs might be different.

```
$ cat /root/.ollama/models/blobs/sha256-42347cd8...021e2e19 | jq
{
  "model_format": "gguf",
  "model_family": "llama",
  "model_families": [
    "llama"
  ],
  "model_type": "7.2B",
  "file_type": "Q4_0",
  "architecture": "amd64",
  "os": "linux",
  "rootfs": {
    "type": "layers",
    "diff_ids": [
      "sha256:ff82381e2bea77d91c1b824c7afb83f6fb73e9f7de9dda631bcdbca564aa
      "sha256:43070e2d4e532684de521b885f385d0841030efa2b1a20bafb76133a5e13
      "sha256:c43332387573e98fdfad4a606171279955b53d891ba2500552c2984a6566
      "sha256:ed11eda7790d05b49395598a42b155812b17e263214292f7b87d15e14003
    ]
  }
}
```

We already know this is a Mistral model, but the manifest also tells us it's in the GGUF binary format, it's a llama-style model, we have the 7.2 billion parameter version, and it uses the Q4 quantization strategy.

You can verify all of this with Ollama as well.

```
# ollama show mistral
  Model
    architecture        llama
    parameters          7.2B
    context length      32768
    embedding length    4096
    quantization        Q4_0

  Parameters
    stop    "[INST]"
    stop    "[/INST]"

  License
    Apache License
    Version 2.0, January 2004
```

See if your model is running.

```
# ollama ps
NAME              ID              SIZE      PROCESSOR      UNTIL
mistral:latest    f974a74358d6    5.9 GB    100% CPU       4 minutes from no
```

Don't worry if yours doesn't show a running model. This is because Ollama loads the model when you interact with it and then unloads it from memory after five minutes of inactivity. Just return to the chatbot's web interface, ask it a question, and run the `ollama ps` command again. Your chatbot might

respond slowly and show as ***Thinking*** while Ollama reloads it, but it will show as running in when you re-run the `ollama ps` command.

The output in the example shows `100% CPU` in the `PROCESSOR` column, indicating this is running on a system without GPUs. Your output will show GPUs if you're running on GPUs.

Type `exit` to log off the container and return to your local terminal.

In summary, the model server runs Ollama to execute a private chatbot based on the Mistral 7b LLM. You've seen how Docker built the image and how Ollama pulled the LLM and served it on the network. You also inspected the LLM's manifest and saw where Ollama keeps it in the filesystem. As it's running locally, nothing from your conversations with it leaves your corporate network.

Clean up

Run the following command to **stop and delete** the app.

Do not use the `--volumes` and `--rmi all` flags if you plan to use the chatbot again soon. This is because they'll force Docker

to delete the volume (and LLM) and images, making restarting the app a much longer process.

```
$ docker compose down --volumes --rmi all

  - Container ai-compose-frontend-1      Removed
  - Container ai-compose-backend-1       Removed
  - Container ai-compose-model-1         Removed
  - Network ai-compose_chatbot           Removed
  - Volume ai-compose_model_data         Removed
  - Image ai-compose-model:latest        Removed
  - Image nigelpoulton/ddd-book:ai-be    Removed
  - Image nigelpoulton/ddd-book:ai-fe    Removed
```

Multi-container apps with Compose – The commands

- `docker compose up` is the command to deploy a Compose app. It downloads or builds all images, containers, networks, and volumes the app needs. It expects you to call the Compose file `compose.yaml`, but you can use the `-f` flag to specify a custom filename. You'll normally start apps in the background with the `--detach` flag.
- `docker compose ls` gives a brief overview of all running Compose apps, including how many services are running and the location of the Compose file.

- `docker compose ps` lists the containers in the Compose app and shows their current state, the command each container is running, and network ports.
- `docker compose stop` stops all containers in a Compose app without deleting them. You can easily restart them with `docker compose restart`, and you shouldn't lose any data.
- `docker compose restart` restarts a stopped Compose app. If you make changes to the Compose file while it's stopped, these changes will **not** appear in the restarted app. Changes to the Compose file only get picked up when you redeploy the app.
- `docker compose down` stops and deletes a running Compose app. By default, it deletes containers and networks but not volumes and images.

Chapter Summary

In this chapter, you learned how to deploy and manage multi-container applications using Docker Compose.

Compose is part of the Docker Engine and has its own `docker compose` sub-command. It lets you define multi-container applications in declarative configuration files and deploy them with a single command.

Compose files define all the containers, networks, volumes, secrets, and other artifacts an application needs. You then use the `docker compose` command to post the Compose file to Docker, and Docker deploys everything defined in it.

Once you've deployed the app, you can manage its entire lifecycle using `docker compose` sub-commands.

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, the ports they publish, the networks and volumes they use, and much more. As such, it can help bridge the gap between development and operations teams. You should treat Compose files as code and manage them in version control systems.

10: Docker and Wasm

Wasm (WebAssembly) is driving the third wave of cloud computing, and Docker is evolving to take advantage.

We built the first wave on virtual machines (VMs), the second on containers, and we're building the third on Wasm. Each wave drives smaller, faster, and more secure workloads, and all three are working together to drive the future of cloud computing.
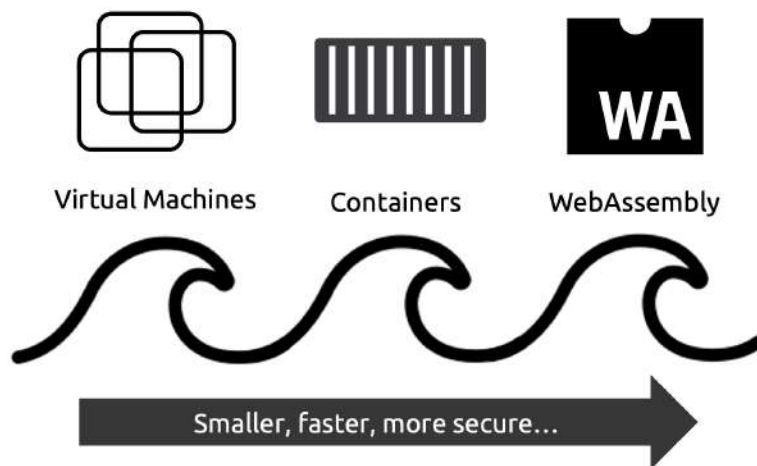
In this chapter, you'll write a simple Wasm application and use Docker to containerize and run it in a container. The goal is to introduce you to Wasm and show you how easy it is to work with Docker and Wasm together.

The terms *Wasm* and *WebAssembly* mean the same thing, and we'll use the term Wasm.

I've divided the chapter as follows:

- Pre-reqs
- Intro to Wasm
- Write a Wasm app
- Containerize a Wasm app
- Deploy a Wasm app

Pre-reqs

You'll need all of the following if you plan on following along:

- Docker Desktop **4.37**+ with Wasm enabled
- Rust 1.82+ with the Wasm target installed
- Spin 3.1+

At the time of writing, support for Wasm is a beta feature in Docker Desktop and doesn't work with Multipass Docker VMs. This may change in the future. It also means there's a higher risk of bugs. I've tested the examples in this chapter on Docker Desktop 4.37.0.

Configure Docker Desktop for Wasm

Open the Docker Desktop UI, click the Settings icon at the top right, and make sure **Use containerd for pulling and storing images** is selected on the **General** tab. Next, click the **Features in development** tab, select the **Enable Wasm** option and click the blue **Apply & restart** button.
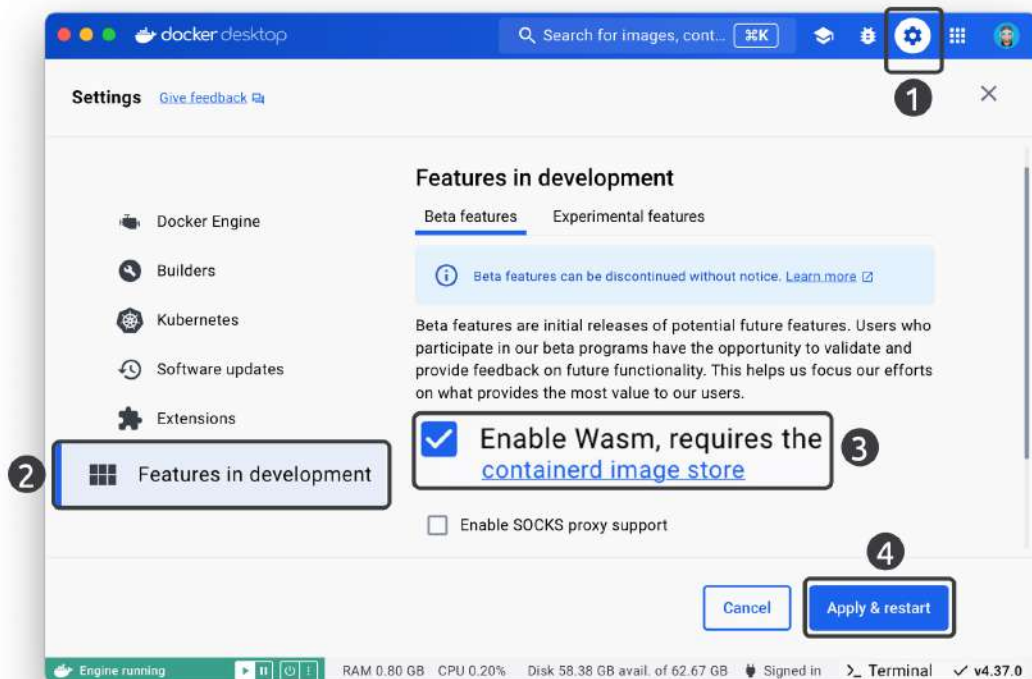
Figure 10.2 shows some of the settings.

Figure 10.2. Docker Desktop Wasm settings

Install Rust and configure for Wasm

Search the web for *how to install Rust* and follow the instructions for your platform.

Once you've installed Rust, run the following command to install the **wasm32-wasip1** target so that Rust can compile to Wasm.

```
$ rustup target add wasm32-wasip1
info: downloading component 'rust-std' for 'wasm32-wasip1'
info: installing component 'rust-std' for 'wasm32-wasip1'
```

Install Spin

Spin is a Wasm framework and runtime that makes building and running Wasm apps easy.

Search the web for *how to install Fermyon spin* and follow the instructions for your system.

Run the following command to verify the installation.

```
$ spin --version
spin 3.1.0 (1aa89da 2024-12-18)
```

You're ready to build and run Wasm apps on your local machine.

Intro to Wasm and Wasm containers

Wasm is a new type of application that is smaller, faster, and more portable than traditional Linux containers. However, traditional Linux containers can do a lot more than Wasm apps. For example, Wasm apps are currently great for AI workloads, serverless functions, plugins, and edge devices, but not so good for complex networking or heavy I/O.

However, Wasm is evolving fast and may become better at other workloads in the future.

Digging a little deeper…

As we're about to see, Wasm is a new virtual machine architecture that programming languages compile to. So, instead of compiling apps to *Linux on ARM* or *Linux on AMD*, you compile them to *Wasm* and they'll run on any system with a *Wasm runtime*. Fortunately, Docker Desktop ships with several Wasm runtimes.

Run the following command to see the list of Wasm runtimes installed as part of your Docker Desktop environment. The first time you run the command, it will download the image.

```
$ docker run --rm -i --privileged --pid=host jorgeprendes420/docker-deskto
io.containerd.wasmtime.v1
io.containerd.wws.v1
io.containerd.slight.v1
io.containerd.wasmer.v1
io.containerd.spin.v2
io.containerd.lunatic.v1
io.containerd.wasmedge.v1
```

My installation has seven Wasm runtimes, including the
`io.containerd.spin.v2` runtime we'll use in the examples.

These Wasm runtimes allow **containerd** to deploy and manage
Wasm containers. A *Wasm container* is a Wasm binary running
inside a minimal *scratch container* so that you can build, ship,
and run them with familiar Docker tools such as the `docker
run` command and Docker Hub.

Talk is cheap, though. Let's see it in action.

Write a Wasm app

In this step, you'll use **spin** to create a simple web server and
compile it as a Wasm app. In a later step, you'll build, share, and
run the app as a Wasm container.

Change into a new directory and then run the following
command to create a new Wasm app called *hello-world*.

Respond to the prompts as shown in the example.

```
$ spin new hello-world -t http-rust
Description: Wasm app
HTTP path: /hello
```

The command creates a new **hello-world** directory and scaffolds a simple Rust-based web app. Change into this directory and inspect the app files. If you don't have the `tree` command, you can run an `ls -l` for similar results.

```
$ cd hello-world

$ tree
.
├── Cargo.toml
├── spin.toml
└── src
    └── lib.rs
```

We're only interested in the `spin.toml` and `src/lib.rs` files.

Edit the `src/lib.rs` file and change the text inside the double quotes as shown in the following snippet. This configures the app to display **Docker loves Wasm**.

```
use spin_sdk::http::{IntoResponse, Request, Response};
<Snip>
    Ok(http::Response::builder()
```

```
        .status(200)
        .header("content-type", "text/plain")
        .body("Docker loves Wasm")?)          <<---- Change text inside
        .build())
  }
```

Once you've saved your changes, run a **spin build** command
to compile the app as a Wasm binary.

```
$ spin build
Building component hello-world with `cargo build --target wasm32-wasip1 --
<Snip>
Finished building all Spin components
```

If you look at the first line of the output, you'll see it's running a
more complex **cargo build** command that compiles the app as
a Wasm binary.

Run another **tree** command to see the Wasm binary.

```
$ tree
<Snip>
└── target
    └── wasm32-wasip1
        └── release
            └── hello_world.wasm
```

The output is much longer this time, and I've trimmed the example in the book so you only see the `hello_world.wasm` binary. This is the Wasm app, and it will run on any system with the **spin** Wasm runtime.

You'll containerize the app in the next section, but you should test it works before proceeding.

Run a `spin up` command to start the app using the local **spin** runtime you installed earlier.

```
$ spin up
Logging component stdio to ".spin/logs/"

Serving http://127.0.0.1:3000
Available Routes:
  hello-world: http://127.0.0.1:3000/hello
```

Point your browser to `http://127.0.0.1:3000/hello` and make sure the app works.

Figure 10.3. Wasm app running locally

Congratulations. You just built a simple web server, compiled it to Wasm, and executed it locally using **spin**! In the next section, you'll containerize it and run it in Docker.

Press `Ctrl-C` to kill the app.

Containerize a Wasm app

Docker Desktop lets you containerize Wasm apps so you can use familiar Docker tools to push and pull them to OCI registries and run them inside containers.

As always, you need a Dockerfile that tells Docker how to package the app as an image. Create a new file called **Dockerfile**

in your current directory and populate it with the following
three lines.

```
FROM scratch
COPY /target/wasm32-wasip1/release/hello_world.wasm .
COPY spin.toml .
```

The file references the **scratch** empty base image because
Wasm containers don't need a Linux OS.

The two **COPY** instructions copy the `hello_world.wasm` Wasm
app and the `spin.toml` file into the image.

If you look closely, you'll see that the `spin.toml` file expects the
Wasm app to be in the `target/wasm32-wasip1/release/`
directory. However, the second `COPY` instruction places it in the
root folder. This means we'll need to update the `spin.toml` file
so it knows where to find the app after the Dockerfile copies it
into the image's root directory.

Edit the `spin.toml` file and remove the leading path for the
source line as shown.

```
<Snip>
[component.hello-world]
source = "hello_world.wasm"        <<---- Remove any leading directories so
<Snip>
```

Save your changes.

Run the following command to containerize the Wasm app. Be sure to tag the image with your own Docker Hub username instead of mine.

```
$ docker build \
  --platform wasi/wasm \
  --provenance=false \
  -t nigelpoulton/ddd-book:wasm .
```

The `--platform wasi/wasm` flag sets the image as a Wasm image.

Some older versions of Docker have an older builder and will fail. If this happens, try running the same command, but change the first line to `docker buildx build \`.

List the images on your system.

```
$ docker images
REPOSITORY             TAG     IMAGE ID      CREATED          SIZE
nigelpoulton/ddd-book  wasm    7b55889f1006  28 seconds ago   104kB
```

See how the Wasm image looks like a regular image, just smaller.

You can push and pull the image to Docker Hub and other OCI registries as normal.

The following command pushes the image to one of my repos in Docker Hub. Be sure to tag the image with your own Docker username.

```
$ docker push nigelpoulton/ddd-book:wasm
The push refers to repository [docker.io/nigelpoulton/ddd-book]
301823195c36: Pushed
8966226af76a: Pushed
wasm: digest: sha256:7b55889f1006285ed6c394dcc7a56aca8955c107587b2216340e5
```

If you look at Docker Hub, you can see it's recognized it as a **wasi/wasm** image. You'll also see there's no vulnerability analysis data. This is because image scanning tools can't analyze Wasm images yet.
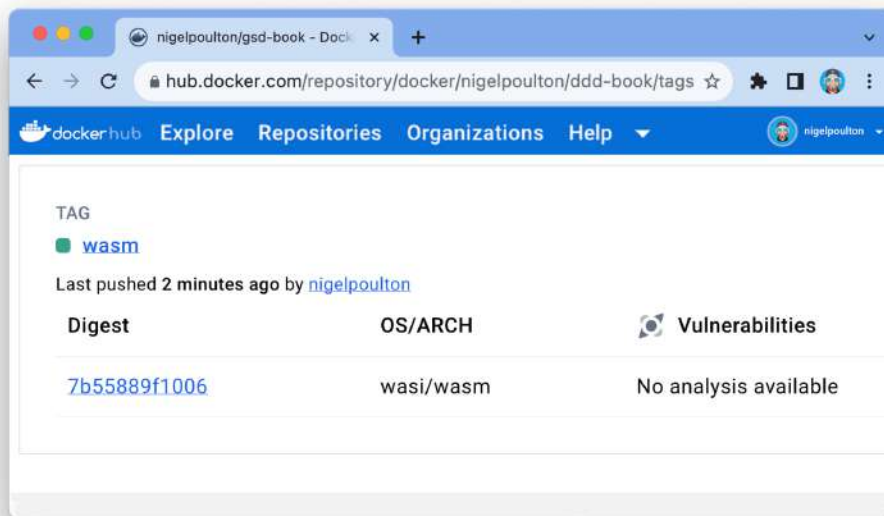
Figure 10.4. Wasm image on Docker Hub

Run a Wasm container

Now that you've packaged the Wasm app as an OCI image and pushed it to a registry, you can run it as a container.

The following command runs it in a new container called **wasm-ctr** and maps it to port `5556` on your Docker host. The `--runtime` flag makes sure Docker executes the container with the **spin** Wasm runtime. Older versions of Docker Desktop may not have the **spin** runtime and will fail.

```
$ docker run -d --name wasm-ctr \
  --runtime=io.containerd.spin.v2 \
  --platform=wasi/wasm \
```

```
  -p 5556:80 \
  nigelpoulton/ddd-book:wasm /
```

You can check it's running with a regular `docker ps` command.

Connect your browser to `http://localhost:5556/hello` to see the app.

Figure 10.5. Wasm app running in container

Congratulations, your Wasm app is running inside a Wasm container.

Clean up

Run the following commands to delete the container and the local image. Use your own image name when deleting the image.

```
$ docker rm wasm-ctr -f
wasm-ctr

$ docker rmi nigelpoulton/ddd-book:wasm
Untagged: nigelpoulton/ddd-book:wasm
Deleted: sha256:7b55889f1006285ed6c394dcc7a56aca8955c107587b2216340e59229
```

You'll still have a copy of the image on Docker Hub and the spin app in your local filesystem. Feel free to delete these as well.

Chapter summary

In this chapter, you containerized a Wasm app and ran it in a Wasm container.

Wasm is a new technology driving a new wave of cloud computing. Wasm apps are smaller, faster, more secure, and more portable than traditional Linux containers. However, they're not as flexible. For example, at the time of writing, Wasm apps aren't great for apps with heavy I/O requirements or complex networking. This will change quickly as the Wasm ecosystem is evolving fast.

Fortunately, Docker already works with Wasm, and Docker Desktop ships with a few popular Wasm runtimes. This means you can use industry-standard tools such as `docker build` and `docker run` to containerize and run Wasm apps. You can even push them to OCI registries such as Docker Hub.

11: Docker Swarm

Now that you know how to install Docker, pull images, and work with containers, the next logical step is to do it all at scale. That's where Docker Swarm comes into play.

I've split the chapter into the following main parts:

- Docker Swarm – The TLDR
- Swarm primer
- Build a secure Swarm
- Deploy and manage an app on swarm
- Service logs
- The commands

Docker Swarm – The TLDR

Docker Swarm is two things:

1. An enterprise-grade cluster of Docker nodes
2. An orchestrator of microservices apps

On the *clustering* front, Swarm groups one or more Docker nodes into 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 automatically

manages and rotates the swarm's certificates. You can even add and remove nodes non-disruptively. We call these clusters *"swarms"*.

On the *orchestration* front, Swarm makes deploying and managing complex microservices apps easy. You define applications declaratively in Compose files and use simple Docker commands to deploy them to the *swarm*. This is similar to deploying Compose apps with `docker compose` but is optimized for scheduling and scaling applications across a swarm.

In many ways, Docker Swarm is similar to Kubernetes — both are clusters that run and manage containerized applications. Kubernetes is more popular and has a more active community and ecosystem. However, Swarm is easier to use and can be a good choice for small-to-medium businesses and smaller application deployments. Learning Swarm also prepares you to learn and work with Kubernetes. In fact, if you plan on learning Kubernetes, you should check out my books, *Quick Start Kubernetes* and *The Kubernetes Book*.

Swarm primer

On the clustering front, a *swarm* is one or more Docker nodes that can be physical servers, VMs, cloud instances, Raspberry

Pi's, and more. The only requirement is that they all run Docker and can communicate over reliable networks.

> **Terminology:** *When referring to the Docker Swarm technology, we'll write Swarm with an uppercase "S". When referring to a swarm cluster, we'll use a lowercase "s".*

Every node in a swarm is either a *manager* or a *worker*:

- Managers run the *control plane* services that maintain the *state* of the cluster and manage applications
- Workers run user applications

Out of the box, Swarm runs user applications on managers **and** workers. However, you can configure your managers to focus on cluster management operations by forcing all user applications to run on worker nodes.

The swarm stores configuration and state data in an in-memory distributed database that replicates across all manager nodes. Fortunately, it's automatically configured and takes care of itself.

Swarm uses TLS to encrypt communications, authenticate nodes, and authorize roles (managers and workers). It also configures and performs automatic key rotation. And as with

the cluster store, all of this is automatically configured and takes care of itself.

Figure 11.1 shows a high-level view of a swarm with three managers, three workers, and a distributed cluster store replicated across all three managers.
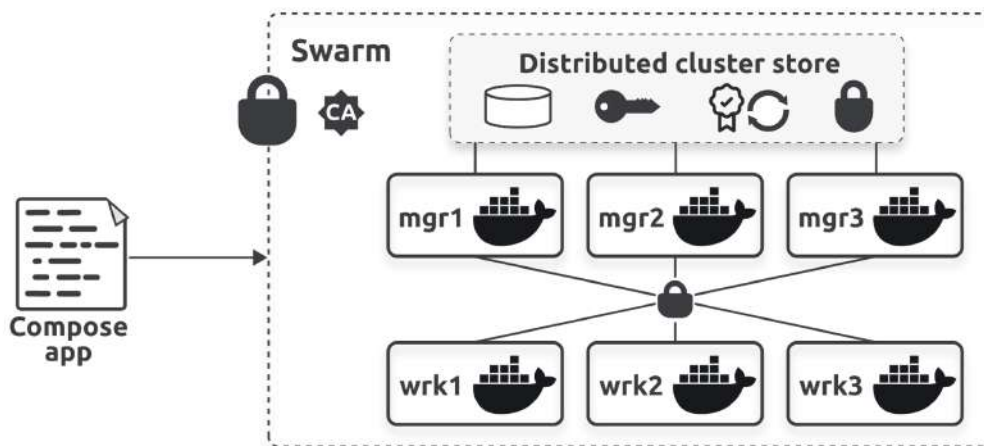

Figure 11.1 High-level swarm

On the orchestration front, Swarm makes it easy to deploy and manage containerized applications. It balances workloads across the cluster and adds cloud-native capabilities such as self-healing, scaling, rollouts, and rollbacks.

That's enough of a primer. Let's build a swarm.

Build a secure swarm

In this section, you'll build a secure swarm cluster. I'll build the cluster shown in Figure 11.2 with three *managers* and two *workers*. Your cluster can be different, but the following rules usually apply:

- Three or five managers are recommended for high-availability
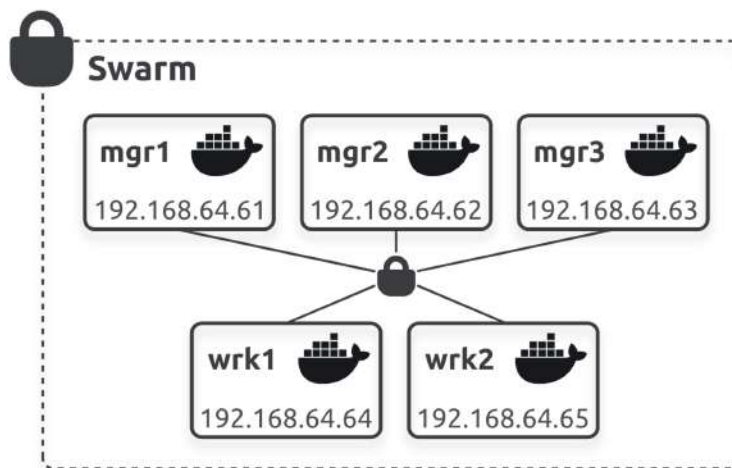- Enough workers to handle application requirements

Figure 11.2 - Sample lab environment

Pre-reqs

If you're following along, I recommend using Multipass to create five Docker VMs on your laptop or local machine. Multipass is free and easy to use, and I'll use it for the examples.

