Early Access

The Ultimate **Docker** Container Book

Build, ship, deploy, and scale containerized applications with Docker, Kubernetes, and the cloud

Fourth Edition

DR. GABRIEL N. SCHENKER

‹packt›

The Ultimate Docker Container Book

Fourth Edition

Build, ship, deploy, and scale containerized applications with Docker, Kubernetes, and the cloud

Dr. Gabriel N. Schenker

‹packt›

The Ultimate Docker Container Book

Fourth Edition

Early Access Publication: The Ultimate Docker Container Book

Early Access Production reference: B32389

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80580-439-0

www.packtpub.com

Table of Contents

Chapter 3: Mastering Containers

Chapter 4: Creating and Managing Container Images

Chapter 5: Data Volumes and Configuration

Welcome to Packt Early Access

The Ultimate Docker Container Book, Fourth Edition: Build, ship, deploy, and scale containerized applications with Docker, Kubernetes, and the cloud

We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time.

You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book.

1

What Are Containers and Why Should I Use Them?

Join our book community on Discord:

https://packt.link/mqfS2

This first chapter will introduce you to the world of containers, showing how they streamline the modern software supply chain and address the security challenges that often arise with traditional deployment models. We'll assume no or minimal prior knowledge of containers, so our first steps focus on illustrating the *friction points* in legacy workflows and demonstrating how containers significantly reduce that friction. Building on this foundation, we'll explore both the classic ecosystem—where upstream OSS components (collectively known as **Moby**) serve as the building blocks behind familiar Docker products—and the latest containerization trends that have emerged or solidified since 2022, the time when the last edition of the book was written. You'll learn not only why containers were a revolutionary concept when they first appeared but also how features such as rootless operation modes, supply chain security enhancements, and new orchestration techniques are shaping today's container landscape. By the end of this chapter, you'll understand how containers are assembled and why they matter more than ever in delivering secure, portable applications.

The chapter covers the following topics:

- What are containers?
- Why are containers important?
- What's the benefit of using containers for me or my company?
- The Moby project
- Docker products
- Container architecture
- What's new in containerization

After completing this chapter, you will be able to do the following:

- Explain what containers are to an interested layperson, using everyday analogies such as physical cargo containers versus bulk shipping
- Justify why containers are so important by likening their approach to the difference between apartment homes and single-family homes, or similar simplified examples
- Name at least four upstream OSS components (united under Moby) that power Docker products such as Docker Desktop
- Draw a high-level sketch of the Docker container architecture to illustrate how layered images and namespaces fit together
- Identify the recent developments (post-2022) in containerization, including new security measures, rootless modes, and enhanced Kubernetes debugging, and explain how they continue to shape modern deployments

Let's get started!

What are containers?

A software container is a pretty abstract thing, so it might help if we start with an analogy that should be pretty familiar to most of you. The analogy is a shipping container in the transportation industry. Throughout history, people have been transporting goods from one location to another by various means. Before the invention of the wheel, goods would most probably have been transported in bags, baskets, or chests on the shoulders of humans themselves, or they might have used animals such as donkeys, camels, or elephants to transport them. With the invention of the wheel, transportation became a bit more efficient as humans built roads along which they could move their carts. Many more goods could be transported at a time. When the first steam-driven machines and, later, gasoline-driven engines were introduced, transportation became even more powerful. We now transport huge amounts of goods on planes, trains, ships, and trucks. At the same time, the types of goods became more and more diverse, and sometimes complex to handle. In all these thousands of years, one thing hasn't changed, and that is the necessity to unload goods at a target location and maybe load them onto another means of transportation. Take, for example, a farmer bringing a cart

full of apples to a central train station where the apples are then loaded onto a train, together with all the apples from many other farmers. Or think of a winemaker bringing their barrels of wine with a truck to the port, where they are unloaded and then transferred to a ship that will transport those barrels overseas.

This unloading from one means of transportation and loading onto another means of transportation was a really complex and tedious process. Every type of product was packaged in its own way and thus had to be handled in its own particular way. Also, loose goods faced the risk of being stolen by unethical workers or damaged in the process of being handled.