However, running five Multipass VMs requires a lot of system resources as each VM needs 2 CPUs, 40GB of disk space, and 4GB of memory. If your system can't handle this, you can create fewer managers and workers or build your lab in [Play with Docker](). Play with Docker (PWD) is a free cloud-based service that lets you create up to five nodes in a 4-hour Docker playground. Other Docker environments will also work. All you need is one or more Docker nodes that can communicate over a reliable network. You'll also find it easier if you configure name resolution so you can reference nodes by name rather than IP address.

I don't recommend following along on Docker Desktop as it only supports a single node, which isn't enough for some of the examples later in the next chapter.

Build your Docker nodes

This section uses Multipass to build the swarm in Figure 11.2.

See Chapter 3 or search the web for how to install multipass and follow the instructions to install it on your system.

Run the following five `multipass launch` commands to create five VMs running Docker. We'll refer to each VM as a *node*, and each will require 2 CPUs, 40GB of disk space, and 4GB of RAM. If

your machine doesn't have those resources, go to Play with Docker and build your lab there.

Name three of the nodes `mgr1`, `mgr2`, and `mgr3`, and name the other two `wrk1` and `wrk2`. Yours will get different IPs, but that doesn't matter. It can take a couple of minutes to create each VM.

```
$ multipass launch docker --name mgr1
Launched: mgr1

$ multipass launch docker --name mgr2
Launched: mgr2

$ multipass launch docker --name mgr3
Launched: mgr3

$ multipass launch docker --name wrk1
Launched: wrk1

$ multipass launch docker --name wrk2
Launched: wrk2
```

Run the following command and note the node IP addresses. Yours will have different addresses, and you're only interested in the `192.168.x.x` address for each.

```
$ multipass ls
Name            State          IPv4               Image
mgr1            Running        192.168.64.61      Ubuntu 24.04 LTS
```

```
                                        172.17.0.1
    mgr2         Running               192.168.64.62        Ubuntu 24.04 LTS
                                        172.17.0.1
    mgr3         Running               192.168.64.63        Ubuntu 24.04 LTS
                                        172.17.0.1
    wrk1         Running               192.168.64.64        Ubuntu 24.04 LTS
                                        172.17.0.1
    wrk2         Running               192.168.64.65        Ubuntu 24.04 LTS
                                        172.17.0.1
```

Once you've built the nodes, you can log on to them with the
`multipass shell` command and log out by typing `exit`.

Log on to one of the nodes and check that Docker is installed
and running.

```
$ multipass shell mgr1

$ docker --version
Docker version 27.3.1, build ce12230
```

Type `exit` to log out of the node and return to your local shell.

Once you've confirmed your nodes are working, you can move
to the next section and build a swarm.

Initializing a new swarm

The process of building a swarm is called *initializing a swarm,*
and this is the high-level process:

1. Initialize the first manager
2. Join additional managers
3. Join workers.

All Docker nodes start in *single-engine* mode and can only run regular containers. Joining them to a swarm switches into *swarm mode* and enables them to run advanced containers called *swarm services*. More on *services* later.

When you run a `docker swarm init` command on a Docker node that's currently in *single-engine mode*, Docker switches it into *swarm mode*, creates a new *swarm*, and makes the node the first *manager* of the swarm. You can then add more nodes as managers and workers, and these nodes also get switched into *swarm mode.*

You're about to complete all of the following:

1. Initialize a new swarm from **mgr1**
2. Join **mgr2** and **mgr3** as additional managers
3. Join **wrk1** and **wrk2** as worker nodes

After completing the procedure, all five nodes will be in swarm mode and part of the same swarm.

The examples will use the names and IPs from Figure 11.2. Yours will be different, and you should use your own.

1. Log on to **mgr1** and initialize a new swarm. Be sure to use the private IP address of *your* **mgr1** node.

```
$ docker swarm init \
  --advertise-addr 192.168.64.61:2377 \
  --listen-addr 192.168.64.61:2377

Swarm initialized: current node (d21lyz...c79qzkx) is now a manager.
<Snip>
```

Let's step through the different parts of the command:

1. `docker swarm init` tells Docker to initialize a new swarm with this node as the first manager.
2. The `--advertise-addr` flag is optional and tells Docker which of the node's IP addresses to advertise as the swarm API endpoint. It's usually one of the node's IP addresses but can also be an external load balancer.
3. The `--listen-addr` flag tells Docker which of the node's interfaces to accept swarm traffic on. It defaults to the same value as `--advertise-addr` if you don't specify it. However, if `--advertise-addr` is a load balancer, you must use `--listen-addr` to specify a local IP.

I recommend specifying both flags in important production environments. However, for most lab environments, you can just run a `docker swarm init` without any flags.

You can also specify a custom port for the advertise and listen addresses, but using the default `2377` is common.

2. List the nodes in the swarm.

```
$ docker node ls
ID              HOSTNAME    STATUS  AVAILABILITY  MANAGER STATUS
d21...qzkx *   mgr1        Ready   Active        Leader
```

mgr1 is the only node in the swarm and is listed as the *Leader*. We'll talk about leaders later.

3. Run the following command from **mgr1** to see the commands and tokens needed to add new workers and managers.

```
$ docker swarm join-token worker
To add a manager to this swarm, run the following command:
   docker swarm join --token SWMTKN-1-0uahebax...c87tu8dx2c 192.168.64.

$ docker swarm join-token manager
To add a manager to this swarm, run the following command:
   docker swarm join --token SWMTKN-1-0uahebax...ue4hv6ps3p 192.168.64.
```

Notice how the outputs are identical except for the join tokens (`SWMTKN...`). You should keep your join tokens

secure, as they're all that's required to join other nodes to your swarm!

4. Log on to **mgr1** and use the correct command and token to join it as a manager node. I've added the optional `advertise-addr` and `listen-addr` flags in the example so you know how to use them. You can leave them off, but if you do use them, be sure to use the correct IPs for your environment.

```
$ docker swarm join \
    --token SWMTKN-1-0uahebax...ue4hv6ps3p \
    10.0.0.1:2377 \
    --advertise-addr 192.168.64.62:2377 \
    --listen-addr 192.168.64.62:2377

This node joined a swarm as a manager.
```

If you experience joining problems, make sure the following network ports are open between all nodes:

1. `2377/tcp:` for secure client-to-swarm communication
2. `7946/tcp and udp:` for control plane gossip
3. `4789/udp:` for VXLAN-based overlay networks

5. Repeat the previous step on **mgr3** so that you have two workers. If you specify the `--advertise-addr` and `--listen-addr` flags, make sure you use **mgr3's** IP addresses.

6. Log on to **wrk1** and **wrk2** and join them as workers using the *worker token* from step 3. Again, you don't have to use the `--advertise-addr` and `--listen-addr` flags, but if you do, be sure to use the correct IPs for your environment.

```
$ docker swarm join \
    --token SWMTKN-1-0uahebax...c87tu8dx2c \
    10.0.0.1:2377 \
    --advertise-addr 192.168.64.64:2377 \
    --listen-addr 192.168.64.64:2377

This node joined a swarm as a worker.
```

7. Log on to any of your manager nodes and run a `docker node ls` to see all the nodes in your swarm.

```
$ docker node ls
ID                  HOSTNAME    STATUS   AVAILABILITY    MANAGER STATUS
0g4rl...babl8 *     mgr2        Ready    Active          Reachable
2xlti...l0nyp       mgr3        Ready    Active          Reachable
d21ly...9qzkx       mgr1        Ready    Active          Leader
8yv0b...wmr67       wrk1        Ready    Active
e62gf...l5wt6       wrk2        Ready    Active
```

If you look closely at the `MANAGER STATUS` column, you'll see all three managers are showing as either *Reachable* or *Leader*. The nodes with nothing in this column are worker nodes. The

manager with the asterisk after its name is the one you're executing commands from.

Congratulations. You've created a five-node swarm with three managers and two workers. The Docker Engine on each node is now operating in swarm mode, and the swarm is secured with TLS.

Swarm manager high availability

Swarm clusters are highly available (HA), meaning one or more managers can fail, and the swarm will keep running.

Technically speaking, Swarm implements *active/passive multi-manager HA*. This means a swarm with three managers will have one active manager and two passive managers. We call the active manager the *"leader"* and the passive managers *"followers"*. The leader is the only manager that can update the swarm configuration, but if it fails, one of the followers is elected as the new leader and the swarm keeps running without any service interruption. If you send commands to a follower, it proxies them to the leader.

Figure 11.3 shows you issuing a command to a follower manager requesting an update to the swarm. Step 1 shows you issuing the command. Step 2 shows the follower manager

receiving and proxying the command to the leader. Step 3
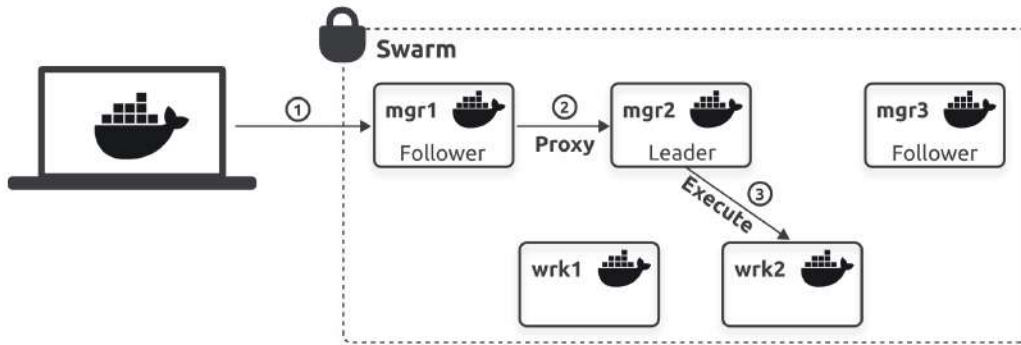shows the leader executing it.

Figure 11.3

Leader and *follower* is *Raft* terminology, and we use it because
Swarm implements the <u>Raft consensus algorithm</u> to maintain a
consistent cluster state across multiple highly-available
managers.

The following good practices apply when it comes to manager
HA:

1. Always deploy an odd number of managers
2. Don't deploy too many managers (3 or 5 is usually enough)
3. Spread managers across availability zones

Consider the two examples shown in Figure 11.4. The swarm on the left has an even number of managers, and a network incident has created a network partition with two managers on either side. We call this a *split brain* because neither side can be sure it has a majority and the cluster goes into read-only mode. When this happens, your **apps** continue working as normal but you can't make changes to them or to the cluster. However, the swarm on the right has an ***odd number of managers*** and remains fully operational in read-write mode because the two managers on the right side of the network partition know they have a majority (quorum). So, even though the swarm on the right has fewer managers than the one on the left, it has better availability.
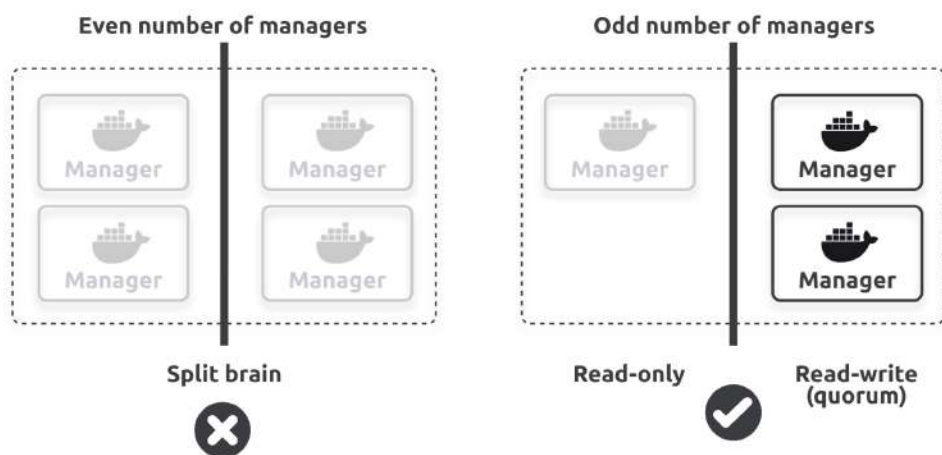

Figure 11.4 - Swarm high availability

As with all consensus algorithms, more participants means longer times to achieve consensus. It's like choosing where to eat — it's always quicker and easier for three people to decide than it is for 33! With this in mind, having three or five managers for HA is considered a best practice. Seven might work, but three or five is best for most swarms.

A final word of caution regarding manager HA. While you should spread your managers across availability zones, they must be connected by fast and reliable networks. This means you should think carefully and do lots of testing before implementing multi-cloud HA.

Built-in Swarm security

Swarm ships with a lot of security features, such as a built-in certificate authority (CA), mutual TLS, an encrypted cluster store, encrypted networks, cryptographic node IDs and join tokens, and more. Fortunately, Swarm automatically configures them with sensible defaults.

Locking a Swarm

Despite all of Swarm's security features, restarting an old manager or restoring an old backup can compromise your cluster. For example, a malicious actor with access to an old

manager node may be able to re-join it to the swarm and gain unwanted access.

Fortunately, you can use Swarm's *autolock* feature to force restarted managers to present a key before being admitted back into the swarm.

You can autolock new swarms at build time by passing the `--autolock` flag to the `docker swarm init` command. If you already have a swarm, you can autolock it with the `docker swarm update` command.

Run the following command from one of your swarm managers to lock your existing swarm cluster:

```
$ docker swarm update --autolock=true
Swarm updated.
To unlock a swarm manager after it restarts, run the `docker swarm unlock`
provide the following key:

    SWMKEY-1-XDeU3XC75Ku7rvGXixJ0V7evhDJGvIAvq0D8VuEAEaw

Please remember to store this key in a password manager...
```

While it's possible to run a `docker swarm unlock-key` command on any manager to view your unlock key, you should

also keep a secure copy outside of your cluster in case you're ever locked out of your swarm.

Log on to one of your managers, restart it, and then run a `docker node ls` command from it.

```
$ docker node ls
Error response from daemon: Swarm is encrypted and needs to be unlocked be
```

Docker has restarted, but the manager hasn't been allowed to re-join the swarm. You can prove this by running a `docker node ls` from one of your other managers and the restarted manager will be listed as `down` and `unreachable`.

Run the `docker swarm unlock` command from the restarted manager to re-admit it. You'll be prompted to provide the unlock key.

```
$ docker swarm unlock
Please enter unlock key: <enter your key>
```

Docker will re-admit the node to the swarm, and it will show as `ready` and `reachable` in future commands.

You should lock your production swarms and protect the unlock keys.

Dedicated manager nodes

By default, Swarm runs user applications on workers **and** managers. However, you may want to configure your production swarm clusters to only run user applications on workers. This allows managers to focus exclusively on control-plane duties.

Run the following command on any manager to prevent it from running user applications. You'll need to use the names of your managers, and you'll need to run it for all managers if you want to force user applications to run only on workers.

```
$ docker node update --availability drain mgr1
```

You'll see the impact of this in later steps when you deploy services with multiple replicas.

Now that you've built a swarm and understand the concepts of *leaders* and *manager HA*, the next chapter focuses on deploying multi-container applications to a swarm.

Docker Swarm – The Commands

- `docker swarm init` creates a new swarm. The node you run the command on becomes the first manager, and that node's Docker Engine switches to swarm mode.
- `docker swarm join-token` reveals the commands and tokens required to join workers and managers to the swarm. You add `manager` to the command to get the manager join token and `worker` to get the worker token. Be sure to keep your join tokens secure!
- `docker node ls` lists managers and workers and shows which manager is the leader.
- `docker swarm update --autolock=true` locks a swarm and requires restarted managers to present the cluster unlock code before being re-admitted to the cluster
- `docker swarm unlock` allows a restarted manager to present the cluster unlock code and re-join a locked swarm after a reboot
- `docker node update --availability drain <node>`

Chapter summary

Swarm is Docker's native technology for managing clusters of Docker nodes and orchestrating microservices applications. It is similar to Kubernetes but less advanced and easier to use.

The enterprise-grade clustering component offers a wealth of security and HA features that are automatically configured and extremely simple to modify.

You'll learn about Swarm orchestration in the next chapter.

12: Deploying apps to Swarm

Deploying and managing cloud-native microservices applications at scale is hard. Fortunately, Docker Swarm makes it easy.

I've split this chapter as follows:

- Deploying apps with Docker Stacks – The TLDR
- Build a Swarm lab
- The sample app
- Deploy the app
- Manage the app

Deploying apps with Docker Stacks – The TLDR

Docker Stacks combine the benefits of Compose and Swarm to create a platform for easy deployment and management of complex multi-container apps on secure, highly available infrastructure.

You build a swarm, define your apps in Compose files, and deploy and manage them with the `docker stack` command.

If you've been following along, you'll already know Compose and Swarm, so you'll find Docker Stacks easy.

Build a Swarm lab

This section shows you how to build a three-node swarm lab with a single manager and two workers. You can skip to the next section if you already have a swarm.

You can build the lab in Multipass, Play with Docker, or just about any Docker environment. However, I don't recommend using Docker Desktop, as the examples work best on multi-node swarms, and Docker Desktop limits you to a single node.

Run the following command from the node that will be your swarm manager.

```
$ docker swarm init
Swarm initialized: current node (lhma...w4nn) is now a manager.
<Snip>
```

Copy the **docker swarm join** command shown in the output and paste it into the two nodes that will be workers.

Worker 1

```
$ docker swarm join --token SWMTKN-1-2hl6...-...3lqg 172.31.40.192:2377
This node joined a swarm as a worker.
```

Worker 2

```
$ docker swarm join --token SWMTKN-1-2hl6...-...3lqg 172.31.40.192:2377
This node joined a swarm as a worker.
```

Run the following command from your manager node to confirm the swarm is initialized with one manager and two workers.

```
$ docker node ls
ID              HOSTNAME    STATUS      AVAILABILITY    MANAGER STATUS
1hm...4nn *     mgr1        Ready       Active          Leader
b74...gz3       wrk1        Ready       Active
o9x...um8       wrk2        Ready       Active
```

Now that you have a swarm lab, let's look at the sample app.

The sample app

Figure 12.1 shows the application we'll use for the rest of the chapter. It's a multi-container microservices application with:

- Two services (`web-fe` and `redis`)
- An encrypted overlay network (`counter-net`)
- A volume (`counter-vol`)
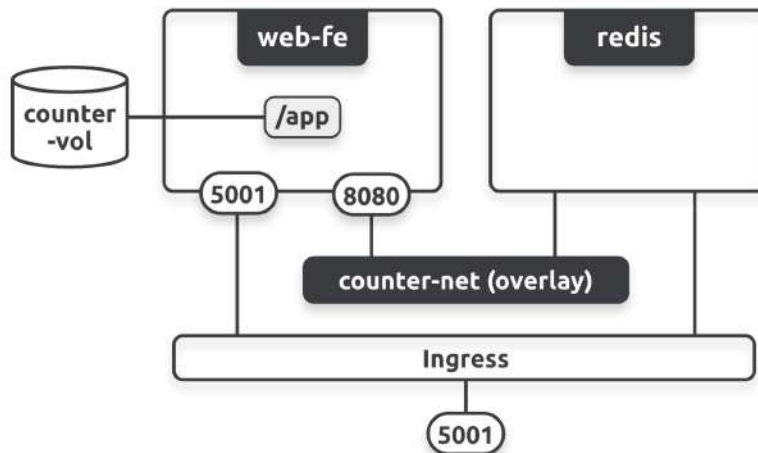- A published port (`5001`)

Figure 12.1

Terminology: *Throughout this chapter, we'll use the term service to refer to the Docker service object that manages one or more identical containers on a swarm. We'll use the terms container and replica interchangably. We'll also refer to the Compose file as the stack file, and sometimes we'll refer to the application as the stack.*

Log on to a swarm manager and clone the book's GitHub repo. You'll need the repo on your swarm manager for the examples to work.

```
$ 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.
```

Change into the `ddd-book/swarm-app` directory.

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

We'll focus on the Compose file, but feel free to inspect the application in the `app` folder.

If you look at the Compose file, you'll see it has three top-level keys.

- networks
- volumes
- services

Networks is where you define the networks the app needs, **volumes** defines the volumes it needs, and **services** defines the microservices that make up the app. The file is a simple example of *infrastructure as code* — the application and its infrastructure are defined in a configuration file that you use to deploy and manage them.

If you expand each top-level key, you'll see how things map to Figure 12.1 with a network called **counter-net**, a volume called **counter-vol**, and two services called **web-fe** and **redis**.

```
networks:
  counter-net:
volumes:
  counter-vol:
services:
  web-fe:
  redis:
```

Let's look at each section of the *stack file.*

Looking closer at the stack file

Stack files are identical to Compose files with a few differences at run-time. For example, Docker Compose lets you build images at run-time, but Docker Stacks don't.

Let's look at the networking elements defined in our stack file.

Networks and networking

One of the first things Docker does when you deploy an app from a stack file is create the networks defined under the `networks` key. If they already exist, Docker doesn't need to do anything. But if they don't exist, it creates them.

This app defines a single encrypted overlay network called
counter-net.

```
networks:
  counter-net:            <<---- Network name
    driver: overlay       <<---- Driver/type of network
    driver_opts:
      encrypted: 'yes'    <<---- Encrypt the data plane
```

It needs to be an overlay network so it can span all nodes in the
swarm.

Encrypting the data plane will keep the application traffic
private, but it 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 your own testing against your own applications.

The stack also publishes the **web-fe** service on port `5001` on the
swarm-wide *ingress network* and maps it back to port `8080` on
all service replicas. This allows external clients to reach the
web-fe service by hitting any swarm node on port `5001`.

```
services:
  web-fe:
   <Snip>
     ports:
```

```
       - target: 8080          <<---- Service replicas listen on this port
         published: 5001        <<---- and are published externally on this por
```

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 `web-fe` replicas. This means all reads and writes to the `/app` folder will be serviced by the volume.

```
volumes:
  counter-vol:                    <<---- Volume name

services:
  web-fe:
    <Snip>
    volumes:
      - type: volume
        source: counter-vol    <<---- Mount the "counter-vol" volume to
        target: /app           <<---- "/app" in all web-fe service replica
```

In this example, the volume will be a directory on the Docker host's filesystem but could easily be on an external storage system.

Let's look at the services.

Services

Services are where most of the action happens. This application defines two, so let's look at each in turn.

The web-fe service

As you can see, the **web-fe** service defines an image, an app, the desired number of replicas, rules for updating the app, a restart policy, which network to connect to, which ports to publish, and how to mount a volume. That's a lot to take in, so I've annotated it in the book. Take a minute to read the file and annotations.

```
web-fe:
  image: nigelpoulton/ddd-book:swarm-app   <<---- Create all replicas with
  command: python app.py                   <<---- Run this command when ea
  deploy:
    replicas: 4                    <<---- Deploy 4 replicas
    update_config:                 ------¬ When you perform an update,
      parallelism: 2                     | update 2 replicas at a time,
      delay: 10s                         | wait 10 seconds in between each
      failure_action: rollback     ------┘ and perform a rollback if you er
    placement:                     ------¬
      constraints:                       | Only run replicas on worker node
        - 'node.role == worker'    ------┘
    restart_policy:                ------¬ Only restart replicas if
      condition: on-failure              | they've failed (non-zero return
      delay: 5s                          | wait five seconds between each r
      max_attempts: 3                    | only attempt three restarts,
      window: 120s                 ------┘ and give up trying after two mir
  networks:
    - counter-net                  <<---- Connect replicas to the "counter-
  ports:
```

```
   - published: 5001              ------¬ Publish the service externally c
     target: 8080                 ------┘ map traffic to each replica on p
  volumes:
   - type: volume
     source: counter-vol          ------¬ Mount the "counter-vol" volume t
     target: /app                 ------┘ "/app" in each service replica
```

The `image` key is the only mandatory key and tells Docker which image to use when creating service replicas (containers). Swarm stacks don't support building images at deployment time, so the image must already exist. Docker is also *opinionated* and assumes you want to pull images from Docker Hub. However, you can add the registry's DNS name before the image name if you want to use a third-party registry. For example, adding `ghcr.io` before an image name will pull it from GitHub Container Registry.

The `command` key tells Docker how to start the app in each replica. Our example tells it to execute `python app.py` in all replicas.

The `deploy.replicas` key tells Swarm to deploy and manage four service replicas (identical containers).

```
  web-fe:
    deploy:
      replicas: 4
```

If you need to change the number of replicas after you've deployed the service, you update the value of `deploy.replicas` in the stack file and then redeploy the stack. We'll see this later.

The `deploy.update_config` block tells Docker how to perform rollouts.

```
web-fe:
  deploy:
    update_config:
      parallelism: 2
      delay: 10s
      failure_action: rollback
```

This application tells Docker to update two replicas at a time, wait 10 seconds between each set of two, and to perform a rollback if it encounters any problems.

Performing a rollback replaces the new replicas with fresh copies of the old version. The default value for `failure_action` is `pause`, which halts the update and may result in some replicas running the old version and some running the new. The other option is `continue`.

The `deploy.placement` block forces all replicas onto worker nodes. You'll need to delete this section if you're running a

single-node swarm with one manager and zero workers.

```
web-fe:
  deploy:
    placement:
      constraints:
        - 'node.role == worker'
```

The **`deploy.restart_policy`** block tells Docker what to do when it encounters a failure during a rollout.

```
web-fe:
  deploy:
    restart_policy:
      condition: on-failure
      max_attempts: 3
      delay: 5s
      window: 120s
```

This app tells Docker to restart replicas if they fail, to attempt a maximum of 3 restarts, to wait 5 seconds between each restart attempt, and to wait up to 120 seconds to decide if the restart worked.