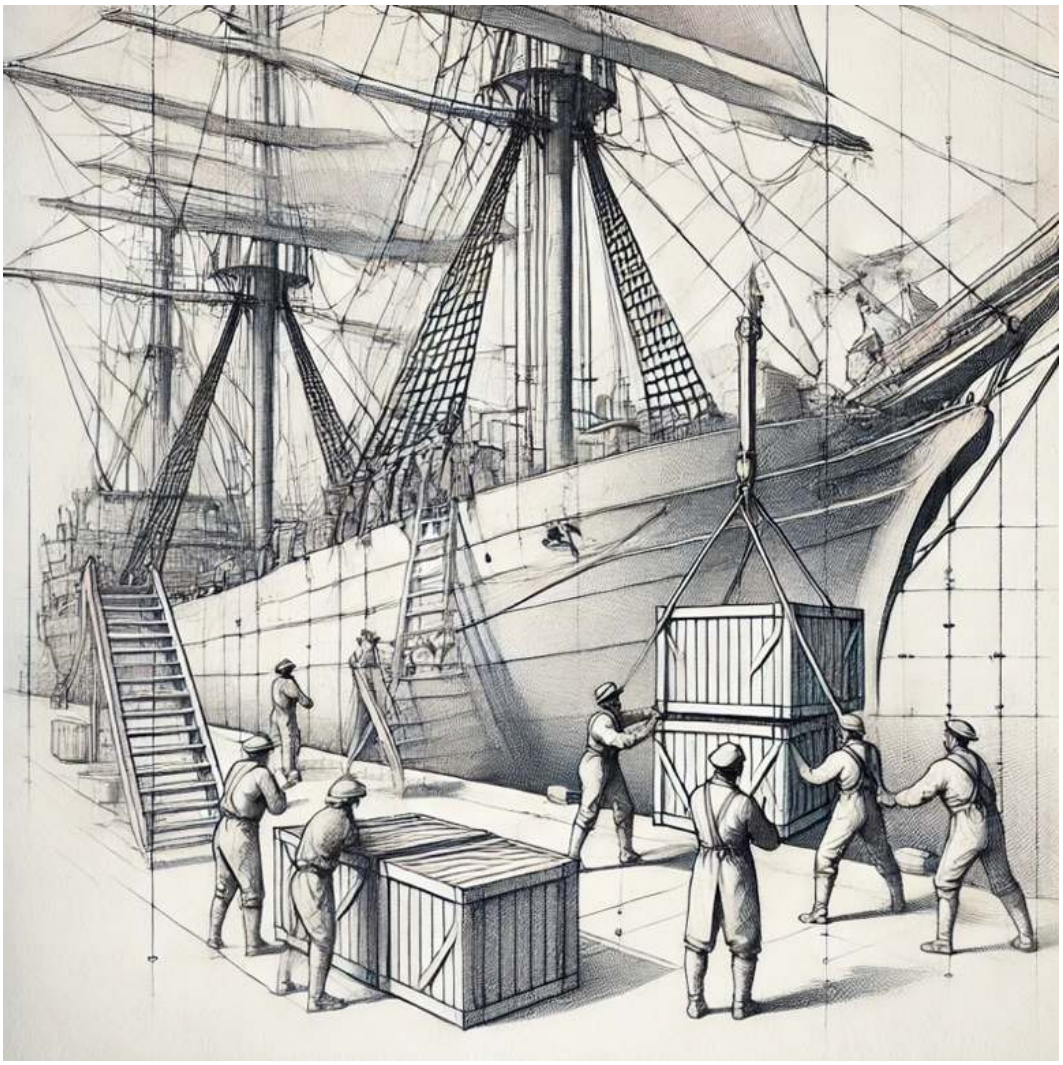


Figure 1.1 – Sailors unloading goods from a ship

Then, containers came along, and they totally revolutionized the transportation industry. A container is just a metallic box with standardized dimensions. The length, width, and height of each container are the same. This is a very important point. Without the world agreeing on a standard size, the whole container thing would not have been as successful as it is now. Now, with standardized containers, companies that want to have their goods transported from A to B package those goods into these containers. Then, they call a shipper, who comes with a standardized means of transportation. This can be a truck that can load a container, or a train whose wagons can each transport one or several containers. Finally, we have ships that are specialized in transporting huge numbers of containers. Shippers never need to unpack and repackage goods. For a shipper, a container is just a black box, and they are not interested in what is in it, nor should they care in most cases. It is just a big iron box with standard dimensions. Packaging goods into containers is now fully delegated to the parties who want to have their goods shipped, and they should know how to handle and package those goods. Since all containers have the same agreed-upon shape and dimensions, shippers can use standardized tools to handle containers—that is, cranes that unload containers, say from a train or a truck, and load them onto a ship, and vice versa. One type of crane is enough to handle all the containers that come along over time. Also, the means of transportation can be standardized, such as container ships, trucks, and trains. Because of all this standardization, all the processes in and around shipping goods could also be standardized and thus made much more efficient than they were before the introduction of containers.

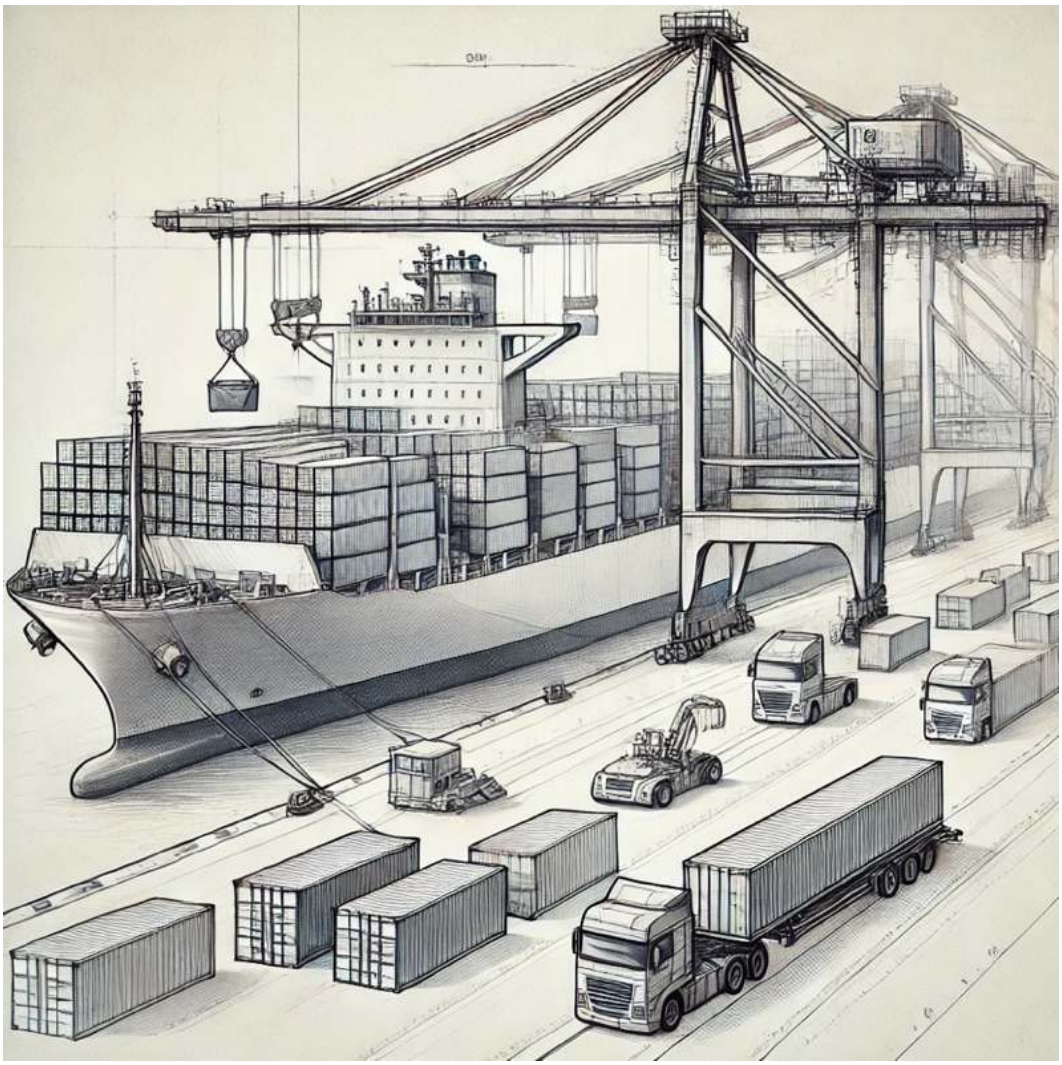


Figure 1.2 – Container ship being loaded in a port

Now, you should have a good understanding of why shipping containers are so important and why they revolutionized the whole transportation industry. I chose this analogy purposefully since the software containers that we are going to introduce here fulfill the exact same role in the so-called software supply chain as shipping containers do in the supply chain of physical goods.