The **`networks`** key tells Docker to attach all replicas to the **`counter-net`** network.

```
web-fe:
  networks:
```

```
      - counter-net
```

The **ports** block publishes the app on port 5001 on the ingress network and port 8080 on the **counter-net** network. This means external traffic can hit any swarm node on 5001 and get redirected to the service replicas on 8080.

```
web-fe:
  ports:
    - published: 5001
      target: 8080
```

Finally, the **volumes** block mounts the **counter-vol** volume into each service replica's **/app** directory.

```
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 previously mentioned, Compose files are a great source of application documentation. For example, we can read this file and know the application has two services, an encrypted network, and a volume. We can also figure out how the services communicate, how Docker publishes them outside of the swarm, how it will deploy them, update them, and when it will attempt restarts.

Let's deploy and manage the app.

Deploy the app

If you plan on following this section, you'll need a swarm and a copy of the book's GitHub repo on your swarm manager. The examples use a three-node swarm with a single manager called **mgr1** and two workers called **wrk1** and **wrk2**.

You deploy Stacks with the `docker stack deploy` command, and in its basic form it accepts two arguments:

- The name of the stack file (Compose file)
- The name of the stack (app)

We'll use the `compose.yaml` file in the `ddd-book/swarm-app` folder, and we'll call the app **ddd**.

Run all of the following commands from the `swarm-app` directory on a swarm manager.

Deploy the stack.

```
$ docker stack deploy -c compose.yaml ddd
Creating network ddd_counter-net
Creating volume ddd_counter-vol
Creating service ddd_web-fe
Creating service ddd_redis
```

A few things to note from the output.

Docker creates the networks and volumes before services. This is because services depend on networks and volumes and may fail to start if they don't exist.

Docker also prefixes the name of the stack to every resource. We called our stack **ddd**, causing Docker to prefix every resource name with **ddd** followed by an underscore:

- `ddd_counter-net`
- `ddd_counter-vol`
- `ddd_web-fe`
- `ddd_redis`

When you executed the `docker stack deploy` command, your Docker client sent the command and the `compose.yaml` file to a swarm manager, and the *leader manager* initiated the work to deploy everything defined in the file. In this case, Docker created a new network called **ddd_counter-net**, a new volume called **ddd_counter-vol**, and deployed the required number of replicas for the **web-fe** and **redis** services.

However, it doesn't end there. Swarm records all of this in the cluster store as your *desired state* and runs a background loop that constantly compares the *observed state* of the cluster with the *desired state* saved in the cluster store. When the two states match, the world is a happy place, and no further action is needed. When they don't match, Swarm takes action to bring the *observed state* into line with *desired state*.

For example, if a *worker* hosting two of the four **web-fe** replicas fails, the *observed state* of the cluster will drop from four **web-fe** replicas to two and no longer match the *desired state.* As soon as the swarm observes the discrepency, it will start two new

replicas to bring the *observed state* back in line with *desired state*. We call this *reconciliation* or *self-healing*, and it's a core tenet of cloud-native applications.

You'll soon learn that desired state and reconciliation are also at the heart of scaling, rollouts, and rollbacks.

Inspect the app

You can check the status of services, volumes, and networks with their usual commands, but the `docker stack` command also has a couple of options for checking the status of stacks:

- `docker stack ls` prints a list of running stacks and how many services they have
- `docker stack ps <stack-name>` gives more detailed information about a specific stack and its services and replicas

Let's look at both.

```
$ docker stack ls
NAME       SERVICES
ddd        2

$ docker stack ps ddd
NAME            IMAGE                           NODE    DESIRED STATE    CL
ddd_redis.1     redis:alpine                    mgr1    Running          Ru
ddd_web-fe.1    nigelpoulton/ddd-book:swarm-app wrk1    Running          Ru
```

```
  ddd_web-fe.2     nigelpoulton/ddd-book:swarm-app   wrk2   Running      Ru
  ddd_web-fe.3     nigelpoulton/ddd-book:swarm-app   wrk2   Running      Ru
  ddd_web-fe.4     nigelpoulton/ddd-book:swarm-app   wrk1   Running      Ru
```

The `docker stack ps` command is great for troubleshooting services. You can see the image each replica is based on, which nodes replicas are running on, as well as *desired state* and *current state*. If you look closely, you'll notice that Swarm has honored the `deploy.placement.constraints` property and ensured **web-fe** replicas only run on worker nodes. It has also evenly balanced the four **web-fe** replicas across both workers. However, the **redis** replica is scheduled on the manager node because it has no such placement constraint.

The following output shows two failed attempts to start `web-fe` replicas on the `wrk2` node.

```
$ docker stack ps ddd
NAME            NODE      DESIRED     CURRENT     ERROR
web-fe.1        wrk2      Shutdown    Failed      "task: non-zero exit (1)"
\_web-fe.1      wrk2      Shutdown    Failed      "task: non-zero exit (1)"
```

You can also use the `docker service logs` command to inspect service logs. If you pass it the service name or ID, you'll

get the logs for all service replicas. If you pass it a specific
replica name or 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.1.i23puo71kq12@wrk1      |  * Serving Flask app 'app'
ddd_web-fe.3.z4otpnjrvc58@wrk2      |  * Debug mode: on
<Snip>
ddd_web-fe.4.novrixi5iuxy@wrk2      |  * Debug mode: on
ddd_web-fe.4.novrixi5iuxy@wrk2      |  * Debugger is active!
ddd_web-fe.4.novrixi5iuxy@wrk2      |  * Debugger PIN: 127-233-151
```

You can follow logs (`--follow`), tail them (`--tail`), and you
can sometimes get extra details (`--details`).

So far, you've used Docker commands to prove the stack is up
and running. However, you can also test it with your browser.

Run a `docker stack services` command to get brief info on
both of the stack's services, including the port the **web-fe**
service is published on.

```
$ docker stack services ddd
NAME          MODE          REPLICAS    IMAGE                                     POF
ddd_redis     replicated    1/1         redis:alpine
ddd_web-fe    replicated    4/4         nigelpoulton/ddd-book:swarm-app     *:5
```

It's published on port `5001`, meaning you can reach it by pointing your browser to the name or IP of any swarm node on `5001`. If you're using Multipass, the node IPs usually start with `192.168`, and you can find them by running a `multipass list` command from your host's terminal. Alternatively, you can run an `ip a` command inside the Multipass VM and use the IP address of the `enp0s1` interface.

Figure 12.2 shows a browser connecting to a swarm node with the `192.168.64.41` IP address and reaching the app.
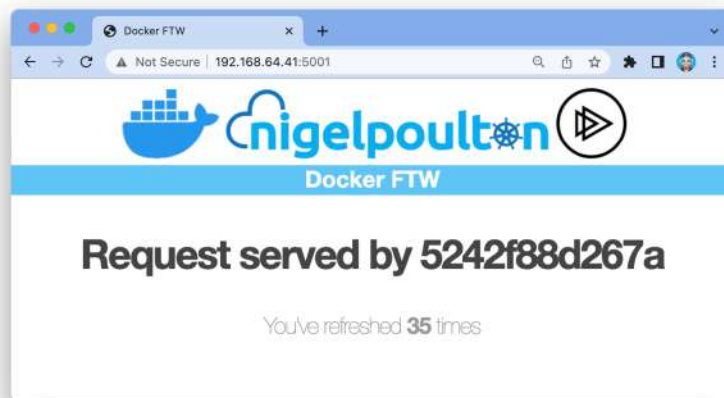
Figure 12.2

Let's switch tack and see how to manage swarm stacks.

Manage the app

There are two ways to manage Docker stacks:

- Imperatively
- Declaratively

The *imperative method* is where you run Docker commands to make changes to the stack. For example, you can use the `docker service scale` command to increase and decrease the number of service replicas.

The *declarative method* is where you make all changes via the stack file. For example, if you want to change the number of service replicas, you edit the stack file with the desired replica count and run another `docker stack deploy` command.

The declarative method is the preferred method.

Consider the following example that demonstrates why you should manage stacks declaratively.

Imagine you've deployed an app from a stack file that includes a **reporting** *service and a* **catalog** *service. The stack file includes other services that are part of the app, but we're only interested in these two. It's currently running five replicas of the reporting service, but year-end reporting has started, and it's experiencing slow performance due to increased demand. You decide to run an*

imperative `docker service scale` *command to increase the number of reporting replicas to 10. This fixes the performance issues, but the current state of the app is no longer in sync with the stack file — the stack file defines five replicas but the cluster is running 10. Later in the day, a colleague is tasked with rolling out a new version of the catalog service — the catalog service is part of the same app and, therefore, is defined in the same stack file as the reporting service. Your colleague makes the changes declaratively by editing the stack file with the new version of the catalog service and then runs a* `docker stack deploy` *command to push the updates. When Docker receives the updated version of the stack file, it rolls out the new version of the catalog service* **and** *changes the number of reporting replicas back to five. This will cause the reporting service to start running slowly again.*

This is why you should make **all changes** declaratively via your stack files and manage them in a version control system.

Getting back to the app you deployed, let's do the following:

- Increase the number of `web-fe` replicas from 4 to 10
- Update the **web-fe** service to the newer **swarm-appv2** image

Edit the `compose.yaml` file, increase the **web-fe** replica count to 10, and change the image to `swarm-appv2` .

```
<Snip>
services:
  web-fe:
    image: nigelpoulton/ddd-book:swarm-appv2     <<---- changed to swarm-a
    command: python app.py
    deploy:
      replicas: 10                               <<---- Changed from 4 to
      <Snip>
```

Save your changes and redeploy the app. This will cause Docker to *rollout* a new version of the **web-fe** service with all 10 replicas running the new image.

```
$ docker stack deploy -c compose.yaml ddd
Updating service ddd_redis (id: ozljsazuv7mmh14ep70pv43cf)
Updating service ddd_web-fe (id: zbbplw0hul2gbr593mvwslz5i)
```

Even though it looks like Docker has redeployed both services, it hasn't. It's clever enough only to redeploy the bits you've changed.

Run a `docker stack ps` to see the rollout's progress.

```
$ docker stack ps ddd
NAME            IMAGE                                      NODE      DESIRED    CUF
```

```
ddd_web-fe.1      nigelpoulton/ddd-book:swarm-app      wrk1     Running    Rur
ddd_web-fe.2      nigelpoulton/ddd-book:swarm-appv2    wrk2     Running    Rur
\_ddd_web-fe.2    nigelpoulton/ddd-book:swarm-app      wrk2     Shutdown   Shu
ddd_web-fe.3      nigelpoulton/ddd-book:swarm-app      wrk2     Running    Rur
<Snip>
```

I've trimmed the output to fit the book, and I've only listed some of the replicas. However, you can see a few things.

Swarm has evenly balanced the six new replicas across both worker nodes. If your swarm has more nodes, Docker will balance the new replicas across them all.

The top line shows the **ddd_web-fe.1** replica running the old image for the last 8 minutes. The next two lines show the **ddd_web-fe.2** replica. You can see that the old instance was running the old image and that it was shut down 26 seconds ago and replaced with a new instance running the new image. The new replica has been running for 13 seconds.

The last line shows the **ddd_web-fe.3** replica is still running the old version.

It's important to emphasize two things.

First, Docker followed the rules in the `deploy.update_config` section of the Compose file. If you look at that section of the file

again, you'll see that Docker can update two replicas, wait for 10 seconds, update the next two, wait 10 seconds, and then attempt a rollback to the previous version if it encounters any issues.

```
web-fe:
  deploy:
    update_config:
      parallelism: 2
      delay: 10s
      failure_action: rollback
```

Second, containers are *immutable* and Docker didn't *update* the existing containers to run the new image. It deleted and replaced them with new containers running the new image.

After a short period, all 10 replicas will be running the updated image.

Before moving on, it's important to remind ourselves of the *reconciliation* process that just happened. The application was running with four **web-fe** replicas, all based on the **swarm-app** image, and this was recorded in the cluster store as *desired state.* We edited the Compose file to change all **web-fe** replicas to use the newer **swarm-appv2** image and increased the replica count from four to ten. We saved our changes, ran a `docker stack deploy` command to push the updated Compose file to the swarm, and the swarm updated the desired state. Shortly

after, swarm compared the *observed state* of the cluster with the new desired state and noticed it had four **web-fe** replicas running the **swarm-app** image but should actually have ten **web-fe** replicas running the **swarm-appv2** image. As such, it deleted the existing four replicas and replaced them with ten new replicas. And it did so according to the update rules defined in the `deploy.update_config` section of the Compose file.

Verify the rollout

Despite it looking like the update worked, there is a problem.

Refresh your browser to make sure you see the updated version of the app with a purple **WebAssembly is the future** banner.

Something didn't work as expected because it's still displaying the old web page!

The issue is with the volume. Let's walk through what happened.

The new image has the updated app version with the purple **WebAssembly is the future** banner. The rollout deleted the old replicas and replaced them with new ones running the new version of the app. So far, so good. However, when Docker started each new replica, it mounted the old volume with the

old app version. This had the effect of overwriting the new version of the app with the old.

You should be aware of this behavior when working with volumes. However, it will usually be fine because you typically only use volumes for data stores, not for hosting application binaries.

Let's assume you realize the **web-fe** service is stateless and doesn't require a volume. The declarative way to remove the volume is to edit the Compose file again, remove the volume and volume mount, and redeploy 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 your changes and redeploy.

```
$ docker stack deploy -c compose.yaml ddd
Updating service ddd_redis (id: ozljsazuv7mmh14ep70pv43cf)
Updating service ddd_web-fe (id: zbbplw0hul2gbr593mvwslz5i)
```

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 as shown in Figure 12.3. Refresh your browser a few times to make sure it works. Don't worry if some requests get the old version while the rollout is in progress.

Figure 12.3 - The updated app

Eventually, all replicas will run the new version and return the new web page. However, the volume will still exist, and you'll need to delete it manually.

Congratulations. You've successfully deployed and managed a multi-container app using Docker Swarm and Docker Stacks. You also learned to deploy and manage apps declaratively by making all changes via the stack file (Compose file).

Clean up

If you've been following along, you've deployed an app with two services, a network, and a volume.

You can delete the stack with the `docker stack rm` command. However, be warned that 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
```

The command deleted the network and services but not the volume. This is because volume lifecycles are decoupled from containers and services, and you need to delete them manually.

Run the following command on every swarm node hosting a replica.

```
$ docker volume rm ddd_counter-vol
ddd_counter-vol
```

Deploying apps with Docker Stacks – The Commands

- `docker stack deploy` is the command you'll run to deploy **and** update stacks. You need to specify the name of the stack and the stack file.
- `docker stack ls` lists all stacks on a swarm and shows the number of services each one has.
- `docker stack ps` gives you detailed information about a stack. It tells you which nodes replicas are running on, which images they're based on, and shows the *desired state* and *current state* of each service replica.
- `docker stack rm` deletes a stack and doesn't ask for confirmation.

Chapter Summary

Stacks are Docker's native solution for running cloud-native microservices applications on Swarm clusters. They offer a simple declarative interface for managing the entire lifecycle of applications and infrastructure.

You start with application code and infrastructure requirements — things like networks, ports, volumes, and secrets. You

containerize the application and combine all the services and infrastructure definitions into a single declarative stack file. You set the number of replicas, as well as rollout and restart policies. Then, you use the `docker stack deploy` command to deploy the application from the stack file.

You should also perform all updates declaratively by updating the stack file and redeploying the app.

13: Docker Networking

It's always the network!

Any time we experience infrastructure issues, we always blame the network. One of the reasons we do this is that networks are at the center of everything. With this in mind, it's important you have a strong understanding of Docker networking.

In the early days of Docker, networking was hard. Fortunately, these days it's almost a pleasure ;-)

This chapter will get you up to speed with the fundamentals of Docker networking. You'll learn all the theory behind the *Container Network Model (CNM)* and *libnetwork*, and you'll get your hands dirty with lots of examples. You'll learn about overlay networks in the next chapter.

I've divided the chapter into the following sections:

- Docker networking – the TLDR
- Docker networking theory
- Single-host bridge networks
- External access via port mappings
- Connecting to existing networks and VLANs
- Service Discovery

- Ingress load balancing

A few quick things before we start.

Everything we'll cover relates to Linux containers, and I recommend you follow along using something like Multipass or Play with Docker, as they give you easy access to some of the Linux commands we'll use. I don't recommend following along on Docker Desktop as it runs everything inside a Linux VM and you won't have access to the Linux commands.

Some of the examples explain how networking works on a swarm. You'll only be able to follow these if you're following along with a Swarm cluster.

Docker Networking – The TLDR

Docker runs microservices applications comprised of many containers that work together to form the overall app. These containers need to be able to communicate, and some will have to connect with external services, such as physical servers, virtual machines, or something else.

Fortunately, Docker has solutions for both of these requirements.

Docker networking is based on *libnetwork*, which is the reference implementation of an open-source architecture called the *Container Network Model (CNM)*.

For a smooth out-of-the-box experience, Docker ships with everything you need for the most common networking requirements, including multi-host container-to-container networks and options for plugging into existing VLANs. However, the model is pluggable, and the ecosystem can extend Docker's networking capabilities via drivers that plug into libnetwork.

Last but not least, libnetwork also provides native service discovery and basic load balancing.

That's the big picture. Let's get into the detail.

Docker networking theory

At the highest level, Docker networking is based on the following three 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 platforms.

Drivers extend the model by implementing specific network topologies such as VXLAN overlay networks.

Figure 13.1 shows all three.
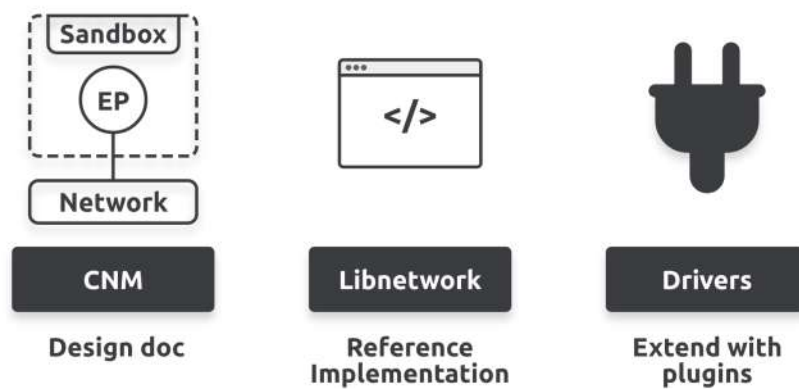
Figure 13.1

Let's take a closer look at each.

The Container Network Model (CNM)

Everything starts with a design.

The design guide for Docker networking is the CNM that outlines the fundamental building blocks of a Docker network.

I recommend you read the [specification document](#), but at a high level, it defines three building blocks:

- Sandboxes
- Endpoints
- Networks

A *sandbox* is an isolated network stack inside a container. It includes Ethernet interfaces, ports, routing tables, DNS configuration, and everything else you'd expect from a network stack.

Endpoints are virtual network interfaces that look, smell, and feel like regular network interfaces. They connect sandboxes to networks.

Networks are virtual switches (usually software implementations of an 802.1d bridge). As such, they group together and isolate one or more endpoints that need to communicate.

Figure 13.2 shows how all three connect and relate to familiar infrastructure components. Using CNM terminology, *endpoints* connect *sandboxes* to *networks*. Every container you create will have a sandbox with at least one endpoint connecting it to a network.
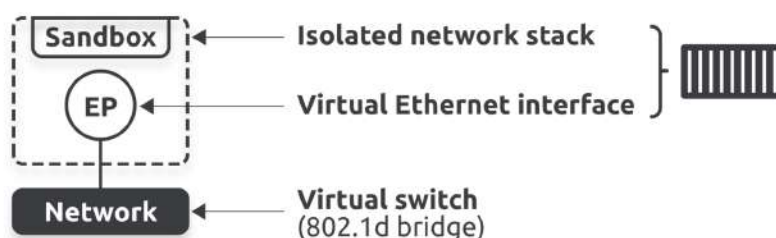
Figure 13.2 The Container Network Model (CNM)

As the name suggests, the <u>Container</u> Network Model is all about providing networking for containers. Figure 13.3 shows how CNM components relate to containers — each container gets its own *sandbox* which hosts the container's entire network stack, including one or more endpoints that act as Ethernet interfaces and can be connected to networks.
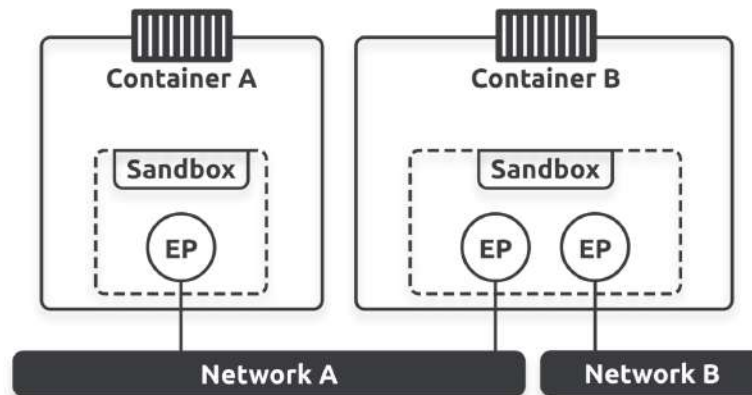
Figure 13.3

Container A has a single interface (endpoint) and is only connected to **Network A**. However, **Container B** has two interfaces connected to **Network A** and **Network B.** The containers can communicate with each other because they are both connected to **Network A**. However, the two endpoints inside of **Container B** cannot communicate with each other as they're on different networks.

It's also important to understand that endpoints behave exactly like regular network adapters, meaning you can only connect them to a single network. This is why **Container B** needs two endpoints if it wants to connect to both networks.

Figure 13.4 extends the diagram further by adding the Docker host. Even though both containers are running on the same

host this time, their network stacks are completely isolated and can only communicate via a network.
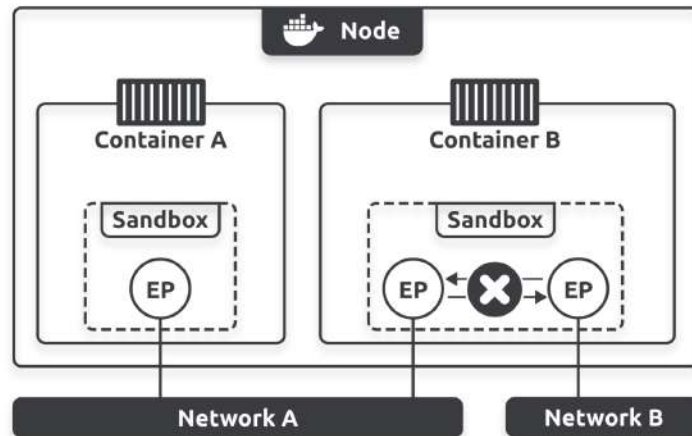
Figure 13.4

Libnetwork

Libnetwork is the reference implementation of the CNM. It's open-source, cross-platform (Linux and Windows), maintained by the Moby project, and used by Docker.

Before Docker created libnetwork, it implemented all of its networking code inside the daemon. However, over time, the daemon became bloated and difficult for other projects to use. As a result, Docker removed the networking code from the daemon and refactored it as an external library called

libnetwork based on the CNM design. Today, Docker implements all of its core networking in libnetwork.

As well as implementing the core components of the CNM, libnetwork also implements the network control plane, including management APIs, service discovery, and ingress-based container load balancing.

Drivers

Libnetwork implements the control plane, but it relies on drivers to implement the data plane. For example, drivers are responsible for creating networks and ensuring isolation and connectivity.

Docker ships with several built-in drivers that we sometimes call *native drivers* or *local drivers*. These include **bridge**, **overlay**, and **macvlan**, and they build the most common network topologies. Third parties can also write network drivers to implement other network topologies and more advanced configurations.

Figure 13.5 shows the roles of libnetwork and drivers and how they relate to control plane and data plane responsibilities.
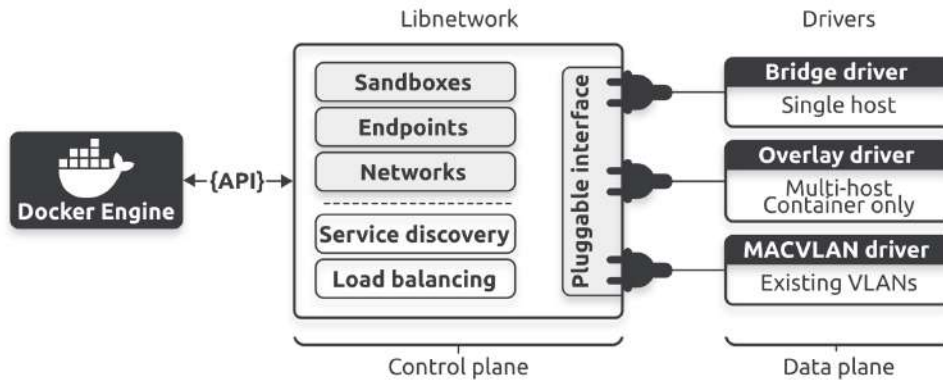
Figure 13.5

Every network you create is owned by a driver, and the driver creates and manages everything about the network. For example, if you create an overlay network called **prod-fe-cuda**, Docker will invoke the overlay driver to create the network and its resources.

To meet the demands of complex, highly fluid environments, a single Docker host or Swarm cluster can have multiple heterogeneous networks managed by different drivers.

Let's look at single-host bridge networks and connecting to existing networks. You'll learn about overlay networks in the next chapter.

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 the network only spans a single Docker host
- **Bridge** tells us that it's an implementation of an 802.1d bridge (layer 2 switch)

Docker creates single-host bridge networks with the built-in **bridge** driver. If you run *Windows containers* you'll need to use the **nat** driver, but for all intents and purposes they work the same.

Figure 13.6 shows two Docker hosts with identical local bridge networks, both called **mynet**. Even though the networks are identical, they are independent and isolated, meaning the containers in the picture cannot communicate, even if the nodes are part of the same swarm.
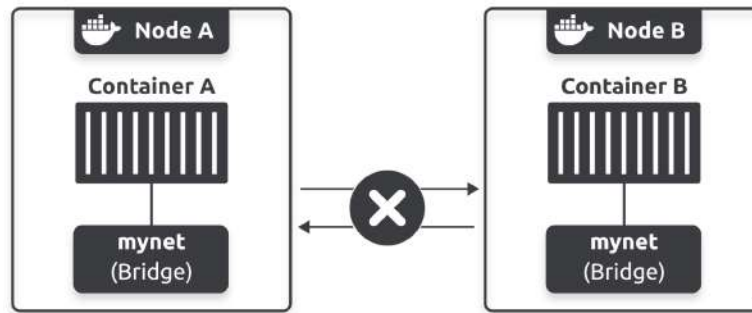
Figure 13.6

Every new Docker host gets a default single-host bridge network called **bridge** that Docker connects new containers to unless you override it with the `--network` flag.

The following commands show the output of a `docker network ls` command on Docker installation.

```
$ docker network ls
NETWORK ID        NAME         DRIVER       SCOPE
c7464dce29ce      bridge       bridge       local     <<---- Default on all
c65ab18d0580      host         host         local
42a783df0fbe      none         null         local
```

As always, you can run `docker inspect` commands to get more information. I highly recommend running the command on your own system and studying the output.

```
$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "c7464dce2...ba2e3b8",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        <Snip>
    }
]
```

All bridge networks are based on the battle-hardened *Linux bridge* technology that has existed in the Linux kernel for over 20 years. This means they're high-performance and highly stable. It also means you can inspect them using standard Linux utilities.

The default *bridge* network on all Linux-based Docker hosts is called **bridge** and maps to an underlying *Linux bridge* in the host's kernel called `docker0` . This is shown in Figure 13.7.
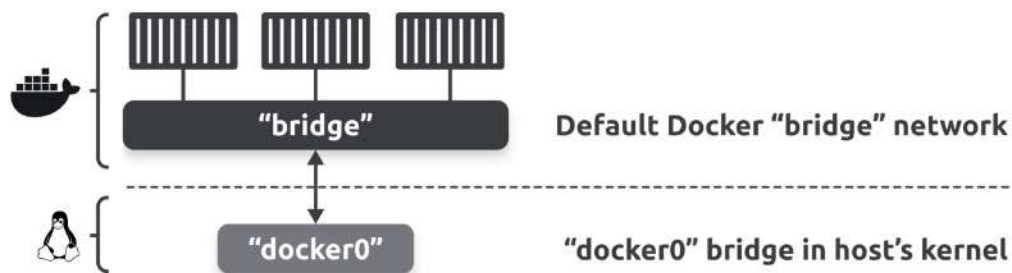

Figure 13.7 - Mapping the default Docker "bridge" network to the "docker0" bridge in the host's kernel

You can run a `docker network inspect` command to confirm that the **bridge** network is based on the **docker0** bridge in the host's kernel. If you're on Windows using PowerShell, you'll need to replace `grep` with `SelectString` .

```
$ docker network inspect bridge | grep bridge.name
"com.docker.network.bridge.name": "docker0",
```

Now run these Linux commands to inspect the **docker 0** bridge from the Linux host. You might need to manually install the `brctl` utility.

```
$ brctl show
bridge name       bridge id           STP enabled     interfaces
docker0           8000.0242aff9eb4f   no
docker_gwbridge   8000.02427abba76b   no

$ 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 first command lists all the *bridges* on your Docker host and shows if they have any devices connected to them. The example in the book shows the `docker0` bridge with no devices connected in the `interfaces` column. You'll only see the `docker_gwbridge` if your host is a member of a swarm cluster.

The second command shows the configuration and state of the `docker0` bridge.

Figure 13.8 shows the complete stack with containers connecting to the `bridge` network, which, in turn, maps to the `docker0` Linux bridge in the host's kernel. It also shows how you can use port mappings to publish connected devices on the Docker host's interface. More on port mappings later.
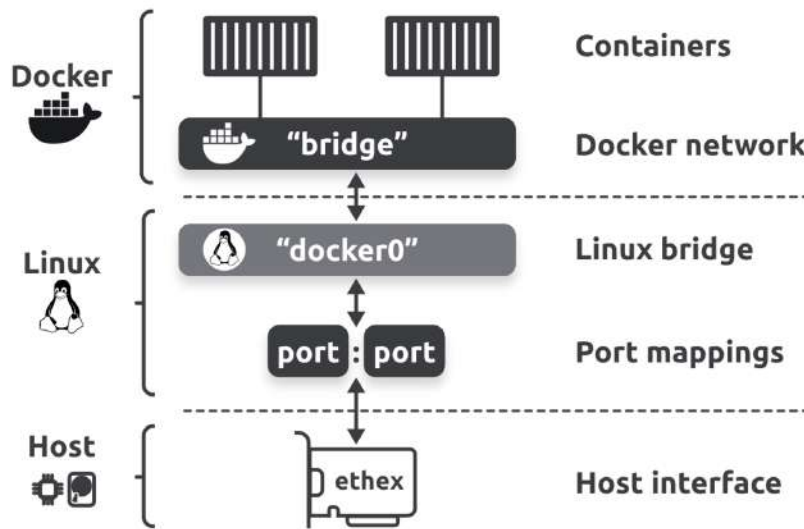
Figure 13.8

In the next few steps, you'll complete all of the following:

1. Create a new Docker bridge network
2. Connect a container to the new network
3. Inspect the new network
4. Test name-based discovery

Run the following command to create a new single-host bridge network called **localnet**.

```
$ docker network create -d bridge localnet
f918f1bb0602373bf949615d99cb2bbbef14ede935fbb2ff8e83c74f10e4b986
```

The long number returned by the command is the network's ID and you'll need it in the next step.

As expected, the command creates a new Docker bridge network called `localnet` that you can list and inspect with the usual `docker` commands. However, behind the scenes, it also creates a new Linux bridge in the host's kernel.

Run another `brctl show` command to see it.

```
$ brctl show
bridge name       bridge id           STP enabled       interfaces
br-f918f1bb0602   8000.0242372a886b   no
docker0           8000.024258ee84bc   no
docker_gwbridge   8000.02427abba76b   no
```

The example in the book shows a new bridge called `br-f918f1bb0602` with no devices connected. If you look closely at the name, you'll recognize `f918f1bb0602` as the first 12 characters from the ID of the new `localnet` network you just created.

At this point, the bridge configuration on the host looks like Figure 13.9, with three Docker networks and three associated bridges in the host's kernel.
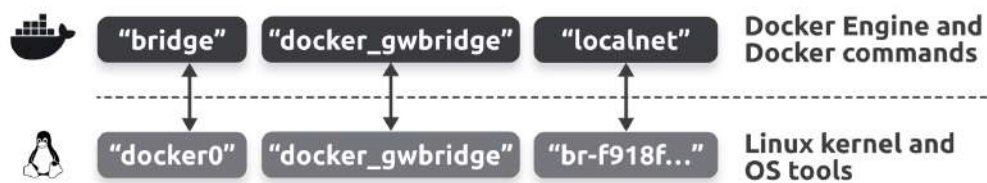
Figure 13.9

Let's create a new container called `c1` and attach it to the new `localnet` bridge network.

```
$ docker run -d --name c1 \
  --network localnet \
  alpine sleep 1d
```

Once you've created the container, inspect the `localnet` network and verify the container is connected to it. You'll need the `jq` utility installed for the command to work. Leave off the `"| jq"` if it doesn't work.

```
$ docker network inspect localnet --format '{{json .Containers}}' | jq
{
  "09c5f4926c87da12039b3b510a5950b3fe9db80e13431dc17d870450a45fd84a": {
    "Name": "c1",
    "EndpointID": "27770ac305773b352d716690fb9f8e05c1b71e10dc66f67b88e93cb
    "MacAddress": "02:42:ac:15:00:02",
    "IPv4Address": "172.21.0.2/16",
    "IPv6Address": ""
  }
}
```

The output shows the `c1` container and its IP address. This proves Docker connected it to the network.

If you run another `brctl show` command, you'll see the `c1` container's interface connected to the `br-1597657726bc` bridge.

```
$ brctl show
bridge name         bridge id           STP enabled     interfaces
br-f918f1bb0602     8000.0242372a886b   no              veth833aaf9
docker0             8000.024258ee84bc   no
docker_gwbridge     8000.02427abba76b   no
```

Figure 13.10 shows the updated configuration. Your veth IDs will be different, but the important thing to understand is that every veth is like a cable with an interface on either end. One end is connected to the Docker network, and the other end is connected to the associated bridge in the kernel.
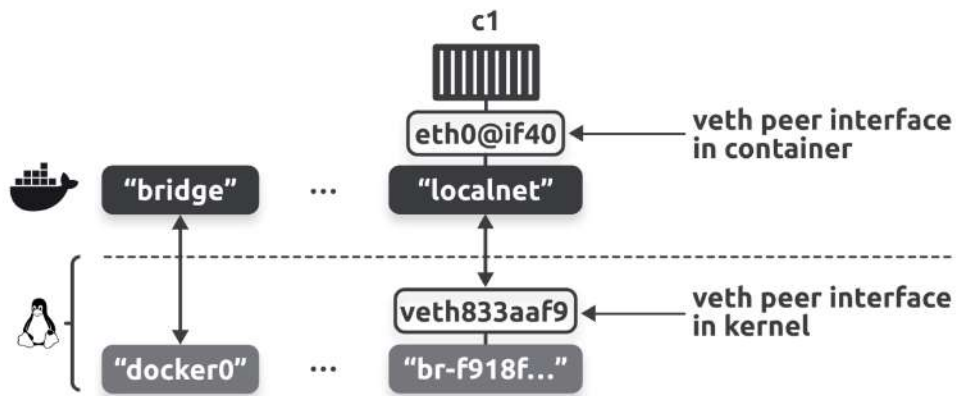
Figure 13.10

If you add more containers to the `localnet` network, they'll all be able to communicate using names. This is because Docker automatically registers container names with an internal DNS service and allows containers on the same network to find each other by name. The exception to this rule is the built-in `bridge` network that does not support DNS resolution.

Let's test name resolution by creating a new container called `c2` on the same `localnet` network and seeing if it can ping the `c1` container.

Run the following command to create the `c2` container on the `localnet` network. You'll need to type `exit` if you're still logged in to the `c1` container.

```
$ docker run -it --name c2 \
  --network localnet \
  alpine sh
```

Your terminal will switch into the `c2` container.

Try to ping the `c1` container by name.

```
# ping c1
PING c1 (172.21.0.2): 56 data bytes
64 bytes from 172.21.0.2: seq=0 ttl=64 time=1.564 ms
64 bytes from 172.21.0.2: seq=1 ttl=64 time=0.338 ms
64 bytes from 172.21.0.2: seq=2 ttl=64 time=0.248 ms
<Control-c>
```

It works! This is because all containers run a DNS resolver that forwards name lookups to Docker's internal DNS server that holds name-to-IP mappings for all containers started with the `--name` or `--net-alias` flag.

Type `exit` to log out of the container and return to your local shell.

External access via port mappings

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 by mapping containers to

ports on the Docker host. It's a bit clunky and has a lot of limitations, but it might be useful for occasional testing and development work.

Figure 13.11 shows a single Docker host running two containers. The `web` container on the right is running a web server on port `80` that is mapped to port `5005` on the Docker host. The `client` container on the left is sending requests to the Docker host on port `5005` and the external client at the bottom is doing the same. Both requests will hit the host's IP on port `5005` and be redirected to the web server running in the `web` container.
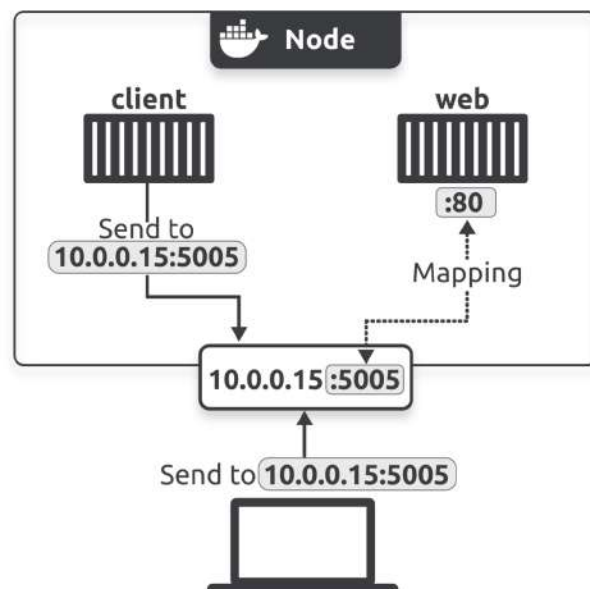

Figure 13.11

Let's test the setup to see if it works.

Create a new container called `web` running NGINX on port `80` and map it to port `5005` on the Docker host. If you're still logged on to the container from the previous example, you'll need to type `exit` first.

```
$ docker run -d --name web \
  --network localnet \
  --publish 5005:80 \
  nginx
```

Verify the port mapping.

```
$ docker port web
80/tcp -> 0.0.0.0:5005
80/tcp -> [::]:5005
```

The output shows the port mapping exists on all interfaces on the Docker host.

You can test external access by pointing a web browser to the Docker host on port `5005`. You'll need to know the IP or DNS name of your Docker host (if you're following along on Multipass it will probably be your Multipass VM's `192.168.x.x` address). You'll see the *Welcome to nginx!* page.

Let's create another container and see if it can reach the web container via the port mapping.

Run the following command to create a new container called `client` on the `bridge` network.

```
$ docker run -it --name client --network bridge alpine sh
#
```

The command will log you into the container and your prompt will change.

Install the `curl` utility.

```
# apk add curl
fetch https://dl-cdn.alpinelinux.org/alpine/v3.19/main/aarch64/APKINDEX.ta
fetch https://dl-cdn.alpinelinux.org/alpine/v3.19/community/aarch64/APKIND
(1/8) Installing ca-certificates (20240226-r0)
<Snip>
```

Now connect to the IP of your Docker host on port `5005` to see if you can reach the container.

```
# curl 192.168.64.69:5005
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
```

```
...
</html>
```

You've reached the NGINX web server running on the `c1` container via a port mapping to the Docker host's IP.

Even though this works, it's clunky and doesn't scale. For example, no other containers or host processes will be able to use port `5005` on the host. This is one of the reasons that single-host bridge networks are only useful for local development or very small applications.

Connecting to existing networks and VLANs

The ability to connect containerized apps to external systems and physical networks is important. A common example is partially containerized apps where the parts running in containers need to be able to communicate with the parts not running in containers.

The built-in **MACVLAN** driver (**transparent** if you're using Windows containers) was created with this in mind. It gives every container its own IP and MAC address on the external physical network, making each one look, smell, and feel like a physical server or VM. This is shown in Figure 13.12.
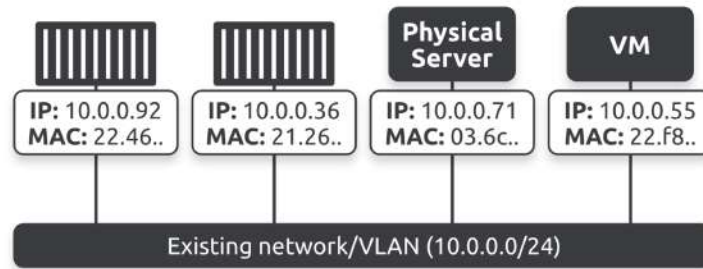
Figure 13.12 - MACVLAN driver making containers visible on external networks

On the positive side, MACVLAN performance is good as it doesn't require port mappings or additional bridges. However, you need to run your host NICs in *promiscuous mode*, which isn't allowed on many corporate networks and public clouds. So, MACVLAN will work on 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 only work if your host NIC is in promiscuous mode on a network that allows it. It also requires an existing VLAN 100. You can adapt it if the VLAN config on your physical network is different. You can follow along without the VLANs, but you won't get the full experience.

Assume you have the network shown in Figure 13.13 with two VLANs:

Figure 13.13

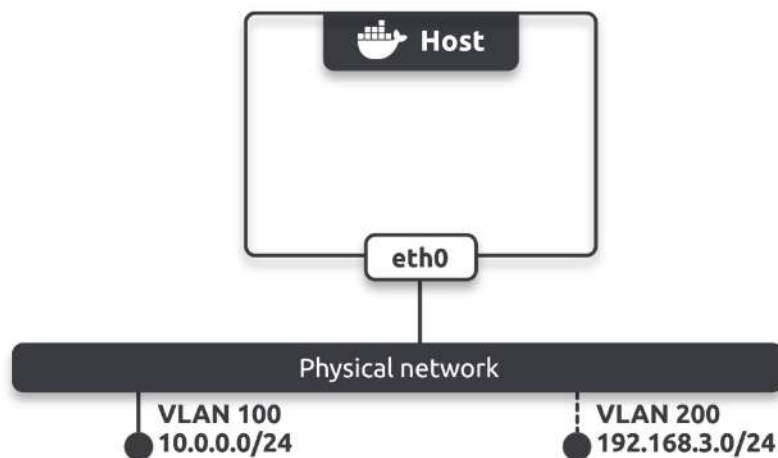Next, you add a Docker host and connect it to the network.

Figure 13.14

Now comes the requirement to attach a container to VLAN 100. To do this, you create a new Docker network with the `macvlan` driver and configure it with all of the following:

- Subnet info
- Gateway
- Range of IPs it can assign to containers
- Which of the host's interfaces or sub-interfaces to use

Run the following command to create a new MACVLAN network called **macvlan100** that will connect containers to VLAN 100. You may need to change the name of the parent interface to match the parent interface name on your system. For example, changing `-o parent=eth0.100` to `-o parent=enp0s1.100`. The parent interface must be connected to the VLAN, and you'll need to type `exit` if you're still logged on to the container from the previous example.

```
$ 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 \        <<---- Make sure this matches your system
  macvlan100
```

Docker will create the `macvlan100` network and a new sub-interface on the host called `eth0.100@eth0`. The config now looks like this.
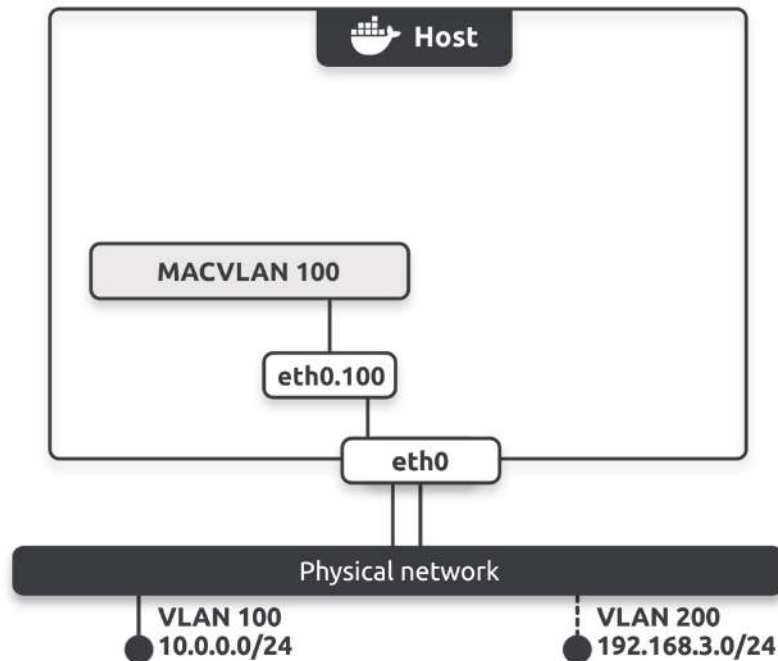
Figure 13.15

The MACVLAN driver creates standard Linux *sub-interfaces* and tags 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 new network which sub-set of IP addresses it can assign to containers. It's vital that you reserve this range of addresses for Docker, as the MACVLAN driver has no management plane feature to check if IPs are already in use.

If you inspect the network, you'll be able to see the important configuration information. I've snipped the output to show the most relevant parts.

```
$ docker network inspect macvlan100
[
    {
        "Name": "macvlan100",
        "Driver": "macvlan",
        "IPAM": {
            "Config": [
                {
                    "Subnet": "10.0.0.0/24",
                    "IPRange": "10.0.0.0/25",
                    "Gateway": "10.0.0.1"
                }
            ]
        },
        "Options": {
            "parent": "enp0s1.100"
        },
    }
]
```

Once you've created the `macvlan100` network, you can connect containers to it and Docker will assign the IP and MAC addresses on the underlying VLAN so they'll be visible to other systems.

The following command creates a new container called `mactainer1` and connects it to the `macvlan100` network.

```
$ docker run -d --name mactainer1 \
  --network macvlan100 \
  alpine sleep 1d
```
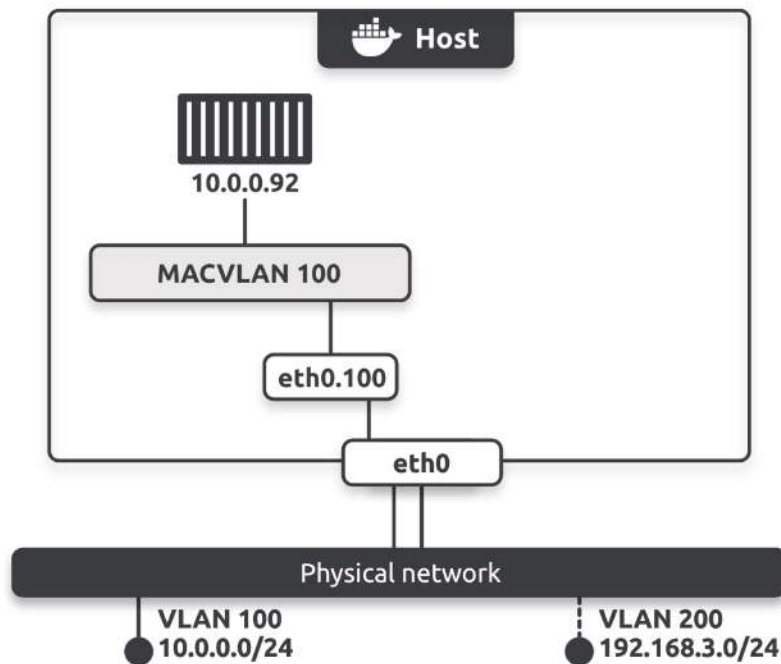
The config now looks like Figure 13.16.

Figure 13.16

However, remember that the underlying network (VLAN 100) does not see any of the MACVLAN magic, it only sees the container with its MAC and IP addresses, meaning the **mactainer1** container will be able to communicate with every other system connected to VLAN 100!

Note: *If you can't get this to work, it might be because your host NIC isn't in promiscuous mode. Also, remember that public cloud platforms normally block promiscuous mode.*

At this point, you've got a MACVLAN network and used it to connect a new container to an existing VLAN. If you have the complete setup, with the existing VLAN, you can test that the container is reachable form other system on the VLAN.

However, it doesn't stop there. The Docker MACVLAN driver supports VLAN trunking. This means you can create multiple MACVLAN networks that connect to different VLANs. Figure 13.17 shows a single Docker host running two MACVLAN networks connecting containers to two different VLANs.
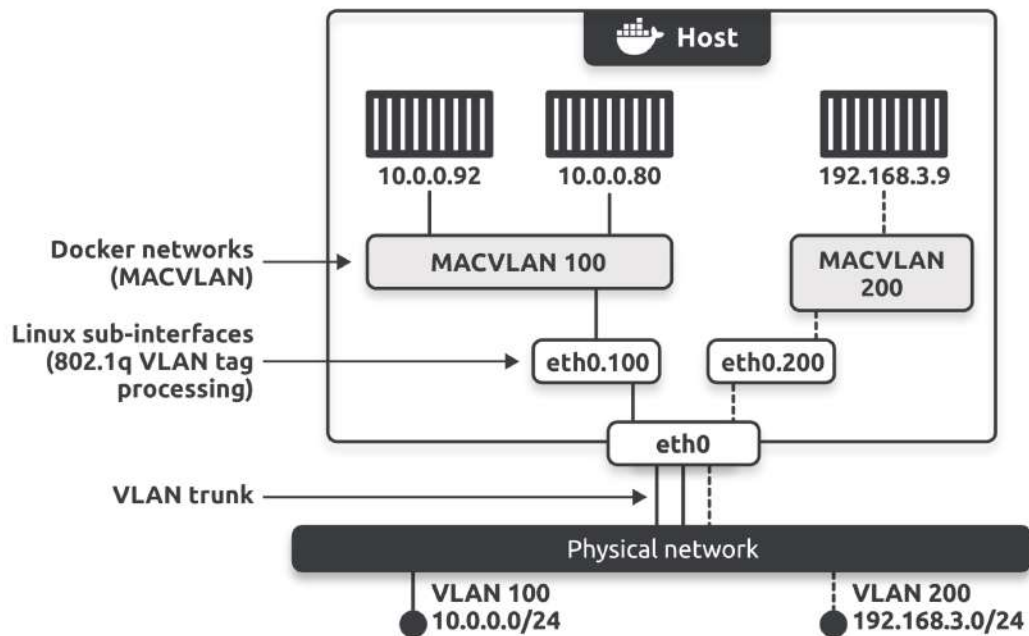
Figure 13.17

Troubleshooting connectivity problems

A quick note on troubleshooting connectivity issues before moving on to service discovery.