Let's then have a look at what this whole thing means translated to the IT industry and software development, shall we? In the old days, developers would develop new applications. Once an application was completed in their eyes, they would hand that application over to the operations engineers, who were then supposed to install it on the production servers and get it running. If the operations engineers were lucky, they even got a somewhat accurate document with installation instructions from the developers. So far, so good,

and life was easy. But things got a bit out of hand when, in an enterprise, there were many teams of developers that created quite different types of applications, yet all of them needed to be installed on the same production servers and kept running there. Usually, each application has some external dependencies, such as the framework it was built on, the libraries it used, and so on. Sometimes, two applications use the same framework but in different versions that might or might not be compatible with each other. Our operations engineers' lives became much harder over time. They had to be really creative with how they loaded their ships, that is, their servers, with different applications without breaking something. Installing a new version of a certain application was now a complex project on its own, and often needed months of planning and testing beforehand. In other words, there was a lot of friction in the software supply chain.

But these days, companies rely more and more on software, and the release cycles need to become shorter and shorter. Companies cannot afford to just release application updates once or twice a year anymore. Applications need to be updated in a matter of weeks or days, or sometimes even multiple times per day. Companies that do not comply risk going out of business due to the lack of agility. So, what's the solution? One of the first approaches was to use **virtual machines** (**VMs**). Instead of running multiple applications all on the same server, companies would package and run a single application on each VM. With this, all the compatibility problems were gone, and life seemed to be good again. Unfortunately, that happiness didn't last long. VMs are pretty heavy beasts on their own since they all contain a full-blown operating system, such as Linux or Windows Server, and all that for just a single application. This is just as if you were in the transportation industry and were using a whole ship just to transport a single truckload of bananas. What a waste! That could never be profitable. The ultimate solution to this problem was to provide something that is much more lightweight than VMs but is also able to perfectly encapsulate the goods it needs to transport. Here, the goods are the actual application that has been written by our developers, plus—and this is important—all the external dependencies of the application, such as its framework, libraries, configurations, and more. This holy grail of a software packaging mechanism is the **Docker container**.

Developers use Docker containers to package their applications, frameworks, and libraries into them, and then they ship those containers to the testers or operations engineers. For testers and operations engineers, a container is just a black box. It is a standardized black box, though. All containers, no matter what application runs inside them, can be treated equally. The engineers know that if any container runs on their servers, then any other containers should run too. And this is actually true, apart from some edge cases, which always exist. Thus, Docker containers are a means to package applications and their dependencies in a standardized way. Docker then coined the phrase *Build, ship, and run anywhere*.

Why are containers important?

These days, the time between new releases of an application becomes shorter and shorter, yet the software itself does not become any simpler. On the contrary, software projects increase in complexity. Thus, we need a way to tame the beast and simplify the software supply chain. Also, every day, we hear that cyberattacks are on the rise. Many well-known companies are and have been affected by security breaches. Highly sensitive customer data gets stolen during such events, such as social security numbers, credit card information, health-related information, and more. Not only is customer data compromised, but sensitive company secrets are stolen, too. Containers can help in many ways. In a published report, Gartner found that applications running in a container are more secure than their counterparts that are not running in a container. Containers use Linux security primitives such as Linux kernel **namespaces** to sandbox different applications running on the same computer and **control groups** (**cgroups**) to avoid the noisy-neighbor problem, where one bad application is using all the available resources of a server and starving all other applications. Since container images are immutable, as we will learn later, it is easy to have them scanned for **common vulnerabilities and exposures** (**CVEs**), and in doing so, increase the overall security of our applications. Another way to make our software supply chain more secure is to have our containers use **content trust**. Content trust ensures that the author of a container image is who they say they are and that the consumer of the container image has a guarantee that the image has not been tampered with in transit. The latter is known as a **man-in-the-middle** (**MITM**) attack.

Everything I have just said is, of course, technically also possible without using containers, but since containers introduce a globally accepted standard, they make it so much easier to implement these best practices and enforce them. OK, but security is not the only reason containers are important. There are other reasons, too. One is the fact that containers make it easy to simulate a production-like environment, even on a developer's laptop. If we can containerize any application, then we can also containerize, say, a database such as Oracle, PostgreSQL, or MS SQL Server. Now, everyone who has ever had to install an Oracle database on a computer knows that this is not the easiest thing to do, and it takes up a lot of precious space on your computer. You would not want to do that to your development laptop just to test whether the application you developed really works end-to-end. With containers at hand, we can run a full-blown relational database in a container as easily as saying *one, two, three*. And when we are done with testing, we can just stop and delete the container, and the database will be gone, without leaving a single trace on our computer. Since containers are very lean compared to VMs, it is common to have many containers running at the same time on a developer's laptop without overwhelming the laptop. A third reason containers are important is that operators can finally concentrate on what they are good at—provisioning the infrastructure and running and monitoring applications in production. When the applications that must run on a production system are all containerized, then operators can start to standardize their infrastructure. Every server becomes just another **Docker host**. No special libraries or frameworks need to be installed on those servers—just an OS and a container runtime such as Docker. Furthermore, operators do not have to have intimate knowledge of the internals of applications anymore, since those applications run self-contained in containers that ought to look like black boxes to them, like how shipping containers look to personnel in the transportation industry.

What is the benefit of using containers for me or my company?

Somebody once said, "*...today every company of a certain size has to acknowledge that they need to be a software company...*"." In this sense, a modern bank is a software company that happens to specialize in the business of finance.

Software runs all businesses, period. As every company becomes a software company, there is a need to establish a software supply chain. For the company to remain competitive, its software supply chain must be secure and efficient. Efficiency can be achieved through thorough automation and standardization. But in all three areas—security, automation, and standardization—containers have been shown to shine. Large and well-known enterprises have reported that when containerizing existing legacy applications (many call them traditional applications) and establishing a fully automated software supply chain based on containers, they can reduce the cost for the maintenance of those mission-critical applications by a factor of 50% to 60%, and they can reduce the time between new releases of these traditional applications by up to 90%. That being said, the adoption of container technologies saves these companies a lot of money, and at the same time, it speeds up the development process and reduces the time to market.