Daemon logs and container logs can be useful when troubleshooting connectivity issues.

If you're running Windows containers, you can view them in the Windows Event Viewer or directly in `~\AppData\Local\Docker`. For Linux containers, it depends on which *init* system you're using. If you're running a systemd,

Docker will post logs to journald and you can view them with the `journalctl -u docker.service` command. If you're using a different init system, you might want to check 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 at `/etc/docker/daemon.json` and set `"debug"` to `"true"` and `"log-level"` 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 your `daemon.json` file doesn't exist, create it. Also, be sure to restart Docker after making any changes to the file.

That was the daemon logs. What about container logs?

You can normally view container logs with the `docker logs` command. If you're running Swarm, you should use the `docker service logs` command. However, Docker supports a few different log drivers, and they don't all work with native Docker commands. For some of them, you might have to view logs using the platform's native tools.

`json-file` and `journald` are probably the easiest to configure and they both work with the `docker logs` and `docker service logs` commands.

The following snippet from a `daemon.json` shows a Docker host configured to use `journald`.

```
{
  "log-driver": "journald"
```

```
    }
```

You can also start a container or a service with the `--log-driver` and `--log-opts` flags to override the settings in `daemon.json`.

Container logs work on the premise that your application runs as PID 1 and sends logs to `STDOUT` and errors to `STDERR`. The logging driver then forwards everything 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` that is configured with the `json-file` logging driver.

```
$ docker logs vantage-db
1:C 2 Feb 09:53:22.903 # oO0oo000o000o Redis is starting oO0oo000o000o
1:C 2 Feb 09:53:22.904 # Redis version=4.0.6, bits=64, commit=00000000, mo
1:C 2 Feb 09:53:22.904 # Warning: no config file specified, using the defa
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
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 in the daemon logs or container logs.

Service discovery

As well as core networking, *libnetwork* also provides *service discovery* that allows all containers and Swarm services to locate each other by name. The only requirement is that the containers be on the same network.

Under the hood, Docker implements a native DNS server and configures every container to use it for name resolution.

Figure 13.18 shows a container called **c1** pinging another container called **c2** by name. The same principle applies to Swarm service replicas.
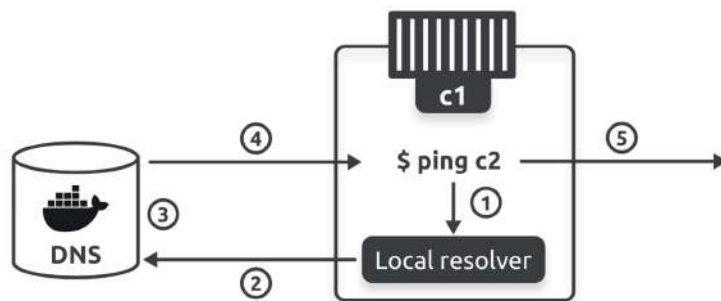

Figure 13.18

Let's step through the process.

- **Step 1:** The **c1** container issues a `ping c2` command. The container's local DNS resolver checks its cache to see if it has an IP address for **c2**. All Docker containers have a local DNS resolver.
- **Step 2:** The local resolver doesn't have an IP address for **c2**, so it initiates a recursive query to the embedded Docker DNS server. All Docker containers are pre-configured to know how to send queries to the embedded DNS server.
- **Step 3:** The Docker DNS server maintains name-to-IP mappings for every container you create with the `--name` or `--net-alias` flags. This means it knows the IP address of the **c2** container.
- **Step 4:** The DNS server returns the IP address of the **c2** container to the local resolver in the **c1** container. If **c1** and **c2** are on different Docker networks it won't return the IP address — name resolution only works for containers on the same network.
- **Step 5:** The **c1** container sends the ping request (ICMP echo request) to the IP address of **c2**.

Just to confirm a few points.

Docker will automatically register the name and IP of every container you create with the `--name` or `net-alias` flag with the embedded Docker DNS service. It also automatically

configures every container to use the embedded DNS service to convert names to IPs. And name resolution (service discovery) is *network scoped*, meaning it only works for containers and services on the same network.

One last point on service discovery and name resolution...

You can use the `--dns` flag to start containers and services with a customized list of DNS servers, and you can use the `--dns-search` flag to add custom search domains for queries against unqualified names (i.e., when the application doesn't specify fully qualified DNS names for services they consume). You'll find both of these useful if your applications query names outside of your Docker environment such as internet services.

Both of these options work by adding entries to the container's `/etc/resolv.conf` file.

Run the following command to start a new container with the infamous `8.8.8.8` Google DNS server and `nigelpoulton.com` as a search domain for unqualified queries.

```
$ docker run -it --name custom-dns \
  --dns=8.8.8.8 \
  --dns-search=nigelpoulton.com \
  alpine sh
```

Your shell prompt will change to indicate you're connected to the container.

Inspect its `/etc/resolv.conf` file.

```
# cat /etc/resolv.conf

  Generated by Docker Engine.
  This file can be edited; Docker Engine will not make further changes on
  has been modified.

nameserver 8.8.8.8
search nigelpoulton.com
```

The file's contents might be slightly different if you connect the container to a custom network, but the options work the same.

Type `exit` to return to your local terminal.

Ingress load balancing

This section only applies to Docker Swarm.

Swarm supports two ways of publishing services to external clients:

- Ingress mode (default)
- Host mode

External clients can access *ingress mode* services via any swarm node — even nodes not hosting a service replica. However, they can only access *host mode* services via nodes running replicas. Figure 13.19 shows both modes.
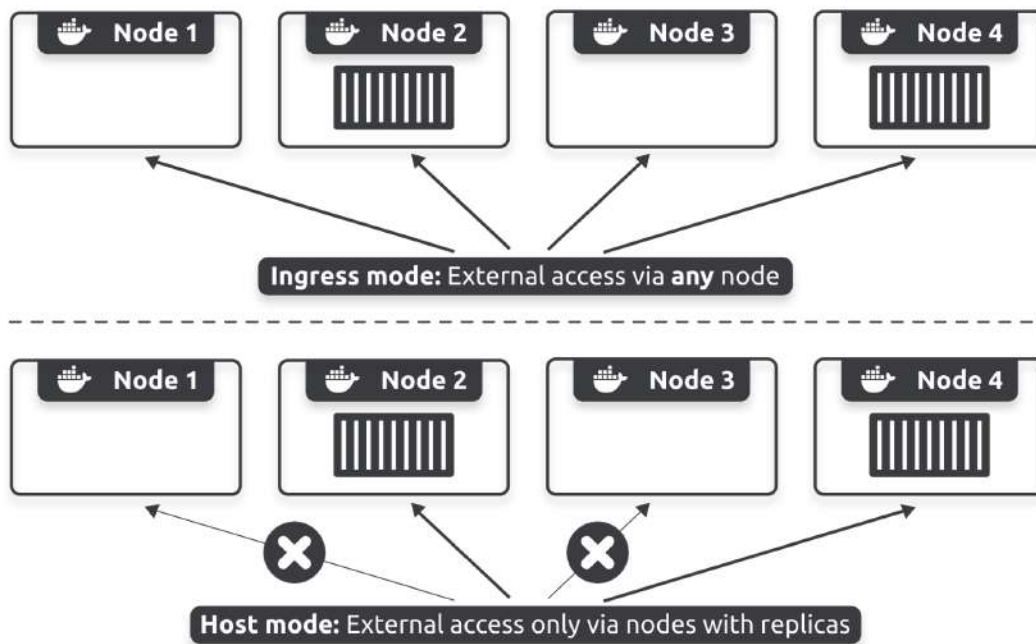

Figure 13.19

Ingress mode is the default, meaning any time you create a service with `-p` or `--publish`, Docker will publish it in *ingress mode*. If you want to publish a service in *host mode*, you'll need to use the `--publish` flag with the `mode=host` option. The following example publishes a service in host mode and will only work on a swarm.

```
$ docker service create -d --name svc1 \
  --publish published=5005,target=80,mode=host \
  nginx
```

A few notes about the command. `docker service create` lets you publish services using either *long form syntax* or *short form syntax*.

The short form looks like `-p 5005:80` and you've seen it a few times already. However, you cannot publish a service in *host mode* using the short form.

Long form looks like this: `--publish published=5005,target=80,mode=host`. It's a comma-separated list with no whitespace after the commands, and the options work as follows:

- `published=5005` makes the service available to external clients via port `5005`
- `target=80` makes sure requests hitting the *published* port get mapped back to port `80` on service replicas
- `mode=host` makes sure requests will only reach the service if they arrive on nodes running a service replica

You'll almost always use ingress mode.

Behind the scenes, ingress mode uses a layer 4 routing mesh that Docker calls the **service mesh** or the **swarm-mode service mesh**. Figure 13.20 shows the basic traffic flow when an external request hits the cluster for a service exposed in ingress mode.
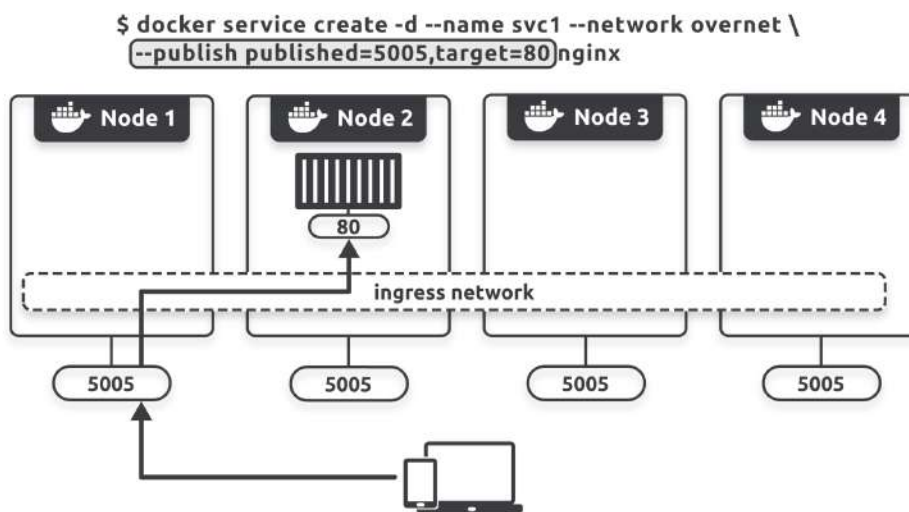

Figure 13.20

Let's quickly walk through the diagram.

The command at the top deploys a new Swarm service called `svc1` with one replica, attaches it to the `overnet` network and publishes it on port `5005` on the *ingress network*. Docker automatically creates the ingress network when you create the swarm, and it attaches every node to it. The act of publishing

the service on port `5005` makes it accessible via port `5005` on every swarm node because every node is connected to the ingress network. Docker also creates a swarm-wide rule to route all traffic hitting nodes on port `5005` to port `80` in the svc1 replicas via the ingress network.

Now let's track that external request.

1. The external client sends a request to **Node 1** on port `5005`
2. **Node** 1 receives the request and knows to forward traffic arriving on port `5005` to the ingress network
3. The ingress network forwards the request to **Node 2** which is running a replica
4. **Node 2** receives the request and passes it to the replica on port `80`

If the service has multiple replicas, swarm is clever enough to balance requests across them all.

Clean up

If you've been following along, you'll have a lot of containers, networks, and services that you probably want to clean up.

Run the following command to delete the services you created.

```
$ docker service rm svc1
```

Now, delete the standalone containers you created.

```
$ docker rm c1 c2 client web mactainer1 -f
```

Finally, delete the networks you created.

```
$ docker network rm localnet macvlan100
```

Docker Networking – The Commands

Docker networking has its own `docker network` sub-command, and the main commands include:

- `docker network ls` lists all the Docker networks available to the host.
- `docker network create` is how you create a new Docker network. You have to give the network a name and you can use the `-d` flag to specify which driver creates it.
- `docker network inspect` provides detailed configuration information about Docker networks.
- `docker network prune` deletes all unused networks on a Docker host.

- **`docker network rm`** Deletes specific networks on a Docker host or swarm.

You also ran some native Linux commands.

- **`brctl show`** prints a list of all kernel bridges on the Docker host and shows if any containers are connected.
- **`ip link show`** prints bridge configuration data. You ran an **`ip link show docker0`** to see the configuration of the **docker0** bridge on your Docker host.

Chapter Summary

The Container Network Model (CNM) is the design document for Docker networks and defines the three major constructs — *sandboxes*, *endpoints*, and *networks*.

Libnetwork is the reference implementation of the CMN and is an open-source project maintained by the Moby project. Docker uses it to implement its core networking, including control plane services such as service discovery.

Drivers extend the capabilities of libnetwork by implementing specific network topologies, such as bridge and overlay networks. Docker ships with built-in drivers, but you can also use third-party drivers.

Single-host *bridge networks* are the most basic type of Docker network but are only suitable for local development and very small applications. They do not scale, and you need to map containers to host ports if you want to publish 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 lets you create Docker networks that connect containers to existing physical networks and VLANs. They make containers first-class citizens on external networks by giving them their own MAC and IP addresses. Unfortunately, you have to run your host NICs in promiscuous mode, meaning they won't work in public clouds.

14: Docker overlay networking

Overlay networks are at the center of most cloud-native microservices apps, and this chapter will get you up to speed on how they work in Docker.

I've divided the chapter into the following sections:

- Docker overlay networking – The TLDR
- Docker overlay networking history
- Building and testing overlay networks
- Overlay networks explained

Let's do some networking magic!

Docker overlay networking – The TLDR

Real-world containers need a reliable and secure way to communicate without caring which host they're running on or which networks those hosts are connected to. This is where overlay networks come into play — they create flat, secure, layer 2 networks that span 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, Docker builds overlay networking 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 machinery to build overlay networks.

Docker overlay networking history

In March 2015, Docker, Inc. acquired a container networking startup called *Socket Plane* with two goals in mind:

1. Bring overlay networking to Docker
2. Make container networking simple for developers

They accomplished both goals, and overlay networking continues to be at the heart of container networking in 2024 and the foreseeable future.

However, there's a lot of complexity hiding behind the simple Docker commands. Knowing the commands is probably enough if you're a casual Docker user. However, if you plan to use Docker in production, especially if you plan to use Swarm and Docker networking, then the things we'll cover will be vital.

Building and testing Docker overlay networks

You'll need at least two Docker nodes configured in a swarm to follow along. The examples in the book show the two nodes on different networks connected by a router, but yours can be on the same network. You can follow along with two Multipass VMs on the same laptop or computer, but any Docker configuration will work as long as the nodes can communicate. I don't recommend using Docker Desktop as you only get a single node and won't get the full experience.

Figure 14.1 shows the initial lab configuration. Remember, your nodes can be on the same network, this will just mean your *underlay network* is simpler. We'll explain underlay networks later.
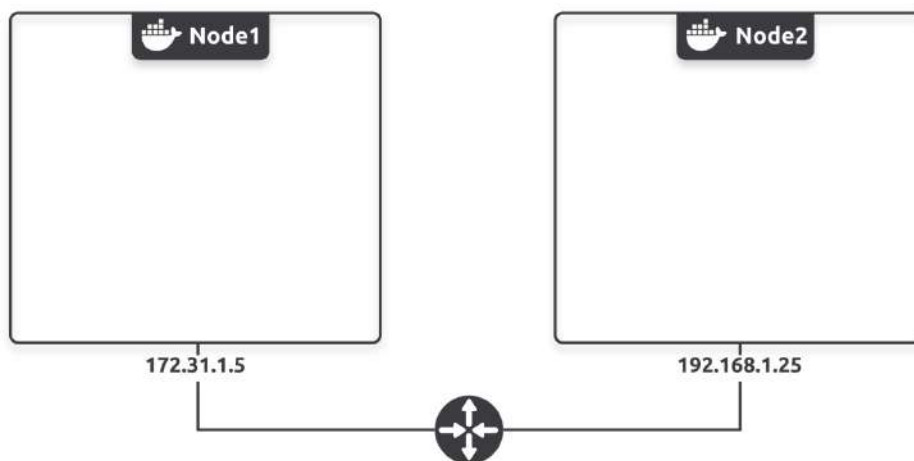

Figure 14.1

Build a Swarm

If you're following along, you'll need a swarm because overlay networks leverage the swarm's key-value store and other security features.

This section builds a two-node swarm with two Docker nodes called **node1** and **node2**. If you already have a swarm, you can skip this section.

You'll need to substitute the IP addresses and names with the values from your environment. You'll also need to ensure the following network ports are open 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

Swarm initialized: current node (1ex3...o3px) is now a manager.
```

The command output includes a `docker swarm join` command. Copy this command and run it **node2**.

```
$ docker swarm join \
  --token SWMTKN-1-0hz2ec...2vye \
  172.31.1.5:2377
This node joined a swarm as a worker.
```

You 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 encrypted overlay network called **uber-net**.

Run the following command from your manager node (**node1**).

```
$ docker network create -d overlay -o encrypted uber-net
vdu1yly429jvt04hgdm0mjqc6
```

That's it. You've created a brand-new encrypted overlay network. The network spans both nodes in the swarm and Docker uses TLS to encrypt it (AES in GCM mode). It also rotates the encryption keys every 12 hours.

If you don't specify the `-o encrypted` flag, Docker will still encrypt the control plane (management traffic) but won't encrypt the data plane (application traffic). This can be important, as encrypting the data plane can decrease network performance by approximately 10%.

List the networks on **node1**.

```
$ docker network ls
NETWORK ID      NAME             DRIVER    SCOPE
65585dda7500    bridge           bridge    local
7e368a1105c7    docker_gwbridge  bridge    local
a38083cdab1c    host             host      local
4dsqo7jc36ip    ingress          overlay   swarm
d97e92a23945    none             null      local
vdu1yly429jv    uber-net         overlay   swarm    <<---- New overlay netw
```

The new network is at the bottom of the list called **uber-net** and is scoped to the entire swarm (`SCOPE = swarm`). This means it spans every node in the swarm. However, if you list networks on **node2** you won't see the **uber-net** network. This is because Docker only extends overlay networks to worker nodes when they need them. In our example, Docker will extend the **uber-net** network to **node2** when it runs a container that needs it. This lazy approach to network deployment improves scalability by reducing the amount of network gossip on the swarm.

Attach a container to the overlay network

Now that you have an overlay network let's connect a container to it.

By default, you can only attach containers that are part of *swarm services* to overlay networks. If you want to add *standalone containers*, you need to create the overlay with the `--attachable` flag.

The example will create a swarm service called **test** with two replicas on the **uber-net** network. One replica will be deployed to **node1** and the other to **node2**, causing Docker to extend the overlay network to **node2**.

Run the following commands from **node1**.

```
$ docker service create --name test \
   --network uber-net \
   --replicas 2 \
   ubuntu sleep infinity
```

Check the status of the service.

```
$ docker service ps test
ID          NAME      IMAGE           NODE      DESIRED STATE    CURRENT STAT
sm1...1nw   test.1    ubuntu:latest   node1     Running          Running
tro...kgk   test.2    ubuntu:latest   node2     Running          Running
```

The `NODE` column shows one replica running on each node.

Switch over to **node2** and run a `docker network ls` to verify it can now see the **uber-net** network.

Congratulations. You've created a new overlay network spanning two nodes on separate underlay networks and attached two containers to it. You'll appreciate the simplicity of what you've done when we reach the theory section and learn about the outrageous complexity going on behind the scenes!

Test the overlay network

Figure 14.2 shows the current setup with two containers running on different Docker hosts but connected to the same overlay.
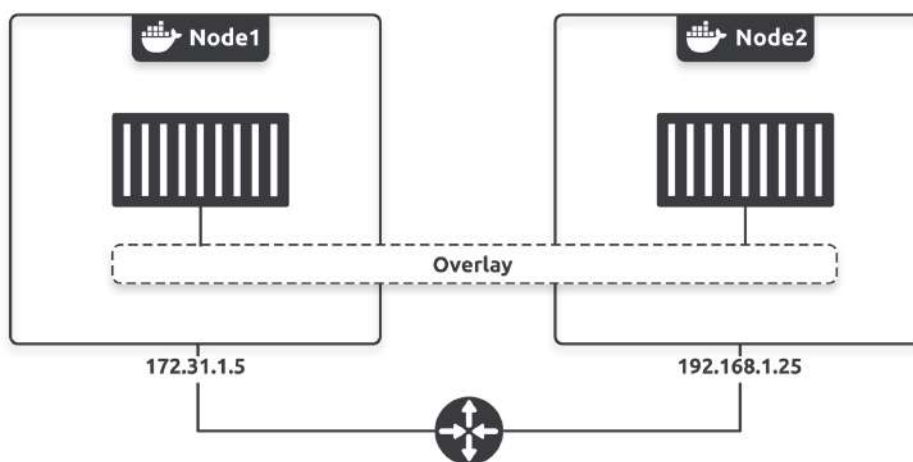

Figure 14.2

The following steps will walk you through obtaining the container names and IP addresses and then seeing if they can ping each other.

Switch back to **node1** and run a `docker network inspect` to see the overlay network's subnet information and any IP addresses it's assigned to service replicas.

```
$ docker network inspect uber-net
[
    {
        "Name": "uber-net",
        "Id": "vdu1yly429jvt04hgdm0mjqc6",
        "Scope": "swarm",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "10.0.0.0/24",              <<---- Subr
                    "Gateway": "10.0.0.1"                 <<---- Subr
                }
            "Containers": {
                    "Name": "test.1.tro80xqwm7k1bsyn3mt1fjkgk",    <<---- Repl
                    "IPv4Address": "10.0.0.3/24",                   <<---- Cont
                    <Snip>
                },
    <Snip>
```

I've snipped the output and highlighted the subnet info and the IPs of connected containers. One thing to note is that Docker only shows you the IP addresses of containers running on the local node. For example, the output in the book only shows the IP of the first replica called `test.1.tro...kgk`. If you run the same command on `node2`, you'll see the name and IP of the other replica.

Run the following commands on both nodes to get the local container names, IDs, and IP addresses of both replicas and make a note of them.

The ID at the end of the second command (`d7766923a5a7`) is the container ID as returned by the `docker ps` command. You'll need to substitute the value from your environment.

```
$ docker ps
CONTAINER ID    IMAGE            COMMAND            CREATED      STATUS       NAM
d7766923a5a7    ubuntu:latest    "sleep infinity"   2 hrs ago    Up 2 hrs     tes

$ docker inspect \
   --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' d776
10.0.0.3
```

I have the following in my environment :

- **replica 1:** `ID=d7766923a5a7, Name=test.1.tr0...kgk,`
  `IP=10.0.0.3`
- **replica 2:** `ID=b6c897d1186d, Name=test.2.sm1...1nw,`
  `IP=10.0.0.4`

Figure 14.3 shows the configuration so far. Subnet and IP addresses may be different in your lab.
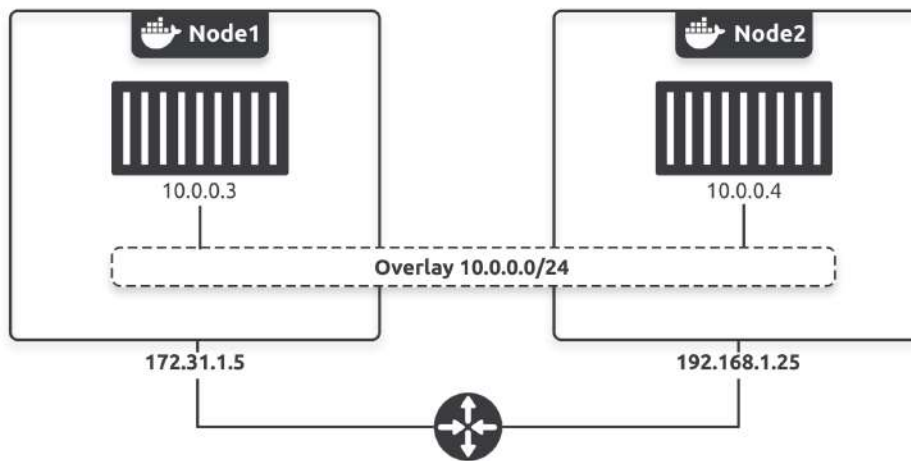

Figure 14.3

As you can see, a layer 2 overlay network spans both nodes, and each container is connected to it with its own IP. This means the container on **node1** can ping the container on **node2** even though both nodes are on different underlay networks.

Let's test it. You'll need the names and IPs of your containers.

Log on to either of the containers and install the `ping` utility.

```
$ docker exec -it d7766923a5a7 bash

# apt 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) ...
```

Now ping the remote container by IP and then by replica ID.

```
# 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

# ping test.2.sm180xqwm7k1bsyn3mt1fj1nw
PING test.2.sm180xqwm7k1bsyn3mt1fj1nw (10.0.0.4) 56(84) bytes of data.
64 bytes from test.2.sm1...1nw.uber-net (10.0.0.4): icmp_seq=1 ttl=64 time
64 bytes from test.2.sm1...1nw.uber-net (10.0.0.4): icmp_seq=2 ttl=64 time
64 bytes from test.2.sm1...1nw.uber-net (10.0.0.4): icmp_seq=3 ttl=64 time
^C
```

Congratulations. The containers can ping each other via the overlay network, and all the traffic is encrypted.

You can also trace the route of the ping command. This will report a single hop, proving that the containers are communicating directly via the overlay network — blissfully unaware of any underlay networks being traversed.

You'll need to install `traceroute` in the container for this to work.

```
# apt install 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.sm180xqwm7k1bsyn3mt1fj1nw.uber-net (10.0.0.4)  1.110ms  1.0
```

So far, you've created an overlay network and a swarm service that connected two containers to it. Swarm scheduled the containers to two different nodes and you proved they could ping each other via the overlay network.

Now that you've seen how easy it is to build and use secure overlay networks, let's find out how Docker builds them behind the scenes.

Overlay networks explained

First and foremost, Docker uses *VXLAN tunnels* to create virtual layer 2 overlay networks. So, let's do a quick VXLAN primer.

VXLAN primer

At the highest level, Docker uses VXLANs to create layer 2 networks on top of existing layer 3 infrastructure. That's a lot of jargon that means you can create simple networks on top of complex networks. The hands-on example in the previous sections created a new 10.0.0.0/24 layer 2 network that abstracted a more complex network topology below. See Figure 14.4 and remember that your underlay network configuration was probably different.
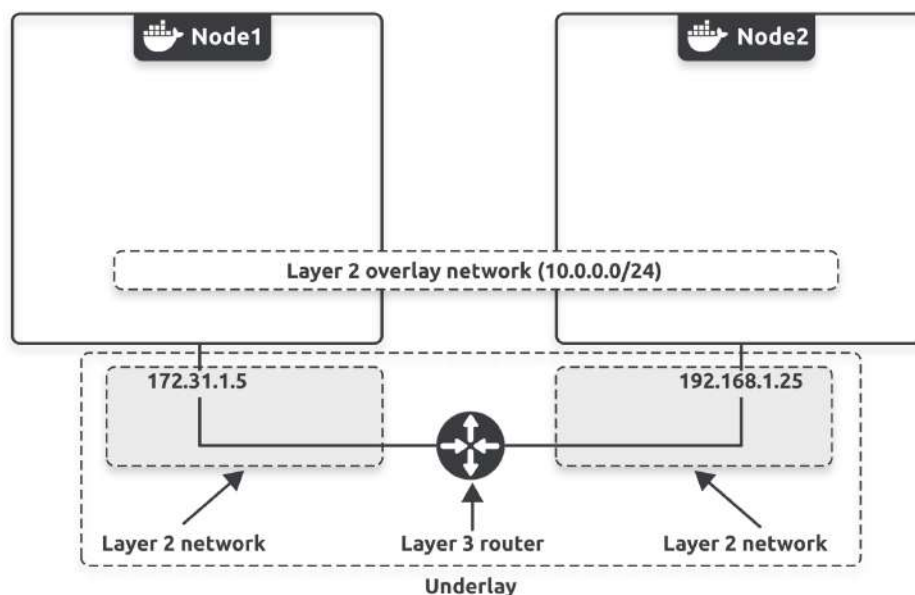

Figure 14.4

Fortunately, VXLAN is an *encapsulation* technology and, therefore, transparent to existing routers and network infrastructure. This means the routers and other infrastructure in the underlay network see the VXLAN/overlay traffic as regular IP/UDP packets and handle it without requiring changes.

To create the overlay, Docker creates a *VXLAN tunnel* through the underlay networks, and this tunnel is what allows the overlay traffic to flow freely without having to interact with the complexity of the underlay networks.

> **Terminology:** *We use the terms underlay networks or underlay infrastructure to refer to the networks the overlay tunnels through.*

Each end of the VXLAN tunnel is terminated by a *VXLAN Tunnel Endpoint (VTEP)*, and it's this VTEP that encapsulates and de-encapsulates the traffic entering and exiting the tunnel. See Figure 14.5.

Figure 14.5
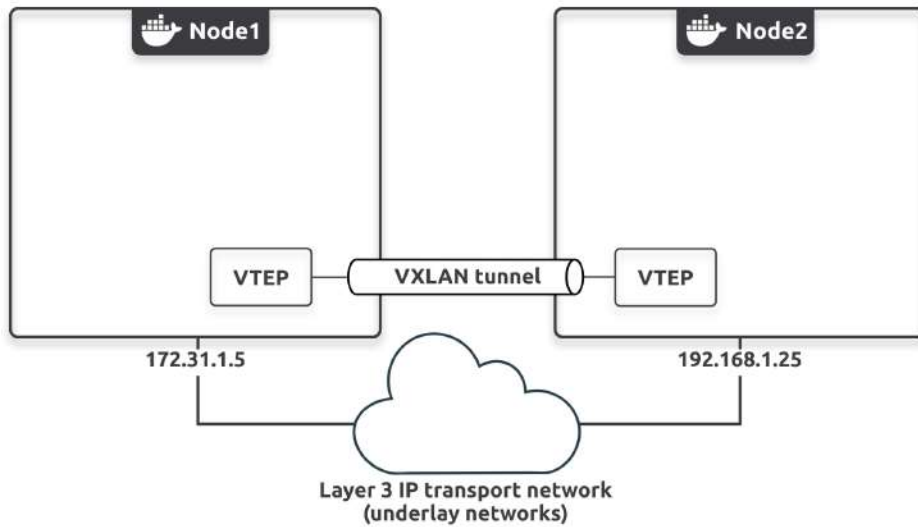
The image shows the layer 3 infrastructure as a cloud for two reasons:

- It can be a lot more complex than the two networks and a single router from the previous diagrams
- The VXLAN tunnel abstracts the complexity and makes it opaque

In reality, the VXLAN tunnel traverses the underlay network. However, I don't show this in the diagram to keep the diagram simple.

Traffic flow example

The hands-on examples from earlier had two hosts connected via an IP network. You deployed an overlay network across both hosts, connected two containers to it, and did a ping test. Let's explain some of the things that happened behind the scenes.

Docker created a new *sandbox* (network namespace) on each host with a new switch called **Br0**. It also created a VTEP with one end connected to the **Br0** virtual switch and the other end connected to the host's network stack. The end in the host's network stack got an IP address on the underlay network that the host is connected to and was bound to UDP port `4789`. Finally, the two VTEPs on each host created a VXLAN tunnel as the backbone for the overlay network.

Figure 14.6 shows the configuration. Remember, the VXLAN tunnel goes through the networks at the bottom of the diagram; I've just drawn it higher up for readability.
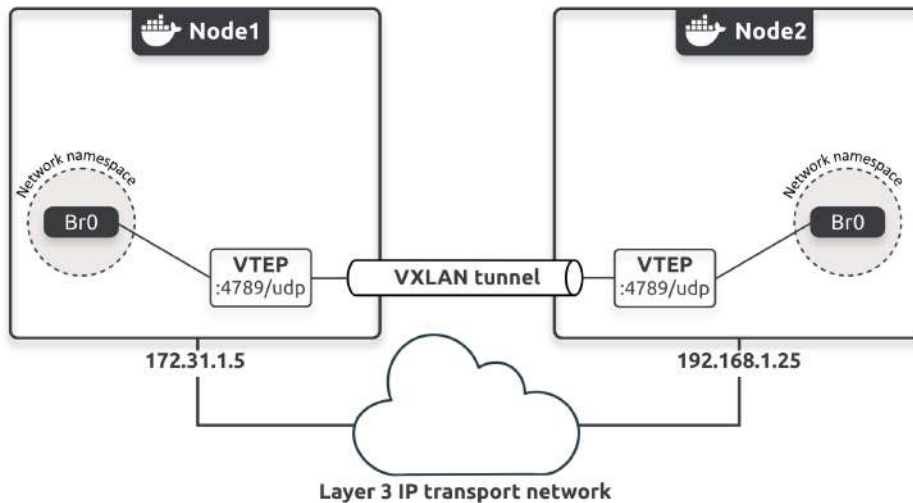
Figure 14.6

At this point, you've created the VXLAN overlay, and you're ready to connect containers.

Docker now creates a virtual Ethernet adapter (**veth**) in each container and connects it to the local **Br0** virtual switch. The final topology looks like Figure 14.7, and although it's complex, you should now see how the containers communicate over the VXLAN overlay despite their hosts being on two separate networks — the overlay is a virtual network tunneled through the underlay networks.
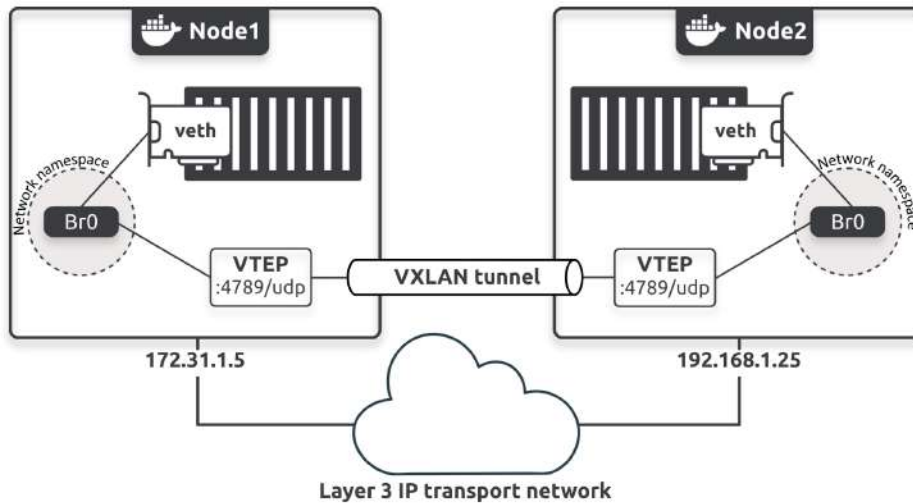
Figure 14.7

Now that you know how Docker creates overlay networks, let's see how the two containers communicate.

> **Warning!** *This section is very technical, and 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"**. We'll also assume **C1** wants to ping **C2** like we did in the practical example earlier. Figure 14.8 shows the full configuration with container names and IPs added.
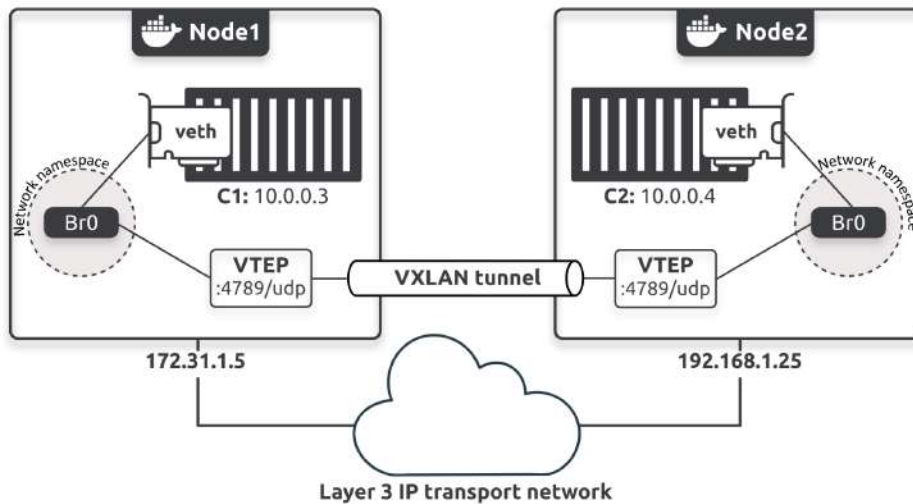
Figure 14.8

C1 initiates a ping request to `10.0.0.4` — the IP address of **C2**.

C1 doesn't have an entry for `10.0.0.4` in its local MAC address table (ARP cache), so it floods the packet on all interfaces, including the veth interface connected to the **Br0** bridge. The **Br0** bridge knows it can forward traffic for `10.0.0.4` to the connected VTEP interface and sends a proxy ARP reply to the container. This results in the veth *learning* how to forward the packet by updating its own MAC table to send all future packets for `10.0.0.4` directly to the local VTEP. The **Br0** switch knew about the **C2** container because Docker propagates details of all new containers to every swarm node via the network's built-in gossip protocol.

Next, the veth in the **C1** container sends the ping to the VTEP interface which encapsulates it for transmission through the VXLAN tunnel. The encapsulation adds a VXLAN header containing a VXLAN network ID (VNID) that maps traffic from VLANs to VXLANs and vice versa — each VLAN gets mapped to its own 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/4789 socket information. This encapsulation allows the packets to be routed across the underlay networks without the underlays knowing anything about VXLAN.

When the packet arrives at node2, the host's kernel sees it's addressed to UDP port `4789` and knows it has a VTEP bound to this socket. This means it sends the packet to the VTEP, which reads the VNID, de-encapsulates it, and sends it to its own local **Br0** switch on the VLAN corresponding to the VNID. From there, it delivers it to the **C2** container.

And that, my friends, is how Docker uses VXLAN to build and operate overlay networks — a whole load of mind-blowing

complexity beautifully hidden behind a single Docker command.

I'm hoping that's enough to get you started and help you when talking to your networking team about the networking aspects of your Docker infrastructure. On the topic of talking to your networking team... 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 a single overlay network with two subnets, and Docker will handle the routing. The following command will create a new overlay called **prod-net** with two subnets. Docker will automatically create two virtual switches called **Br0** and **Br1** inside the *sandbox* and handle all the routing.

```
$ docker network create --subnet=10.1.1.0/24 --subnet=11.1.1.0/24 -d overl
```

Clean up

If you followed along, you'll have created an overlay network called **uber-net** and deployed a service called **test**. You *may* also have created a swarm.

Run the following command to delete the **test** service.

```
$ docker service rm test
```

Delete the **uber-net** network with the following command. You may have to wait a few seconds while Docker deletes the service using it.

```
$ docker network rm uber-net
```

If you no longer need the swarm, you can run a `docker swarm leave -f` command on both nodes. You should run it on **node2** first.

Docker overlay networking – The commands

- `docker network create` tells Docker to create a new network. You use the `-d overlay` flag to use the overlay driver to create an overlay network. You can also pass the `-o encrypted` flag to tell Docker to encrypt network traffic. However, performance may drop in the region of 10%.
- `docker network ls` lists all the container networks visible to a Docker host. Docker hosts running in *swarm mode* only see overlay networks if they run containers attached to the

network. This keeps network-related management traffic to a minimum.

- `docker network inspect` shows detailed information about a particular container network. You can find out the scope, driver, IPv4 and IPv6 info, subnet configuration, IP addresses of connected containers, VXLAN network ID, encryption state, and more.
- `docker network rm` deletes a network.

Chapter Summary

In this chapter, you created a new Docker overlay network and learned about the technologies Docker uses to build them.

15: Volumes and persistent data

Stateful applications that create and manage data are a big part of modern cloud-native apps. This chapter explains how Docker volumes help stateful applications manage their data.

I've split the chapter into the following parts:

- Volumes and persistent data – The TLDR
- Containers without volumes
- Containers with volumes
- The commands

Volumes and persistent data – The TLDR

There are two main types of data — persistent and non-persistent.

Persistent data is the stuff you care about and need to *keep*. It includes things like customer records, financial data, research results, audit data, and even some types of logs. *Non-persistent data* is the stuff you don't care about and don't need to keep. We call applications that create and manage persistent data *stateful apps*, and applications that don't create or manage persistent data *stateless apps*.

Both are important, and Docker has solutions for both.

For stateless apps, Docker creates every container with an area of non-persistent local storage that's tied to the container lifecycle. This storage is suitable for scratch data and temporary files, but you'll lose it when you delete the container or the container terminates.

Docker has *volumes* for stateful apps that create and manage important data. Volumes are separate objects that you mount into containers, and they have their own lifecycles. This means you don't lose the volumes or the data on them when you delete containers. You can even mount volumes into different containers.

That's the TLDR. Let's take a closer look.

Containers without volumes

In the early days of Docker, containers were only good for stateless applications that didn't generate important data. However, despite being *stateless,* many of these apps still needed a place to write temporary scratch data. So, as shown in Figure 15.1, Docker creates containers by stacking read-only image layers and placing a thin layer of local storage on top. The

same technology allows multiple containers to share the same read-only image layers.

Figure 15.1 Ephemeral container storage

This thin layer of local storage is integral to the read-write nature of containers. For example, if an application needs to update existing files or add new files, it makes the changes in the local storage layer, and Docker merges them into the view of the container. However, the local storage is coupled to the container's lifecycle, meaning it gets created when you create the container, and deleted when you delete it. This means it's not a good place for data that you need to keep (persist).

Docker keeps the local storage layer on the Docker host's filesystem, and you'll hear it called various names such as *the thin writeable layer, ephemeral storage, read-write storage*, and *graphdriver storage*. It's usually located in the following locations on your Docker hosts:

- Linux containers: `/var/lib/docker/<storage-driver>/...`
- Windows containers:
  `C:\ProgramData\Docker\windowsfilter\...`

Even though the local storage layer allows you to update live containers, you should never do this. Instead, you should treat containers as *immutable objects* and never change them once deployed. For example, if you need to fix or change the configuration of a live container, you should create and test a new container with the changes and then replace the live container with the new one.

To be clear, applications like databases can change the data they manage. But users and configuration tools should never change the container's *configuration*, such as its network or application configuration. You should always make changes like these in a new container and then replace the old container with the new one.

If your containers don't create persistent data, this thin writable layer of local storage will be fine, and you'll be good to go. However, if your containers create persistent data, you need to read the next section.

Containers with volumes

There are three main reasons you should use *volumes* to handle persistent data in containers:

- Volumes are independent objects that are not tied to the lifecycle of a container
- You can map volumes to specialized external storage systems
- Multiple containers on different Docker hosts can use volumes to access and share the same data

At a high level, you create a volume, then create a container, and finally mount the volume into the container. When you mount it into the volume, you mount it into a directory in the container's filesystem, and anything you write to that directory gets stored in the volume. If you delete the container, the volume and data will still exist. You'll even be able to mount the surviving volume into another container.

Figure 15.2 shows a Docker volume outside the container as a separate object. The volume is mounted into the container's filesystem at `/data`, and anything you write to that directory will be stored on the volume and exist after you delete the container.
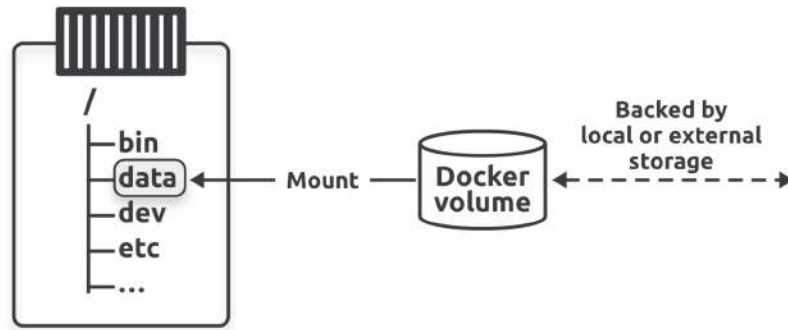
Figure 15.2 High-level view of volumes and containers

The image also shows that you can map the volume to an external storage system or a directory on the Docker host. External storage systems can be cloud services or dedicated storage appliances, but either way, the volume's lifecycle is decoupled from the container. All of the container's other directories use the thin writable layer in the local storage area on the Docker host.

Creating and managing Docker volumes

Volumes are first-class objects in Docker. This means there's a `docker volume` sub-command, and a **volume** resource in the API.

Run 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. And, as the name of the driver suggests, these volumes are only available to containers on the same node as the volume. You can use the `-d` flag to specify a different driver, but you'll need to install the driver first.

Third-party drivers provide advanced features and access to external storage systems such as cloud storage services and on-premises storage systems such as SAN and NAS. Figure 15.3 shows a Docker host connected to an external storage system via a plugin (driver).
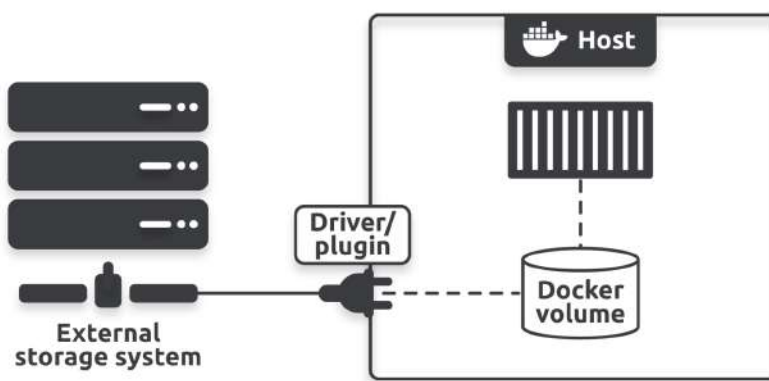
Figure 15.3 Plugging external storage into Docker

Once you've created the volume, you can see it with the `docker volume ls` command and inspect it with the `docker volume inspect` command.

```
$ docker volume ls
DRIVER              VOLUME NAME
local               myvol

$ docker volume inspect myvol
[
    {
        "CreatedAt": "2024-05-15T12:23:14Z",
        "Driver": "local",
        "Labels": null,
        "Mountpoint": "/var/lib/docker/volumes/myvol/_data",
        "Name": "myvol",
        "Options": null,
        "Scope": "local"
    }
]
```

Notice that the `Driver` and `Scope` fields are both set to `local`. This means you created the volume with the **local** driver, and it's only available to containers on this Docker host. `Mountpoint` tells you where the volume exists in the Docker host's filesystem.

By default, Docker gives every volume created with the `local` driver its own directory on the host under `/var/lib/docker/volumes`. This means anyone with access to

the Docker host can bypass the container and access the volume's contents directly in the host's filesystem. You saw this in the Docker Compose chapter when 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. However, that's not a recommended practice.

Now that you've created a volume, you can create containers to use it. However, before you do that, there are two ways to delete Docker volumes:

- `docker volume prune`
- `docker volume rm`

The `docker volume prune --all` command deletes **all volumes** not mounted into a container or service replica, so use it with caution!

The `docker volume rm` command is more precise and lets you specify which volumes to delete.

Neither command will delete a volume in use by a container or service replica.

The **myvol** volume you created isn't used by a container, so you can delete it with either command. Be careful if you use the

`prune` command, as it may also delete other volumes.

```
$ docker volume prune --all

WARNING! This will remove all local volumes not used by at least one conta
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 none of the actions involved a container. This proves that volumes are decoupled from containers.

At this point, you know all the commands to create, list, inspect, and delete Docker volumes. You've even seen how to deploy them via Compose files in the Compose and Swarm stacks chapters. However, you can also deploy volumes via Dockerfiles by using the `VOLUME` instruction. The format is `VOLUME <container-mount-point>`. Interestingly, you cannot specify a host directory when you define volumes in a Dockerfile. This is because host directories can differ depending on your host OS, and you could easily break your builds if you specified a directory that doesn't exist on a host. As a result, defining a volume in a Dockerfile requires you to specify host directories at deployment time.

Using volumes with containers

Let's see how to use volumes with containers.

Run the following command to create a new standalone container called **voltainer** that mounts a volume called **bizvol**.

```
$ docker run -it --name voltainer \
    --mount source=bizvol,target=/vol \
    alpine
```

The command specified the `--mount` flag, telling Docker to mount a volume called **bizvol** into the container at `/vol`. The command completed successfully even though you didn't have a volume called **bizvol**. This raises an important point:

- If you specify a volume that already exists, Docker will use it
- If you specify a volume that does not exist, Docker will create it

In our case, **bizvol** didn't exist, so Docker created it and mounted it into the container.

Type `Ctrl PQ` to return to your local shell, and then list volumes to make sure Docker created it.

```
# <Ctrl-PQ>

$ docker volume ls
DRIVER              VOLUME NAME
local               bizvol
```

Even though volumes are decoupled from containers, Docker won't let you delete this one because it's in use by the **voltainer** container.

Try to delete it.

```
$ docker volume rm bizvol
Error response from daemon: remove bizvol: volume is in use - [b44d3f82...
```

As expected, you can't delete it.

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 write a book review on Amazon" > /vol/file1
```

The command writes some text to a file called **file1** in the **/vol** directory where the volume is mounted.

Run a few commands to make sure the file and data exist.

```
# ls -l /vol
total 4
-rw-r--r-- 1 root  root   50 May 23 08:49 file1

# cat /vol/file1
I promise to write a book review on Amazon
```

Type **exit** to return to your Docker host's shell, and then delete the container with the following commands.

```
# exit

$ docker rm voltainer -f
voltainer
```

Check that Docker deleted the container but kept the volume.

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

$ docker volume ls
DRIVER               VOLUME NAME
local                bizvol
```

As the volume still exists, you can view its contents in the Docker host's local filesystem. Remember, though, that it's not recommended to access volumes directly via the host's

filesystem. We're just showing you how to do it for demonstration and educational reasons.

Run the following commands from your Docker host terminal. They'll show the contents of the volume's directory on your Docker host. The first command will show that the file still exists, and the second will show its contents.

This step won't work on Docker Desktop, as Docker Desktop runs inside a VM. You may have to prefix the commands with `sudo`.

```
$ 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 write a book review on Amazon
```

Great, the volume and the data still exist.

Let's see if you can mount the existing **bizvol** volume into a new service or container. Run the following command to create a new container called **newctr** that mounts bizvol at `/vol`.

```
$ docker run -it \
  --name newctr \
```

```
    --mount source=bizvol,target=/vol \
    alpine sh
```

Your terminal is now attached to the **newctr** container. Check to see if the volume and data are mounted as expected.