The Moby project

Originally, when Docker (the company) introduced Docker containers, everything was open source. Docker did not have any commercial products then. Docker Engine, which the company developed, was a monolithic piece of software. It contained many logical parts, such as the container runtime, a network library, a RESTful (REST) API, a command-line interface, and much more. Other vendors or projects, such as Red Hat or Kubernetes, were using Docker Engine in their own products, but most of the time, they were only using part of its functionality. For example, Kubernetes did not use the Docker network library for Docker Engine but provided its own way of networking. Red Hat, in turn, did not update Docker Engine frequently and preferred to apply unofficial patches to older versions of Docker Engine, yet they still called it Docker Engine.

Out of all these reasons, and many more, the idea emerged that Docker had to do something to clearly separate Docker's open source part from Docker's commercial part. Furthermore, the company wanted to prevent competitors from using and abusing the name *Docker* for their own gains. This was the main reason the Moby project was born. It serves as an umbrella for most of the open source components Docker developed and continues to develop. These open source projects do not carry the name *Docker* anymore. The Moby

project provides components used for image management, secret management, configuration management, and networking and provisioning. Also, part of the Moby project is special Moby tools that are, for example, used to assemble components into runnable artifacts. Some components that technically belong to the Moby project have been donated by Docker to the **Cloud Native Computing Foundation** (**CNCF**) and thus do not appear in the list of components anymore. The most prominent ones are **notary**, **containerd**, and **runc**, where the first is used for content trust, and the latter two form the container runtime.

In the words of Docker, "*... Moby is an open framework created by Docker to assemble specialized container systems without reinventing the wheel. It provides a '"Lego set'" of dozens of standard components and a framework for assembling them into custom platforms....*"

Docker products

In the past, up until 2019, Docker separated its product lines into two segments. There was the **Community Edition** (**CE**), which was closed source yet completely free, and then there was the **Enterprise Edition** (**EE**), which was also closed source and needed to be licensed yearly. These enterprise products were backed by 24/7 support and were supported by bug fixes.

In 2019, Docker felt that what they had were two very distinct and different businesses. Consequently, they split away the EE and sold it to Mirantis. Docker itself wanted to refocus on developers and provide them with optimal tools and support to build containerized applications.

Docker Desktop

Part of the Docker offering includes products such as Docker Toolbox and Docker Desktop, with their editions for macOS, Windows, and Linux. All these products are mainly targeted at developers. Docker Desktop is an easy-to-install desktop application that can be used to build, debug, and test dockerized applications or services on a macOS, Windows, or Linux machine. Docker Desktop is a complete development environment that is deeply integrated with the hypervisor framework, network, and filesystem of the respective underlying operating system. These tools are the fastest and most reliable ways to run Docker on macOS, Windows, or Linux.

NOTE

Note

Docker Toolbox has been deprecated and is no longer in active development. Docker recommends using Docker Desktop instead.

Docker Hub

Docker Hub is the most popular service for finding and sharing container images. It is possible to create individual, user-specific accounts and organizational accounts under which Docker images can be uploaded and shared inside a team, an organization, or with the wider public. Public accounts are free while private accounts require one of several commercial licenses. Later in this book, we will use Docker Hub to download existing Docker images and upload and share our own custom Docker images.

Docker EE

Docker sold its Enterprise Edition (Docker EE) to Mirantis in November 2019 as part of a strategic realignment. Despite pioneering container technology, Docker, Inc. found itself under financial strain as the container ecosystem rapidly shifted toward Kubernetes. Docker EE—comprising **Universal Control Plane** (**UCP**), **Docker Trusted Registry** (**DTR**), and the enterprise-specific engine—no longer fitted Docker's evolving focus on developer workflows, Docker Desktop, and the Docker Hub ecosystem.

By transferring Docker EE to Mirantis, Docker obtained funding and freedom to concentrate on developer tooling and collaboration, while Mirantis acquired the enterprise business and its customer base, as well as engineers and IP. Mirantis continues to build upon Docker EE's core technologies, integrating them into its own Kubernetes-focused solutions. Meanwhile, Docker thrives as a primary driver of developer-centric container tooling, demonstrating how the container market has matured and specialized over time.

NOTE

Docker Swarm

Docker Swarm is Docker's native container orchestration feature that comes integrated with the Docker Engine—it is not a separate product. It provides a robust and flexible platform for deploying and

managing containerized applications at scale. With Swarm mode enabled, developers and operators can build, deploy, and operate distributed applications using the same familiar Docker CLI, benefitting from built-in features such as load balancing, service discovery, rolling updates, and secure multi-host networking.

Container architecture

Now, let us discuss how a system that can run Docker containers is designed at a high level. The following diagram illustrates what a computer on which Docker has been installed looks like. Note that a computer that has Docker installed on it is often called a Docker host because it can run or host Docker containers:

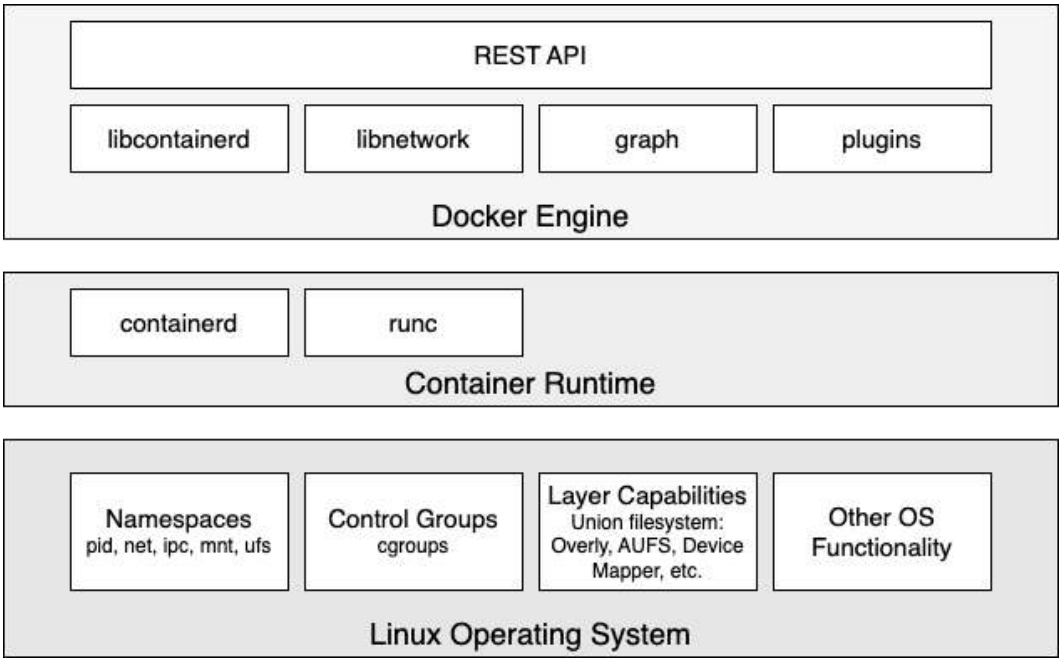


Figure 1.3 – High-level architecture diagram of Docker Engine

In the preceding diagram, we can see three essential parts:

- At the bottom, we have the **Linux operating system**
- In the middle, we have the **container runtime**
- At the top, we have **Docker Engine**

Containers are only possible because the Linux OS supplies some primitives, such as namespaces, control groups, layer capabilities, and more, all of which are used in a specific way by the container runtime and Docker Engine. Linux

kernel namespaces, such as process ID (`pid`) namespaces, or network (`net`) namespaces, allow Docker to encapsulate or sandbox processes that run inside the container. Control groups make sure that containers do not suffer from noisy-neighbor syndrome, where a single application running in a container can consume most or all the available resources of the whole Docker host. Control groups allow Docker to limit the resources, such as CPU time or the amount of RAM, that each container is allocated. The container runtime on a Docker host consists of `containerd` and `runc`. The latter is the low-level functionality of the container runtime, such as container creation or management, while `containerd`, which is based on `runc`, provides higher-level functionality, such as image management, networking capabilities, or extensibility via plugins. Both are open source and have been donated by Docker to the CNCF. The container runtime is responsible for the whole life cycle of a container. It pulls a container image (which is the template for a container) from a registry, if necessary, creates a container from that image, initializes and runs the container, and eventually stops and removes the container from the system when asked. Docker Engine provides additional functionality on top of the container runtime, such as network libraries or support for plugins. It also provides a REST interface over which all container operations can be automated. The Docker **command-line interface** (**CLI**) that we will use often in this book is one of the consumers of this REST interface.

What's new in containerization

Although containers have been around for nearly a decade, the ecosystem has not been standing still. Over the past few years, you've likely seen an explosion in complementary tools, runtimes, and security features. Not all of these are strictly "new"—some debuted earlier in beta form—but 2022 onward has been a tipping point, pushing once-experimental ideas firmly into mainstream adoption. Here's a summary of the standout developments that have truly gained ground in that timeframe.

Enhanced supply chain security

One of the biggest stories in containerization since 2022 has been the shift to deeper security. Gone are the days when we simply scanned images after

shipping them. Now, organizations demand full transparency and traceability from the earliest point in the supply chain. A few highlights are as follows:

- **Image signing and verification**: Tools such as Notary v2 and Cosign are moving beyond prototypes and finding real usage in production pipelines. They let you sign your images cryptographically so that any downstream user (whether developer, QA engineer, or operator) can be certain the image hasn't been tampered with along the way. Since 2022, these signing workflows have become far more common, fueled by high-profile supply chain attacks that exposed just how vulnerable unverified images can be.
- **SBOM generation**: While **software bills of materials** (**SBOMs**) were a talking point back in 2021, they truly landed on the mainstream radar by mid-2022, especially with developer-friendly tools such as Syft, Anchore, and a variety of plugins for existing CI/CD solutions. The typical approach is to generate an SBOM at build time, capturing exactly which versions of libraries and frameworks went into your container. This "ingredient list" makes it much easier to react to newly discovered vulnerabilities—or track down dangerous dependencies such as log4j.

Debugging and operations in Kubernetes

As more enterprises transitioned to Kubernetes at scale, operational workflows matured. By 2022, one feature in particular (namely, ephemeral containers) began appearing in everyday cluster operations.

Originally introduced before 2022, **ephemeral containers** gained real traction once folks realized how straightforward it is to attach a debugging container to a Pod already running in production. You can spin up a short-lived container image with the needed diagnostic tools (think: cURL, netcat, specialized log scrapers) and run them right alongside your main application process. By 2022 and into 2023, ephemeral containers cemented their status as a go-to mechanism for live troubleshooting without stopping or rebuilding your entire Pod.

Docker Desktop extensions

Since mid-2022, Docker Desktop gained an Extensions Marketplace, allowing users to integrate third-party tools directly into the Docker Desktop UI. While

Docker Desktop has long provided a seamless way to build and run containers locally, these new extensions push it further:

- **Security scanning extensions**: Many teams now adopt Docker Desktop extensions for image scanning (for instance, the Snyk or Trivy extensions) right as they're building locally. This shortens feedback loops, catching vulnerabilities before code ever makes it to a shared repository.
- **Multi-service management**: Some extensions help you orchestrate and monitor multiple services, letting you visualize containers or tweak volumes and networks from a single interface. Because these extensions are curated on Docker's marketplace, developers can install them with one click, making for a frictionless setup that even new team members can handle.

Evolving resource management

Lastly, container engines and orchestrators have kept refining how they handle resource isolation:

- **cgroups v2 adoption**: Although cgroups v2 was initially introduced earlier, full Docker support and stable usage across major Linux distros were locked in during 2022. Operators now benefit from more precise accounting of CPU, memory, and I/O usage at scale, which is crucial for multi-tenant environments. Docker's improved stability with cgroups v2 means that if you're running the latest Linux kernels, you can rely on better insight and control over container performance.
- **Rootless modes (more mature)**: Running containers as rootless—thus mitigating some of the biggest security concerns—saw broader real-world deployments last year. Formerly considered "experimental," rootless Docker modes are now stable enough that companies with strict security requirements are confidently rolling them out in production. While some features are still limited compared to traditional Docker, the overall experience for rootless has become smoother and far better documented.

Where do we go from here?