```
# cat /vol/file1
I promise to write a book review on Amazon
```

Congratulations. You've created a volume, written some data to it, deleted the original container, mounted it in a second container, and verified the data still exists.

Type `exit` to leave the container and jump over to Amazon to leave the book review you promised to write.

If you left a review, thanks! If you didn't, I'll cry, but I'll live ;-)

Sharing storage across cluster nodes

Integrating Docker with *external storage systems* lets you present shared storage to multiple nodes so that the containers running on different nodes can share the same volumes. These external systems can be cloud storage services or enterprise storage systems in your on-premises data centers. For example, you can present a single storage LUN or NFS share (shared

volume) to multiple Docker hosts so that any container on those hosts can access and share the volume. Figure 15.4 shows an external storage system presenting a shared volume to two Docker nodes. The Docker nodes use the appropriate driver for the external system to make the shared volume available to either or both containers.
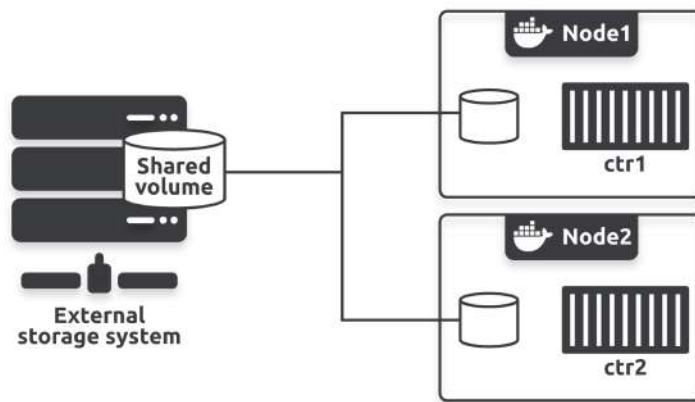

Figure 15.4

Building a shared setup like this requires a lot of things. You need access to specialized storage systems and knowledge of how they work. You also need a volume driver/plugin that works with the external storage system. Finally, you need to know how your applications read and write to the shared storage to avoid potential data corruption.

Potential data corruption

Data corruption is a major concern for any shared storage configuration.

Assume the following example based on Figure 15.4.

The application running in **ctr1** writes an update to the shared volume. However, instead of directly committing the update, it keeps it in a local cache for faster recall. At this point, the application in **ctr1** *thinks* it's written data to the volume. However, before **ctr1** flushes its cache and commits the data to the volume, the app in **ctr2** 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 **ctr2** has. A few seconds later, **ctr1** flushes the data to the volume and overwrites the changes made by the application in **ctr2**. However, neither of the applications is aware of the changes the other has made.

This is why you need to design applications that share data to coordinate updates to shared volumes.

Clean up

If you've been following along, you'll have a container and a volume.

Run the following command to delete the container.

```
$ docker rm
```

Now, run this command to delete the volume.

```
$ docker volume rm bizvol
```

Volumes and persistent data – The Commands

- `docker volume create` creates new volumes. By default, it creates them with the **local** driver, but you can use the `-d` flag to specify a different driver.
- `docker volume ls` lists all volumes on your Docker host.
- `docker volume inspect` shows you detailed volume information. You can use this command to see where a volume exists in the Docker host's filesystem.
- `docker volume prune` deletes **all** volumes not in use by a container or service replica. **Use with caution!**
- `docker volume rm` deletes specific volumes that are not in use.

Chapter Summary

There are two main types of data: *persistent* and *non-persistent.*

Persistent data is data you need to keep, and non-persistent data is data 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 sometimes call this *local storage*, and it's ideal for non-persistent data. However, if your apps create data you need to keep, you should store the data in a Docker volume.

Docker volumes are first-class objects in the Docker API, and you manage them independently of containers using their own `docker volume` sub-command. This means deleting containers doesn't delete the data in their volumes.

A few third-party plugins exist that provide Docker with access to specialized external storage systems.

Volumes are the recommended way to work with persistent data in Docker environments.

16: Docker security

If security is hard, we're less likely to implement it. Fortunately, most of the security in Docker is easy and pre-configured with sensible defaults. This means you get a *moderately secure* experience with zero effort. The defaults are not perfect, but they're a good starting point.

Docker supports all major Linux security technologies and adds some of its own. As such, I've divided the chapter so we cover the Linux security technologies first and finish the chapter covering the Docker technologies:

- Docker security – The TLDR
- Linux security technologies
  - Kernel namespaces
  - Control Groups
  - Capabilities
  - Mandatory Access Control
  - seccomp
- Docker security technologies
  - Swarm security
  - Docker Scout and vulnerability scanning
  - Docker Content Trust
  - Docker secrets

The chapter focuses heavily on Linux, but the sections relating to Docker security technologies apply to Linux and Windows containers.

Docker security – The TLDR

Good security is about layers and defence in depth, and more layers is always better. Fortunately, Docker offers a lot of security layers, including the ones shown in Figure 16.1.
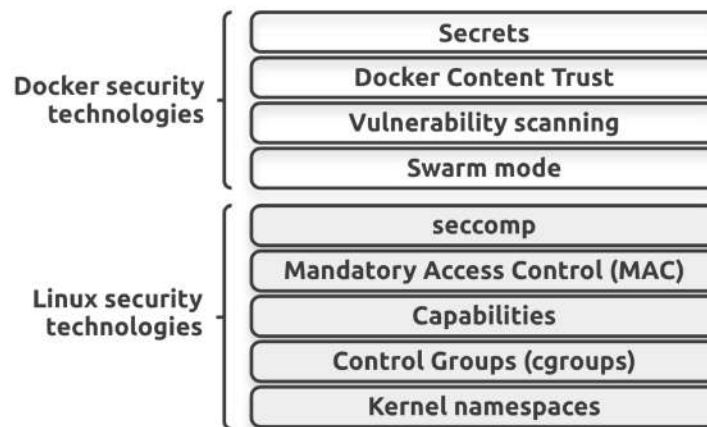

Figure 16.1

As you can see, Docker leverages the common Linux security and workload isolation technologies, including *namespaces, control groups, capabilities, mandatory access control (MAC),* and *seccomp.* It ships with sensible defaults for each one, but you can customize them to your specific requirements.

Docker also has its own security technologies, including **Docker Scout** and **Docker Content Trust**.

Docker Scout offers class-leading *vulnerability scanning* that scans your images, provides detailed reports on known vulnerabilities, and recommends solutions. Docker Content Trust (DCT) lets you cryptographically sign and verify images.

If you use Docker Swarm, you'll also get all of the following that Docker automatically configures: cryptographic node IDs, mutual authentication (TLS), automatic CA configuration and certificate rotation, secure cluster join tokens, an encrypted cluster store, encrypted networks, and more.

Other security-related technologies also exist, but the important thing to know is that Docker works with the major Linux security technologies and adds a few of its own. Sometimes, the Linux security technologies can be complex and challenging to work with, but the native Docker ones are always easy.

Kernel Namespaces

Kernel namespaces, usually shortened to *namespaces*, are the main technology for building containers.

Let's quickly compare namespaces and containers with hypervisors and virtual machines (VM).

Namespaces virtualize operating system constructs such as process trees and filesystems, whereas *hypervisors virtualize physical resources* such as CPUs and disks. In the VM model, hypervisors create virtual machines by grouping virtual CPUs, virtual disks, and virtual network cards so that every VM looks, smells, and feels like a physical machine. In the container model, *namespaces* create virtual operating systems (containers) by grouping virtual process trees, virtual filesystems, and virtual network interfaces so that every container looks, smells, and feels exactly like a regular OS.

At a very high level, namespaces provide lightweight isolation but do not provide a strong security boundary. Compared with VMs, containers are more efficient, but virtual machines are more secure.

Don't worry, though. Platforms like Docker implement additional security technologies, such as cgroups, capabilities, and seccomp, to improve container security.

Namespaces are a tried and tested technology that's existed in the Linux kernel for a very long time. However, they were

complex and hard to work with until Docker came along and hid all the complexity behind the simple `docker run` command and a developer-friendly API.

At the time of writing, every Docker container gets its own instance of the following namespaces:

- Process ID (pid)
- Network (net)
- Filesystem/mount (mnt)
- Inter-process Communication (ipc)
- User (user)
- UTS (uts)

Figure 16.2 shows a single Docker host running two containers. The host OS has its own collection of namespaces we call the *root namespaces*, and each container has its own collection of equivalent isolated namespaces. Applications in containers think they're running on their own host and are unaware of the *root namespaces* or namespaces in other containers.
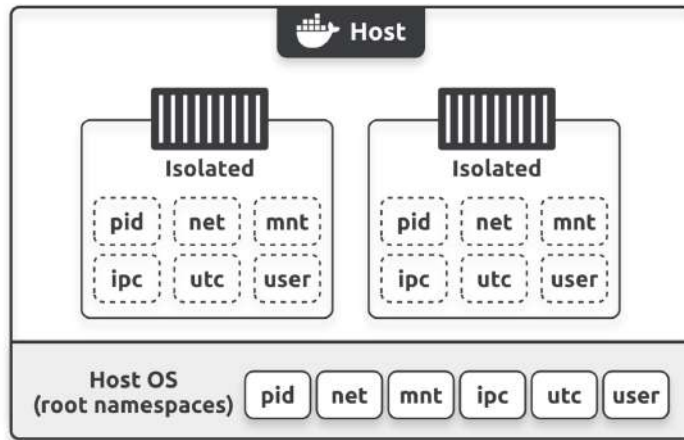
Figure 16.2

Let's briefly look at how Docker uses each namespace:

- **Process ID namespace:** Docker uses the **pid** namespace to give each container its own isolated process tree. This means every container gets its own PID 1 and cannot see or access processes running in other containers. Nor can any container see or access processes running on the host.
- **Network namespace:** Docker uses the **net** namespace to provide each container with an 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 has its own **mnt** namespace with 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 host's filesystem or filesystems in other containers.

- **Inter-process Communication namespace:** Docker uses the **ipc** namespace for shared memory access within a container. It also isolates the container from shared memory on the host and other containers.
- **User namespace:** Docker gives each container its own users that are only valid inside the container. It also lets you map those users to different users on the Docker host. For example, you can map a container's **root** user to a non-root user on the 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 Docker organizes to look like a regular OS. These namespaces provide isolation, but they are not a strong enough security boundary on their own. This is why Docker augments container security with the technologies we're about to discuss.

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 to be isolated, they actually share a lot of things such as water supply, electricity supply, air conditioning, swimming pool, gym, elevators, breakfast bar, and more. Containers are similar — even though they're isolated, they share a lot of common resources such as the host's CPU, RAM, network I/O, and disk I/O.

Docker uses *cgroups* to limit a container's use of these shared resources and prevent any container from consuming them all and causing a denial of service (DoS) attack.

Capabilities

The Linux root user is extremely powerful, and you shouldn't use it to run apps and containers.

However, it's not as simple as running them as non-root users, as most non-root users are so powerless that they are practically useless. What's needed is a way to run apps and containers with the exact set of permissions they need — nothing more, nothing less.

This is where *capabilities* come to the rescue.

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 leverages capabilities so that you can run containers as **root** but strip out all the capabilities you don't need. For example, suppose the only capability your container needs is the ability to bind to low-numbered network ports. In that case, Docker can start the container as root, *drop all root capabilities,* and then add back the **CAP_NET_BIND_SERVICE** capability.

This is a good example of implementing the principle of *least privilege* as you end up with a container that only has the capabilities it needs. Docker also sets restrictions to prevent containers from re-adding dropped capabilities.

Docker ships with sensible out-of-the-box capabilities, but you should configure your own for your production apps and

containers. However, configuring your own requires extensive effort and testing.

Mandatory Access Control systems

Docker works with major Linux MAC technologies such as AppArmor and SELinux.

Depending on your Linux distribution, Docker applies default AppArmor or SELinux profiles to all new containers, and according to the Docker documentation, the default profiles are *moderately protective while providing wide application compatibility*.

You can tell Docker to start containers without these policies, and you can configure your own. However, as with *capabilities*, configuring your own policies is very powerful but requires **a lot** of effort and testing.

seccomp

Docker uses seccomp to limit which syscalls a container can make to the host's kernel.

Syscalls are how applications ask the Linux kernel to perform tasks. At the time of writing, Linux has over 300 syscalls and the default Docker profile disables approximately 40-50.

As per the Docker security philosophy, all new containers get a default seccomp profile configured with *sensible defaults* designed to provide *moderate security without impacting application compatibility.*

As always, you can customize your own seccomp profiles or tell Docker to start containers without one. Unfortunately, the Linux syscall table is long, and configuring custom seccomp policies may be prohibitively complex for some users.

Final thoughts on the Linux security technologies

Docker supports most of the important Linux security technologies and ships with sensible defaults that add security without being too restrictive. Figure 16.3 shows how Docker uses them to build a *defence in depth* security posture with multiple layers.
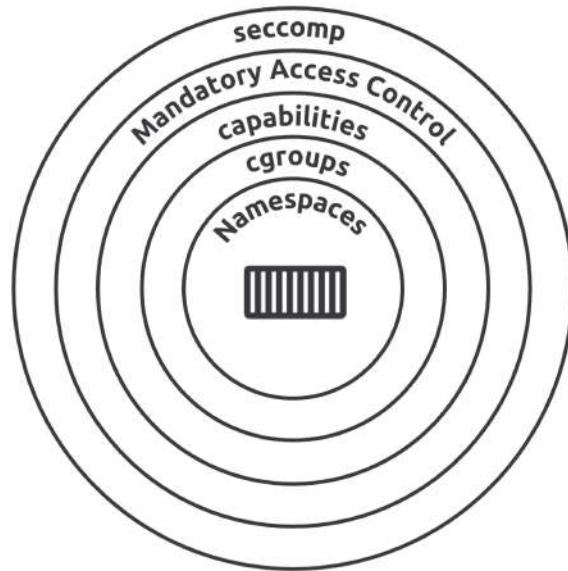
Figure 16.3 - Linux security defense in depth

Some of these technologies require knowledge of the Linux kernel and can be complex to customize. Fortunately, many platforms, including Docker, ship with defaults that are a good place to start.

Docker security technologies

Let's switch our focus to some of the security technologies Docker offers.

Swarm security

Docker Swarm lets you cluster multiple Docker hosts and manage applications declaratively. Every Swarm comprises

manager nodes and *worker nodes* that can be Linux or Windows. Managers host the control plane and are responsible for configuring the cluster and dispatching work tasks. Workers run application containers.

Fortunately, *swarm mode* includes many security features that Docker automatically configures 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 building a secure swarm and configuring some of the security aspects.

If you're following along, you'll need three Docker hosts that can ping each other by name. The examples use three hosts called **mgr1, mgr2**, and **wrk1**.

Configure a secure Swarm

Run the following command from the node you want to be the first manager. We'll run the example from **mgr1**.

```
$ docker swarm init

Swarm initialized: current node (7xam...662z) is now a manager.
```

That's it! You've configured a secure swarm with a cryptographic cluster ID, an encrypted cluster store, a certificate authority (CA) with a 90-day certificate rotation policy, a set of secure join tokens to use when adding new managers and workers, and configured the current manager with a client certificate for mutual TLS — all with a single command!

The CA is for internal Swarm security, and you should be careful using it for anything else.

Figure 16.4 shows the current swarm configuration. Some of the details may be different in your lab.
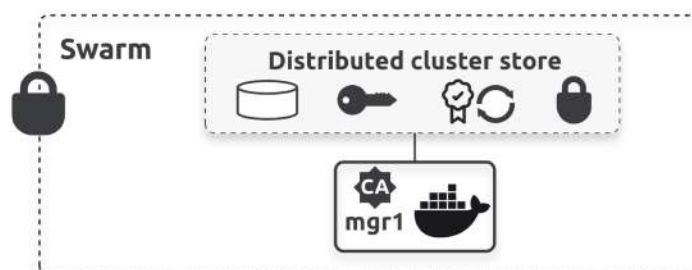
Figure 16.4

Let's join **mgr2** as an additional manager.

Joining new managers is a two-step process:

- Extract the secure join token
- Execute a `docker swarm join` command with the join token on the node you're adding

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 you the full command and join token to run on **mgr2**. 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-1-1dmtwu...r17stb-2axi5...8p7glz 172.31

This node joined a swarm as a manager.
```

List the nodes in your swarm.

```
$ docker node ls
ID               HOSTNAME   STATUS    AVAILABILITY    MANAGER STATUS
7xamk...ge662z   mgr1       Ready     Active          Leader
i0ue4...zcjm7f * mgr2       Ready     Active          Reachable
```

You now have a two-node swarm with **mgr1** and **mgr2** as managers. Both have access to the cluster store and are configured with client certificates for mutual TLS.

In the real world, you'll always run three or five managers for high availability.

Figure 16.5 shows the updated swarm with both managers.
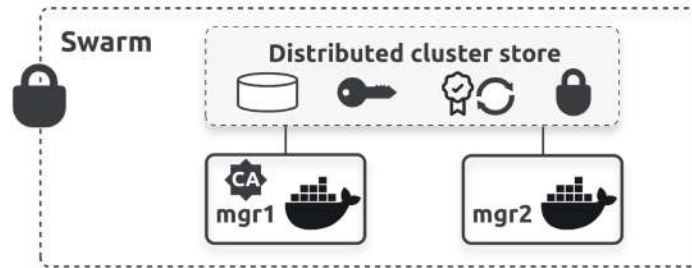
Figure 16.5

Adding worker nodes 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 command and token.

```
$ docker swarm join-token worker

To add a worker to this swarm, run the following command:

    docker swarm join --token \
    SWMTKN-1-1dmtw...17stb-ehp8g...w738q \
    172.31.5.251:2377
```

Copy the command and run it on **wrk1**:

```
$ docker swarm join --token SWMTKN-1-1dmtw...17stb-ehp8g...w738q 172.31.5.

This node joined a swarm as a worker.
```

Run another `docker node ls` from either of your 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
```

Your swarm has two managers and a worker. The managers are configured for high availability (HA) and the cluster store is replicated to both. The worker node is part of the swarm but cannot access the cluster store. Figure 16.6 shows the final configuration.
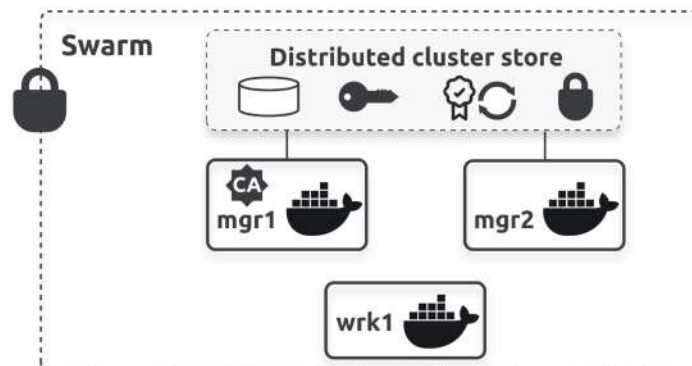
Figure 16.6

Now that you've built a secure Swarm, let's examine some of the security aspects.

Swarm join tokens

The only requirement for joining managers and workers is possession of the secure join token. This means you should keep them safe and never post them on public repos or even internal repos that are not restricted.

Every swarm maintains two distinct join tokens:

- Manager token
- Worker token

Every join token has four distinct fields separated by dashes (`-`):

- `PREFIX - VERSION - SWARM ID - TOKEN`

The prefix is always `SWMTKN` and allows you to pattern-match against it to 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 the worker or manager token.

As you can see in the following table, the manager and worker tokens for any given swarm are identical except for the final `TOKEN` field.

Role	Prefix	Version	Swarm ID	Token
Manager	SWMTKN	1	1dmtwusdc… r17stb	**2axi5**
Worker	SWMTKN	1	1dmtwusdc… r17stb	**ehp8g**

If you suspect either of your join tokens are compromised, you can revoke them and issue new ones with a single command. The following example revokes the existing *manager* token and issues a new one.

```
$ docker swarm join-token --rotate manager

Successfully rotated manager join token.
```

Existing managers are unaffected, but you can only add new ones with the new token.

As expected, the last field is the only difference between the old and new tokens.

Docker keeps a copy of join tokens in the encrypted cluster store.

TLS and mutual authentication

Docker issues every manager and worker with a client certificate that they use for mutual authentication. It identifies the node, the swarm 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 2024 GMT
            Not After : Aug 21 09:23:00 2024 GMT
        Subject: O = tcz3w1t7yu0s4wacovn1rtgp4, OU = swarm-manager,
            CN = 2gxz2h1f0rnmc3atm35qcd1zw
        Subject Public Key Info:
<SNIP>
```

As shown in Figure 16.7, the `Subject` field 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.

You can also see the certificate rotation period in the `Validity` section.
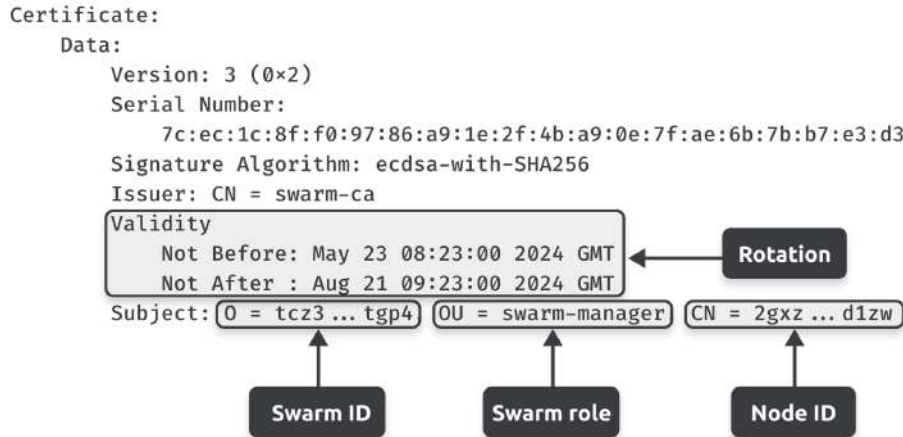
Figure 16.7

You can match these values to the corresponding values from a `docker info` command.

```
$ docker info
<SNIP>
 Swarm: active
  NodeID: 2gxz2h1f0rnmc3atm35qcd1zw     <<---- Relates to the CN field
  Is Manager: true                      <<---- Relates to the OU field
  ClusterID: tcz3w1t7yu0s4wacovn1rtgp4  <<---- Relates to the O field
 <SNIP>
  CA Configuration:
  Expiry Duration: 3 months             <<---- Relates to the validity blo
  Force Rotate: 0
  Root Rotation In Progress: false
 <SNIP>
```

Swarm CA configuration

You can use the **docker swarm update** command to configure the certificate rotation period. The following example changes it to 30 days.

```
$ docker swarm update --cert-expiry 720h
```

Swarm allows nodes to renew certificates early so that all nodes don't update at exactly the same time.

You can configure a new swarm to use an external CA by passing the **--external-ca** flag to **docker swarm init** command, and you can use the **docker swarm ca** command to manage other CA-related settings.

```
$ docker swarm ca --help

Usage:  docker swarm ca [OPTIONS]

Display and rotate the root CA

Options:
      --ca-cert pem-file          Path to the PEM-formatted root CA certif
                                  for the new cluster
      --ca-key pem-file           Path to the PEM-formatted root CA key to
                                  new cluster
      --cert-expiry duration      Validity period for node certificates (r
                                  (default 2160h0m0s)
  -d, --detach                    Exit immediately instead of waiting for
                                  to converge
      --external-ca external-ca   Specifications of one or more certificat
```

```
    -q, --quiet                          Suppress progress output
        --rotate                         Rotate the swarm CA - if no certificate
                                         provided, new ones will be generated
```

The cluster store

The cluster store is where Docker keeps the configuration and state of a swarm. It's also critical to other Docker technologies, such as overlay networks and secrets. This is why overlay networks and many other advanced security features only work in swarm mode.

The cluster store is based on the popular **etcd** distributed database and is automatically encrypted and replicated to all managers.

Docker handles day-to-day maintenance, but you should implement strong backup and recovery procedures for production clusters.

That's enough about swarm mode security for now. Let's look at some Docker security technologies that don't require swarm mode.

Docker Scout and vulnerability scanning

Every container runs multiple software packages that are susceptible to bugs and vulnerabilities that malicious actors can exploit.

Image scanning analyzes your images and produces a detailed list of all the software packages it uses. We call this list a *software bill of materials (SBOM),* and the image scanning system compares the SBOM against databases of known vulnerabilities and provides a report of vulnerabilities in your software. Most vulnerability scanners will rank the vulnerabilities and provide advice on fixes.

Vulnerability scanning is now an integral part of most software supply chains.

Docker Scout is Docker's native scanning platform and works with Docker Hub, Docker Desktop, the Docker CLI, and even has its own Docker Scout Dashboard. However, it's a subscription-based service.

Other scanning platforms are available, but most of these also require some form of subscription.

If you're using Docker Desktop, you can run the following command to see an example of Docker Scout.

```
$ docker scout quickview nigelpoulton/tu-demo:latest

    ✓ Provenance obtained from attestation
    ✓ Pulled
    ✓ Image stored for indexing
    ✓ Indexed 66 packages

  Target                  |  nigelpoulton/tu-demo:latest  |    0C      4H      2M
    digest                |  b4210d0aa52f                  |
  Base image              |  python:3-alpine              |    0C      2H      1M
  Updated base image      |  python:3.11-alpine           |    0C      1H      1M
                          |                               |
◄ ▐▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬                 ►
```

The output shows zero critical vulnerabilities (0C), four high
(4H), two medium (2M), and zero low (0L).