Altogether, these developments show that the container world no longer revolves solely around a single Docker Engine or a single orchestrator. Instead, we have a rapidly evolving toolkit that covers everything from building more secure images (complete with SBOMs, signed content, and rootless isolation) to debugging distributed applications in real time (Kubernetes ephemeral containers). If you're coming from older container setups, you'll notice a dramatic uptick in built-in security checks, official disclaimers about package versions, and integrated services that keep watch on every step of your build-and-run pipeline.

More than ever, containers aren't just a developer convenience. They're cornerstones of resilient, auditable, and secure systems. As you read on in this book, you'll see these newer features intersect with our core principles of Dockerized workflows—speed, consistency, and the power to easily scale up or shift environments without the legacy overhead. Keep an eye on these tools and trends, because we're likely to see even tighter integrations and more advanced capabilities in the very near future.

Summary

In this chapter, we saw how containers dramatically reduce software supply chain friction while reinforcing overall security—a benefit rooted in the open source Moby components at Docker's core. We also introduced emerging trends from 2022 onward, such as enhanced image signing and rootless operation, to show why containers remain a central force in modern deployments. In the next chapter, we'll go hands-on with Docker commands, learning how to run, stop, and inspect containers while exploring their basic anatomy. This is where you'll begin to see these theoretical concepts take shape in practical, everyday scenarios. Stay tuned!

Further reading

The following is a list of links that lead to more detailed information regarding the topics we discussed in this chapter:

- *Docker overview*: `https://docs.docker.com/engine/docker-overview/`
- *The Moby project*: `https://mobyproject.org/`
- *Docker products*: `https://www.docker.com/get-started`

- *Docker Desktop*: https://www.docker.com/products/docker-desktop/
- *Cloud-Native Computing Foundation*: https://www.cncf.io/
- *containerd – an industry-standard container runtime*: https://containerd.io/
- *Mirantis Kubernetes Engine 4*: https://www.mirantis.com/software/mirantis-kubernetes-engine/
- *Rootless Docker Documentation*: https://docs.docker.com/engine/security/rootless/
- *Kubernetes Ephemeral Containers*: https://kubernetes.io/docs/concepts/workloads/pods/ephemeral-containers/
- *Image Signing and Supply Chain Security*:
- *Notary v2*: https://github.com/notaryproject/notaryproject
- *Cosign / Sigstore*: https://docs.sigstore.dev
- *cgroups v2 in Practice*: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html
- *SBOM Generation Tools*:
- *Syft*: https://github.com/anchore/syft
- *CycloneDX*: https://cyclonedx.org/

Questions

Please answer the following questions to assess your learning progress:

1. Which statements are correct regarding containers? (Multiple answers may apply.)
    a. A container is essentially the same as a lightweight VM.
    b. A container only runs on a Linux host.
    c. A container can run exactly one process and no more.
    d. The main process in a container always has PID 1 inside that container's namespace.
    e. A container is one or more processes encapsulated by Linux namespaces and restricted by cgroups.

2. In your own words, by using analogies, explain what a container is. (Hint: Compare it to a physical shipping container or a standardized way of packaging.)
3. Why are containers considered game-changers in IT? Name three or four key reasons. (Think about portability, reduced friction, cloud integration, immutability, and security.)
4. What does it mean when we claim, "If a container runs on a given platform, then it runs anywhere?" Give two or three reasons why this is true.
5. True or false? "Docker containers are only useful for modern greenfield applications based on microservices." Provide a justification for your answer.
6. How much do enterprises typically save on maintenance when containerizing their legacy applications?
    a. 20%
    b. 33%
    c. 50%
    d. 75%
7. On which two core Linux concepts are containers based? (Hint: This includes a method to isolate processes and another to control resource usage.)
8. Which operating systems currently support Docker Desktop? (Note: Keep in mind recent developments regarding Docker Desktop for Linux.)
9. Name at least two new containerization features or practices that gained traction from 2022 onward, and briefly explain why they are important. (Hint: Consider ephemeral containers for Kubernetes debugging, rootless Docker, cgroups v2, supply chain security enhancements, and/or image signing.)

Answers

1. The correct answers are D and E:
    - d. Within a container's own namespace, the main process has PID 1.

    - e. A container is one or more processes encapsulated by Linux namespaces and restricted by cgroups.
2. A helpful analogy compares software containers to the standardized shipping containers used in global trade. Much like physical containers, software containers provide a uniform packaging mechanism. Once developers place an application and its dependencies inside the container, it can be shipped and run anywhere that supports containers, simplifying logistics and boosting consistency across environments.
3. The following are the reasons why containers are considered gamechangers in IT:
    - They **standardize and isolate** applications and dependencies, reducing environment conflicts
    - They're **portable**, enabling the same container to run on-premises, in the cloud, or in hybrid scenarios
    - They encourage **rapid, consistent releases** because images are immutable, and builds are developer-driven
    - They **strengthen security** through namespaces, cgroups, and container-scanning tools
4. A few reasons why the "If a container runs on a given platform, then it runs anywhere" statement is true are as follows:
    - A container **bundles all dependencies** inside its image, making it self-contained
    - Containers adhere to **widely accepted open standards** (OCI), meaning any conforming engine can run them
    - They **abstract away OS-level quirks**, so compatibility issues are minimal across different hosts or cloud providers
5. This statement is **false**. Containers are equally beneficial for existing monolithic or legacy applications. Enterprises have reported over 50% cost savings and significantly faster release cycles when containerizing older systems ("lift and shift")—all without rewriting their application logic.
6. The correct answer is **C. 50% or more**. In many published success stories, organizations have seen at least a 50% reduction in maintenance overhead, along with faster deployment timelines.

7. Containers rely on **Linux namespaces** (to isolate processes, network, users, etc.) and **cgroups** (to control and limit resource usage).
8. Docker Desktop is supported on **macOS, Windows, and Linux** (with official Linux support becoming broadly available more recently).
9. Here are a few new containerization features or practices that gained traction from 2022 onward, and why they matter:
    - **Kubernetes ephemeral containers**: Allow real-time debugging by attaching short-lived containers to a running Pod, simplifying on-the-fly troubleshooting
    - **Rootless Docker modes**: Let you run Docker with non-root privileges, reducing security risks and broadening adoption for compliance-heavy environments
    - **cgroups v2 adoption**: Provides finer-grained resource isolation and reporting, making multi-tenant workloads more efficient and stable
    - **Image signing with Notary v2 or Cosign**: Adds cryptographic guarantees and traceability to containers, a major step toward mitigating software supply chain attacks

2

Setting Up a Working Environment

Join our book community on Discord:

https://packt.link/mqfS2

In the previous chapter, we learned what Docker containers are and why they're important. We learned what kinds of problems containers solve in a modern software supply chain. In this chapter, we are going to prepare our personal or working environment to work efficiently and effectively with Docker. We will discuss in detail how to set up an ideal environment for developers, DevOps, and operators that can be used when working with Docker containers.

This chapter covers the following topics:

- Distinguishing the major operating systems
- The Linux command shell
- PowerShell for Windows
- Installing and using a package manager
- Installing Git and cloning the code repository
- Choosing and installing a code editor
- Installing Docker Desktop on macOS or Windows
- Using Docker with WSL 2 on Windows
- Installing Docker Toolbox
- Enabling Kubernetes on Docker Desktop
- Installing Podman
- Installing minikube
- Installing kind

After completing this chapter, you will be able to do the following:

- Set up a professional-grade development environment for containerized software development on macOS, Windows, or Linux
- Use a package manager, shell, and code editor to configure your local system for working with containers
- Install and verify Docker, Podman, and Kubernetes tooling such as minikube and Kind across all supported platforms
- Test your setup end-to-end to ensure containers and Kubernetes workloads can be built, run, and orchestrated locally without issues

Technical requirements

For this chapter, you will need a laptop or a workstation with either macOS or Windows, preferably Windows 11 Professional, installed. You should also have free internet access to download applications and permission to install those applications on your laptop. It is also possible to follow along with this book if you have a Linux distribution as your operating system, such as Ubuntu 24.10 or newer. I will try my best to indicate where commands and samples differ significantly from the ones on macOS, which will be my primary platform throughout this book.

Distinguishing the major operating systems

While Docker is available for all three major platforms—macOS, Windows, and Linux—each environment has its nuances. Before we dive deeper into the details of the chapter, let's give a brief summary of all three operating systems:

macOS

- **System requirements**: Intel-based Macs require macOS 10.14 or above, while Apple Silicon (M1/M2) chips need macOS 11 or later. Also note that older versions may need **Docker Toolbox**.
- **Preferred installation**: Use the dedicated Docker Desktop for Mac (`https://www.docker.com/products/docker-desktop`). It seamlessly integrates with the macOS hypervisor (HyperKit on Intel, Apple's own hypervisor framework on Apple Silicon).
- **Package manager**: Installing additional tools (such as `git` or `jq`) is typically easiest via Homebrew.

Windows

- **System requirements**: Windows 10 or 11 Professional or Enterprise editions support Docker Desktop with WSL2 or Hyper-V. Home editions can often use WSL2 but may require extra configuration.
- **Preferred installation**: Docker Desktop for Windows uses Hyper-V or WSL2 as its underlying virtualization. If you're on Windows Home, you can still install WSL2 and run Docker Desktop with it.
- **Package manager**: Chocolatey (or the newer Windows Package Manager, `winget`) simplifies installing developer tools.

Linux

- **System requirements**: A modern Linux distribution (Ubuntu, Debian, Fedora, CentOS, etc.). Kernel must support cgroups and namespaces. For older distros, check Docker's official documentation.
- **Preferred installation**: Install Docker Engine directly from your distribution's package repositories or use Docker's official repository. Tools such as **minikube** may require a specific hypervisor (KVM, VirtualBox).
- **Package manager**: Varies by distribution (`apt` for Debian/Ubuntu, `dnf` or `yum` for Fedora/CentOS, etc.).

The Linux command shell

Docker containers were first developed on Linux for Linux. Hence, it is natural that the primary command-line tool used to work with Docker, also called a shell, is a Unix shell; remember, Linux derives from Unix. Most developers use the Bash shell. On some lightweight Linux distributions, such as Alpine, Bash is not installed, and consequently, you must use the simpler Bourne shell, just called `sh`. Whenever we are working in a Linux environment, such as inside a container or on a Linux VM, we will use either `/bin/bash` or `/bin/sh`, depending on their availability.

Although Apple's macOS is not a Linux OS, Linux and macOS are both flavors of Unix and hence support the same set of tools. Among those tools are the shells. So, when working on macOS, you will probably be using the Bash or Zsh shell.

In this book, we expect you to be familiar with the most basic scripting commands in Bash and PowerShell, if you are working on Windows. If you are an absolute beginner, then we strongly recommend that you familiarize yourself with the following cheat sheets:

- *Linux Command Line Cheat Sheet* by Dave Child at `http://bit.ly/2mTQr8l`
- *PowerShell Basic Cheat Sheet* at `http://bit.ly/2EPHxze`

PowerShell for Windows

On a Windows computer, laptop, or server, we have multiple command-line tools available. The most familiar is the command shell. It has been available on any Windows computer for decades. It is a very simple shell. For more advanced scripting, Microsoft has developed PowerShell. PowerShell is very powerful and very popular among engineers working on Windows. Finally, on Windows 10 or later, we have the so-called Windows Subsystem for Linux, which allows us to use any Linux tool, such as the Bash or Bourne shells. Apart from this, other tools also install a Bash shell on Windows, such as the Git Bash shell. In this book, all commands will use Bash syntax. Most of the commands also run in PowerShell.

Therefore, we recommend that you either use PowerShell or any other Bash tool to work with Docker on Windows.

Installing and using a package manager

The easiest way to install software on a Linux, macOS, or Windows laptop is to use a good package manager. On macOS, most people use Homebrew, while on Windows, the Windows package manager (`winget`) or Chocolatey are good choices. If you're using a Debian-based Linux distribution such as Ubuntu, then the package manager of choice for most is `apt`, which is installed by default.

Installing Homebrew on macOS

Homebrew is the most popular package manager on macOS, and it is easy to use and very versatile. Installing Homebrew on macOS is simple; just follow the instructions at `https://brew.sh/`:

1. In a nutshell, open a new Terminal window and execute the following command to install Homebrew:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

2. Once the installation has finished, test whether Homebrew is working by entering `brew --version` in the Terminal. You should see something like this:

```
$ brew --version
Homebrew 4.4.23
```

3. Now, we are ready to use Homebrew to install tools and utilities. If we, for example, want to install the iconic Vi text editor (note that this is not a tool we will use in this book; it serves just as an example), we can do so like this:

```
$ brew install vim
```

This will download and install the editor for you.