You can also run a `docker scout cves` command to get more
detailed information, including remediation advice.

```
$ docker scout cves nigelpoulton/tu-demo:latest

    ✓ SBOM of image already cached, 66 packages indexed
    ✗ Detected 6 vulnerable packages with a total of 8 vulnerabilities
## Overview
                        |          Analyzed Image
    ────────────────────┼──────────────────────────────────────
  Target                |    nigelpoulton/tu-demo:latest
    digest              |    b4210d0aa52f
    platform            |  linux/arm64
    vulnerabilities     |     0C      4H      2M      0L
    size                |  26 MB
    packages            |  66
```

```
## Packages and Vulnerabilities
   0C     1H     1M     0L   expat 2.5.0-r2
pkg:apk/alpine/expat@2.5.0-r2?os_name=alpine&os_version=3.19

    ✗ HIGH CVE-2023-52425
      https://scout.docker.com/v/CVE-2023-52425
      Affected range : <2.6.0-r0
      Fixed version  : 2.6.0-r0

   ✗ MEDIUM CVE-2023-52426
      https://scout.docker.com/v/CVE-2023-52426
      Affected range : <2.6.0-r0
      Fixed version  : 2.6.0-r0
<Snip>
```

I've snipped the output, so it only shows some of the vulnerabilities. However, even from the snipped output in the book, you can see:

- Scout has scanned 66 packages and detected several vulnerabilities
- We're using version `2.5.0-r2` of the **expat** package which has one high (1H) and one medium (1M) vulnerability
- The high vulnerability is listed as `CVE-2023-52425` and the medium as **CVE-2023-52426**
- The report includes links to Scout reports containing more info on each vulnerability
- Scout recommends updating to **expat** version `2.6.0-r0` which contains fixes for both

Figure 16.8 shows what it looks like in in Docker Desktop, and you get similar integrations and views in Docker Hub.
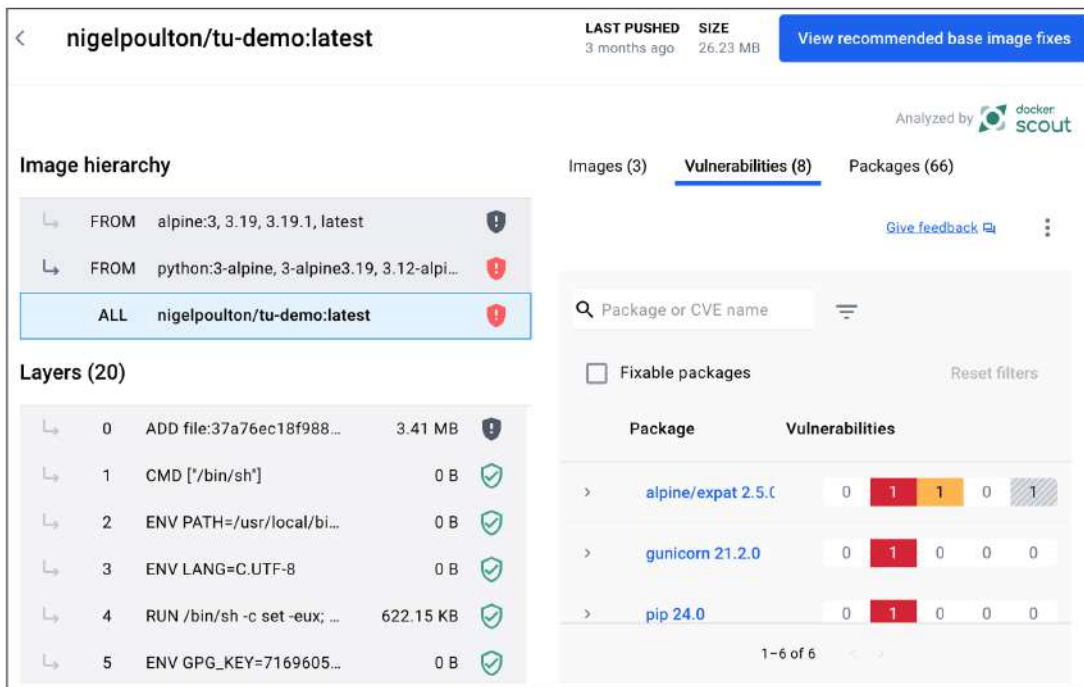
Figure 16.8 - Docker Scout integration with Docker Desktop

If you subscribe to Docker Scout, you can use the `scout.docker.com` portal to configure policies and integrations with Docker Hub and other registries.

As good as vulnerability scanning is, it only scans images and doesn't detect security problems with networks, nodes, or orchestrators. Also, not all image scanners are equal. For example, the best ones perform deep binary-level scans,

whereas others may just look at package names and do not inspect content closely.

In summary, scanning tools are great for inspecting your images and detecting known vulnerabilities. Beware though, with great knowledge comes great responsibility — once you're aware of vulnerabilities, you're responsible for mitigating or fixing them.

Signing and verifying images with Docker Content Trust

Docker Content Trust (DCT) makes it simple for you to verify the integrity and publisher of images and is especially important when you're pulling images over untrusted networks such as the internet.

At a high level, DCT lets you sign your images when you push them to registries like Docker Hub. It also lets you verify the images you pull and run as containers.

Figure 16.9 shows the high-level process.
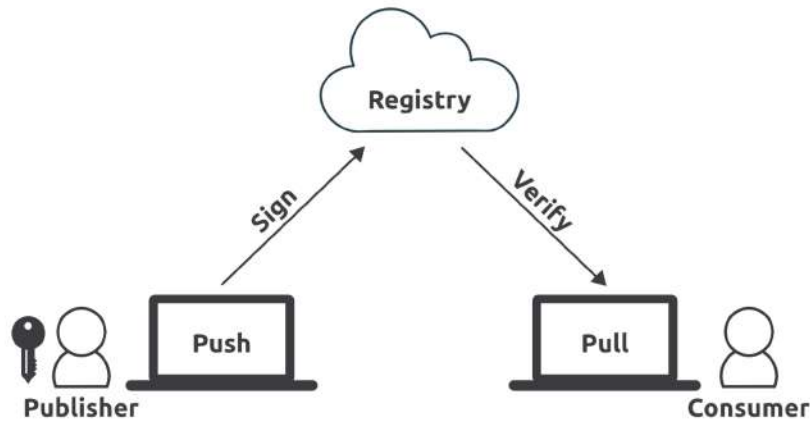
Figure 16.9 - Docker Content Trust image signing and verification

You can also use DCT to provide *context,* such as whether or not a developer has signed an image 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 walk you through configuring Docker Content Trust, signing and pushing an image, and then pulling the signed image.

If you plan on following along, you'll need a cryptographic key pair. If you don't already have one, you can run the following `docker trust` command to generate one. The command generates a new key pair called **nigel** and loads it to the local trust store ready for use. It will prompt you to enter a passphrase; don't forget it :-)

```
$ 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.... key available: /Users/ni
```

If you already have a key pair, you can import and load it with `docker trust key load key.pem --name nigel`.

The next step is associating your key pair with the image repository to which you'll push signed images. This example associates the `nigel.pub` key with the `nigelpoulton/ddd-trust` repo on Docker Hub. 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
```

Now that you've loaded the key pair and associated it with a repository, the final step is to sign an image and push it to the repo.

The following command signs a local image called `nigelpoulton/ddd-trust:signed` and pushes it to Docker Hub. Your image will have a different name and you'll push it to a different repo.

```
$ docker trust sign nigelpoulton/ddd-trust:signed
Signing and pushing trust data for local image nigelpoulton/ddd-trust:sign
The push refers to repository [docker.io/nigelpoulton/ddd-trust]
6495b414566f: Mounted from nigelpoulton/ddd-book
798676f7ef8b: Mounted from nigelpoulton/ddd-book
bca4290a9639: Mounted from nigelpoulton/ddd-book
28ad2149d870: Mounted from nigelpoulton/ddd-book
4f4fb700ef54: Mounted from nigelpoulton/ddd-book
5e1fc7f5df34: Mounted from nigelpoulton/ddd-book
signed: digest: sha256:b65f9a1aa4e670bbafd0fbb91281ea95f9cdc5728aa546579e2
Signing and pushing trust metadata
Enter passphrase for nigel key with ID 92330ea:
Successfully signed docker.io/nigelpoulton/ddd-trust:signed
```

The push operation creates the repo on Docker Hub and then signs and pushes the image. You can view the repo on Docker Hub, and you can run the following command to inspect its signing data.

```
$ docker trust inspect nigelpoulton/ddd-trust:signed --pretty

Signatures for nigelpoulton/ddd-trust:signed
  SIGNED TAG    DIGEST                              SIGNERS
  signed        30e6d35703c578ee...4fcbbcbb0f281    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 export the `DOCKER_CONTENT_TRUST` variable with a value of `1` to force a Docker host to sign and verify all images.

```
$ export DOCKER_CONTENT_TRUST=1
```

Once enabled, you won't be able to pull and work with unsigned images.

Test it by trying to pull an unsigned image.

```
$ docker pull nigelpoulton/ddd-book:web0.2
Error: remote trust data does not exist for docker.io/nigelpoulton/ddd-boo
does not have trust data for docker.io/nigelpoulton/ddd-book
```

You can no longer pull images without trust data!

Delete the local copy of the image you just signed and pushed so that you can try pulling it from Docker Hub. Your image name will be different.

```
$ docker rmi nigelpoulton/ddd-trust:signed
Untagged: nigelpoulton/ddd-trust:signed@sha256...
<Snip>
```

Now, try pulling the image.

```
$ docker pull nigelpoulton/ddd-trust:signed
Pull (1 of 1): nigelpoulton/ddd-trust:signed@sha256:30e6...
docker.io/nigelpoulton/ddd-trust@sha256:30e6... Pulling from nigelpoulton/
08409d417260: Pull complete
Digest: sha256:30e6d35703c578ee703230b9dc87ada2ba958c1928615ac8a674fcbbcbb
Status: Downloaded newer image for nigelpoulton/ddd-trust@sha256:30e6...
Tagging nigelpoulton/ddd-trust@sha256:30e6d... as nigelpoulton/ddd-trust:s
docker.io/nigelpoulton/ddd-trust:signed
```

The pull worked because the image has valid trust data.

In summary, Docker Content Trust is an important technology
that helps you verify the integrity of the images you pull and
run. It's simple to configure in its basic form, but more
advanced features, such as *context*, can be more complex.

Docker Secrets

Most applications leverage sensitive data such as passwords,
certificates, and SSH keys. Fortunately, Docker lets you wrap
them inside *secrets* to keep them secure.

Note: *Secrets only work in swarm mode as they leverage the cluster store.*

Behind the scenes, Docker encrypts secrets when they're *at rest* in the cluster store and while they're *in flight* on the network. It also uses *in-memory filesystems* to mount secrets into containers and operates a least-privilege model, where secrets are only available to services that have been explicitly granted access. There's even a `docker secret` command.

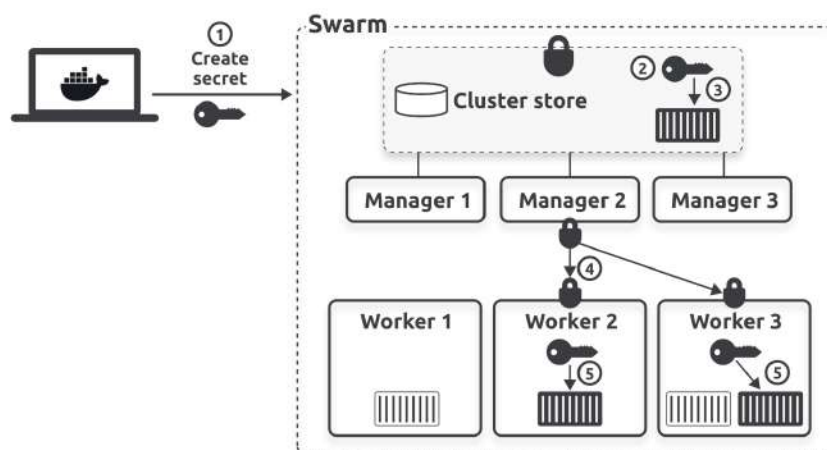Figure 16.10 shows the high-level workflow of creating a secret and deploying it to service replicas:

Figure 16.10 - Secret workflow

Let's go through the five steps in the diagram. I've used a key symbol to show the secret, and it's only available to the dark containers.

1. You create the secret
2. Docker stores it in the encrypted cluster store
3. You create a service (the dark containers) and grant it access to the secret
4. Docker encrypts the secret when sending it over the network to service replicas
5. Docker mounts the secret into service replicas as an unencrypted file in an in-memory filesystem

The light-colored containers are part of a different service and cannot access the secret.

As soon as replicas using the secret terminate, Docker destroys the in-memory filesystem and flushes the secret from the node.

Docker mounts secrets in their unencrypted form so that applications can use them without needing keys to decrypt them.

You can create and manage secrets with the `docker secret` command and attach them to services by passing the `--secret` flag to the `docker service create` command.

Clean up

If you've followed along, you've created a swarm, added a signer, created a new repo on Docker Hub, and exported an environment variable to sign and verify images automatically.

Run the following command to disable Docker Content Trust. You'll need to run it on every node where you enabled Docker Content Trust.

```
$ unset DOCKER_CONTENT_TRUST
```

Remove the signer from the repository you created. Your signer and repository will have different names.

```
$ docker trust signer remove nigel nigelpoulton/ddd-trust
Removing signer "nigel" from nigelpoulton/ddd-trust...
all signed tags are currently revoked, use docker trust sign to fix
```

You may also want to delete the repositories you created on Docker Hub and delete the local key files on your system (usually a .pub file in your home directory)

Delete the swarm by running the following command on all swarm nodes. You should run it on the swarm managers last.

```
$ docker swarm leave -f
```

Chapter Summary

You can configure Docker 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 and even disable them.

In addition to the 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. Docker Scout performs binary-level image scans and provides detailed reports of known vulnerabilities and suggested fixes. Docker Content Trust lets you sign and verify images, and Docker secrets allow you to share sensitive data with swarm services.

What next

Thank you so much for reading my book. You're on your way to mastering Docker and containers, and you've worked with a multi-container AI chatbot app.

Get involved with the community

There's a vibrant cloud-native 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 understand 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 Kubernetes books.

Feedback and reviews

Books live and die by Amazon reviews and stars.

I live and breathe this book, ensuring you get the most up-to-date content that's easy to read and understand. So, **please take a moment to leave a kind review on Amazon or Goodreads.**

Also, ping me at `ddd@nigelpoulton.com` if you want to suggest content or fixes for future editions.

Connect with me

Finally, thanks for reading my book. Feel free to connect with me on any of the usual platforms to discuss Docker, Kubernetes, Wasm, AI, and other technologies.

- LinkedIn: [Nigel Poulton](#)

- Web: [nigelpoulton.com](#)

- BlueSky: [@nigelpoulton](#)

- X: [@nigelpoulton](#)

- Email: ddd@nigelpoulton.com

Terminology

This glossary defines some of the most common Docker and container-related terms used in the book.

If you think I've missed anything important, ping me at **ddd@nigelpoulton.com**.

Term	Definition (according to Nigel)
API	Application Programming Interface. In the case of Docker, all resources are defined in the Docker API, which is RESTful and exposed via the *Docker Daemon*.
Base image	The first layer of all container images. Created by the Dockerfile `FROM` instruction and usually contains a minimal set of OS constructs required by an application.
Build	The process of building a new container image. Docker *builds* images by stepping through a set of instructions defined in a Dockerfile.

Term	Definition (according to Nigel)
Build Cloud	A subscription service that performs fast and efficient image builds in Docker's cloud infrastructure. It allows you to share a common build-cache among teams for very fast builds.
BuildKit	Docker's build engine that implements advanced build features such as advanced caching, multi-stage builds, and multi-architecture builds.

Term	Definition (according to Nigel)
Buildx	Docker's latest and greatest build client that supports all the latest features of BuildKit, such as multi-stage builds and multi-architecture images. Buildx has been Docker's default build client since Docker Engine v23.0 and Docker Desktop v4.19.
Capability	Linux kernel technology used by Docker to create user accounts with the precise set of system access they need.

Term	Definition (according to Nigel)
Chatbot	A computer program that can participate in text-based human conversations and often indistinguishable from a human.
Cloud native	A loaded term that means different things to different people. Cloud native is a way of designing, building, and working with modern applications and infrastructure. I consider an application to be *cloud native* if it can self-heal, scale on-demand, perform rolling updates, and versioned rollbacks.

Term	Definition (according to Nigel)
Cluster store	Docker Swarm's distributed database that holds the state of the cluster and apps. Based on the **etcd** distributed database, it is automatically encrypted and automatically distributed across all swarm managers for high availability.
Compose	An open specification for defining, deploying, and managing multi-container microservices apps. Docker implements the Compose spec and provides the `docker compose` command to make it easy to work with Compose apps.

Term	Definition (according to Nigel)
Container	A container is a collection of kernel namespaces organized to look, smell, and feel like a regular operating system. Each container runs a single application, and containers are smaller, faster, and more portable than virtual machines. We sometimes call them *Docker containers* or *OCI containers*
Container Network Model	Pluggable interface enabling different network topologies and architectures. Third parties provide CNM plugins for overlay networks and BGP networks, as well as various implementations of each.

Term	Definition (according to Nigel)
Container runtime	Software running on every Docker node responsible for pulling container images, starting containers, stopping containers, and other low-level container operations. Docker uses two runtimes that work together: **containerd** is Docker's high-level runtime that manages lifecycle events such as starting and stopping containers, whereas **runc** is Docker's low-level runtime that interfaces with kernel constructs such as namespaces and cgroups.
containerd	Industry-standard container runtime used by Docker and most Kubernetes clusters. Donated to the CNCF by Docker, Inc. Pronounced "container dee".

Term	Definition (according to Nigel)
Containerize	The process of packaging an application and all dependencies into a container image.
Control Groups (cgroups)	Linux kernel feature that Docker uses to limit the amount of host CPU, RAM, disk, and network resources a container uses.
Desired state	How your cluster and applications should be. For example, the *desired state* of an application microservice might be five replicas of xyz container listening on port 8080/tcp. Vital to reconciliation.

Term	Definition (according to Nigel)
Docker	Platform that makes it easy to work with containerized apps. It allows you to build images, as well as run and manage standalone containers and multi-container apps.
Docker Debug	Docker CLI plugin that lets you easily debug slim images and containers that don't ship with any debugging tools.

Term	Definition (according to Nigel)
Docker Desktop	Desktop application for Linux, Mac, and Windows that makes working with Docker easy. It has a slick UI and many advanced features like image management, vulnerability scanning, and Wasm support.
Docker Hub	High-performance OCI-compliant image registry. Docker Hub has over 57PB of storage and handles an average of 30K requests per second.
Docker, Inc.	US-based technology company making it easy for developers to build, ship, and run containerized applications. The company behind the Docker platform.

Term	Definition (according to Nigel)
Docker init	A new Docker CLI plugin that creates high-fidelity Dockerfiles and makes it easy to scaffold Compose apps.
Docker Scout	Docker's native vulnerability scanning service. Scout is a subscription service that integrates with the Docker CLI, Docker Desktop, Docker Hub, and other image registries.
Dockerfile	Plain text file with instructions telling Docker how to build an application into a container image.

Term	Definition (according to Nigel)
etcd	The open-source distributed database used by Docker Swarm.
Image	Archive containing application code, all dependencies, and the metadata required to start a single application as a container. We sometimes call them *OCI images*, *container images*, or *Docker images*.
Ingress network	Hidden network on all Docker Swarm clusters used to publish services to external clients.

Term	Definition (according to Nigel)
Kernel namespace	Feature of the Linux kernel used by Docker to isolate containers from processes running on the host and in other containers.
Large Language Model (LLM)	An AI application that can participate in human conversations and create human-like answers and ideas. The book's AI chatbot app uses an LLM that is trained as a coding assistant that can help answer coding questions and provide coding samples.

Term	Definition (according to Nigel)
Layer	Image layers contain modifications to the base image or the layer below them. Docker builds images by stacking layers, each containing changes to the layer below it. A simple example is a base layer that has basic OS constructs, followed by a layer with the application. The two combined layers create the image with the OS and app.
libcontainer	A Go library that uses namespaces, cgroups, and capabilities to build containers. Docker uses libcontainer via the **runc** low-level runtime that is a CLI wrapper around libcontainer.

Term	Definition (according to Nigel)
libnetwork	The Go library used by Docker to create and manage container networks.
Microservices	Design pattern for modern applications. Individual application features are developed as their own small applications (microservices/containers) and communicate via APIs. They work together to form a useful application.

Term	Definition (according to Nigel)
Multi-architecture builds (sometimes called multi-platform builds)	Allows you to build images for multiple architectures and platforms with a single `docker build` command. For example, you can run a single `docker build` command on an AMD-based Windows system to build an AMD image **and** an ARM image.

Term	Definition (according to Nigel)
Multi-stage build	Allows you to create very small images (slim images). You build your images in stages and only carry forward the necessary artifacts for each next stage. Each build stage is represented by its own `FROM` instruction in your Dockerfile, and later build stages use the `COPY --from` instruction to use artifacts from previous stages and leave everything else behind.
Observed state	Also known as *current state* or *actual state*. The most up-to-date view of the cluster and running applications. Docker Swarm is always working to make *observed state* match *desired state*.

Term	Definition (according to Nigel)
Ollama	Open-source runtime for running LLMs locally. A bit like Docker for LLMs — Ollama can pull and push LLMs and run them locally on your computer.
Open Container Initiative (OCI)	Lightweight governance body responsible for creating and maintaining standards for low-level container technologies such as images, runtimes, and registries. Docker creates OCI-compliant images, implements an OCI-compliant runtime, and Docker Hub is an OCI-compliant registry.

Term	Definition (according to Nigel)
Orchestrator	Software that deploys and manages apps. Docker Swarm and Kubernetes are examples of orchestrators that manage microservices apps, keep them healthy, scale them up and down, and more...
Overlay network	A large flat layer-2 network that spans multiple swarm nodes. All containers on the same overlay network can communicate with each other even if they're on different Docker hosts that are on different networks. The built-in **overlay** driver creates overlay networks using advanced VXLAN technologies. They are only supported by Docker Swarm.

Term	Definition (according to Nigel)
Push	Upload an image to a registry.
Pull	Download an image from a registry.
Reconciliation	The process of watching the state of an application and ensuring observed state matches desired state. Docker Swarm runs reconciliation loops, ensuring applications run how you want them to.

Term	Definition (according to Nigel)
Registry	Central place for storing and retrieving images. We sometimes call them *OCI registries, container registries,* or *Docker registries*.
Repository	An area of a registry where you store related container images. You can set access controls per repository.
Seccomp	Secure computing Linux kernel feature used by Docker to restrict the syscalls available to a container.

Term	Definition (according to Nigel)
Secret	The way Docker Swarm lets you inject sensitive data into a container at run-time.
Service	Capital "S" is a Docker Swarm feature that augments containers with self-healing, scaling, rollouts, and rollbacks.
Spin	Framework that makes it easy to build, deploy, and run Wasm apps. Docker Desktop ships with the **spin** runtime. Created by Fermyon Technologies, Inc.

Term	Definition (according to Nigel)
Swarm (also known as Docker Swarm)	Docker's native orchestration platform. A lightweight and easy alternative to Kubernetes.
Volume	Where containers store important data they need to keep. You can create and delete volumes independently from containers.
Wasm (WebAssembly)	New virtual machine architecture that is smaller, faster, more portable, and more secure than traditional containers. Wasm apps run anywhere with a Wasm runtime.

Term	Definition (according to Nigel)
YAML	Yet Another Markup Language. You write Compose files in YAML. It's a superset of JSON.

More from the author

QUICK START
KUBERNETES
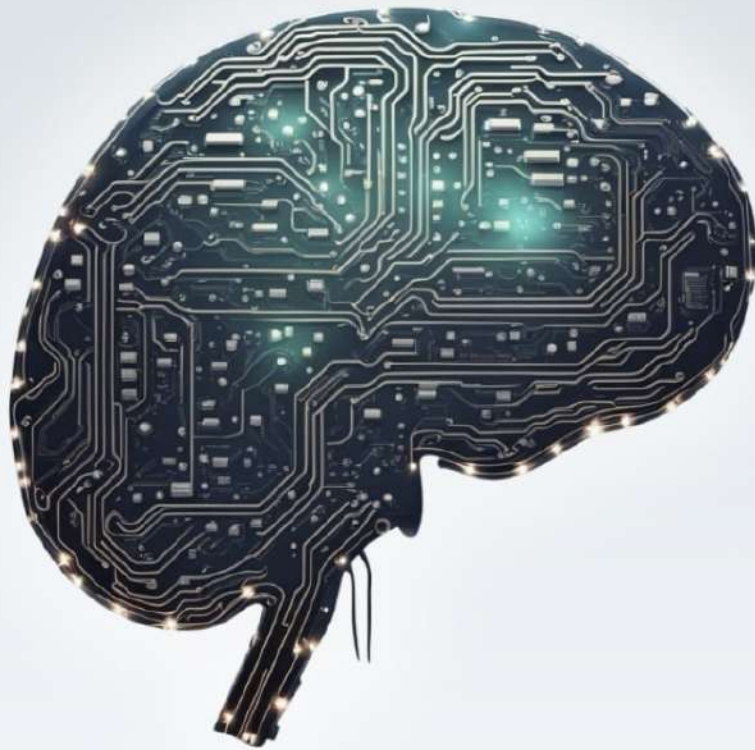
Nigel Poulton

```
$ kubectl get pod jump-pod -o yaml
apiVersion: v1
kind: Pod
metadata:
  name: jump-pod
  namespace: default
spec:
  containers:
  - image: nigelpoulton/curl:1.0
    imagePullPolicy: IfNotPresent
    name: jump-ctr
    stdin: true
    tty: true
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: default-token-2g29h
      readOnly: true
  dnsPolicy: ClusterFirst
```

THE KUBERNETES BOOK

Nigel Poulton
& Pushkar Joglekar

AI
EXPLAINED
Facts, Fiction & Future

Nigel Poulton