Installing Chocolatey on Windows

Chocolatey is a popular package manager for Windows, built on PowerShell. To install the Chocolatey package manager, please follow these instructions:

1. Open PowerShell as administrator: Press *Win* + *S*, type `"PowerShell"`, and select **"Run as administrator"**.
2. Set the execution policy:
    a. In the PowerShell window, check the current execution policy by typing the following:

    ```
    Get-ExecutionPolicy
    ```

    b. If it returns `Restricted`, change it to `AllSigned` or `Bypass` to allow the installation script to run:

    ```
    Set-ExecutionPolicy Bypass -Scope Process -Force
    ```

3. To install Chocolatey, run the following command in the PowerShell window:

```
[System.Net.ServicePointManager]::SecurityProtocol =
[System.Net.ServicePointManager]::SecurityProtocol -bor 3072;
iex ((New-Object
System.Net.WebClient).DownloadString('https://community.chocolat
ey.org/install.ps1'))
```

4. After the installation completes, verify that Chocolatey is installed by typing the following:

```
choco
```

5. Check the installed version of Chocolatey by typing the following command and pressing *Enter*:

```
choco --version
```

   You should see an output like this:

```
2.4.3
```

   This indicates that, at the time of writing, Chocolatey was at version `2.4.3`.

6. Try to install an application with Chocolatey, such as Vim:

```
choco install -y vim
```

   The `-y` parameter makes sure that the installation happens without Chocolatey asking for a reconfirmation. As mentioned previously, we will not use Vim in our exercises; it has only been used as an example.

NOTE

Note

Once Chocolatey has installed an application, you may need to open a new PowerShell window to use that application.

Installing Git and cloning the code repository

We will be using Git to clone the sample code accompanying this book from its GitHub repository. If you already have Git installed on your computer, you can skip this section:

1. To install Git on macOS, use the following command in a Terminal window:

```
$ brew install git
```

2. To install Git on Windows, open a PowerShell window and use Chocolatey to install it:

```
PS> choco install git -y
```

3. Finally, on your Debian or Ubuntu machine, open a Bash console and execute the following command:

```
$ sudo apt update && sudo apt install -y git
```

4. Once Git has been installed, verify that it is working. On all platforms, use the following command:

```
$ git --version
```

   This should output the version of Git that's been installed. On the author's MacBook Pro M2, the output is as follows:

```
git version 2.49.0
```

NOTE

Note

If you see an older version, then you are probably using the version that came installed with macOS by default. Use Homebrew to install the latest version by running `$ brew install git`.

1. Now that Git is working, we can clone the source code accompanying this book from GitHub. Execute the following command:

```
$ cd ~
$ git clone  https://github.com/PacktPublishing/The-Ultimate-Docker-Container-Book-Fourth-
Edition.git
```

   This will clone the content of the main branch into your local folder, `~/The-Ultimate-Docker-Container-Book-v4`. This folder will now contain

all of the sample solutions for the labs we are going to do together in this book. Refer to these sample solutions if you get stuck.

Now that we have installed the basics, let's continue with the code editor.

Choosing and installing a code editor

Using a good code editor is essential to working productively with Docker. Of course, which editor is the best is highly controversial and depends on your personal preference. A lot of people use Vim, or others such as Emacs, Atom, Sublime, or **Visual Studio Code** (**VS Code**), to name just a few. VS Code is a completely free and lightweight editor, yet it is very powerful and is available for macOS, Windows, and Linux. According to Stack Overflow, it is currently by far the most popular code editor. If you are not yet sold on another editor, I highly recommend that you give VS Code a try.

But if you already have a favorite code editor, then please continue using it. So long as you can edit text files, you're good to go. If your editor supports syntax highlighting for Dockerfiles and JSON and YAML files, then even better. The only exception will be *Chapter 6*, *Debugging Code Running in a Container*. The examples presented in that chapter will be heavily tailored toward VS Code.

Installing VS Code on macOS

Follow these steps for installation:

1. Open a new Terminal window and execute the following command:

```
$ brew install -cask visual-studio-code
```

2. Once VS Code has been installed successfully, navigate to your home directory:

```
cd ~
```

3. Now, open VS Code from within this folder:

```
$ code The-Ultimate-Docker-Container-Book-v4
```

   VS will start and open the `The-Ultimate-Docker-Container-Book-v4` folder, where you just downloaded the repository that contains the source code for this book, as the working folder.

NOTE

Note

If you already have VS Code installed without using brew, then the guide at `https://code.visualstudio.com/docs/setup/mac#_launching-from-the-command-line` will add code to your PATH.

Use VS Code to explore the code that you can see in the folder you just opened.

Installing VS Code on Windows

Follow these steps for installation:

1. Open a new PowerShell window in *admin mode* and execute the following command:

```
PS> choco install vscode -y
```

2. Close your PowerShell window and open a new one to make sure VS Code is in your path.
3. Now, navigate to your home directory:

```
PS> cd ~
```

4. Now, open VS Code from within this folder:

```
PS> code The-Ultimate-Docker-Container-Book-v4
```

   VS will start and open the `The-Ultimate-Docker-Container-Book-v4` folder, where you just downloaded the repository that contains the source code for this book, as the working folder.

Use VS Code to explore the code that you can see in the folder you just opened.

Installing VS Code on Linux

Follow these steps for installation. We will use snap for this:

1. First, we need to ensure that Snap is installed. Most Debian and Ubuntu systems come with Snap pre-installed. To verify, open a terminal and run the following:

```
$ snap --version
```

2. If Snap isn't installed, you'll need to install it. For Debian-based systems, use the following:

```
$ sudo apt update
$ sudo apt install snapd
```

3. Now, to install VS Code via Snap, in your Bash Terminal, execute the following statement:

```
$ sudo snap install --classic code
```

4. The `--classic` flag ensures VS Code has the necessary permissions to function correctly.
5. If you're using a Linux distribution that's not based on Debian or Ubuntu, then please follow the following link for more details: https://code.visualstudio.com/docs/setup/linux.
6. Once VS Code has been installed successfully, navigate to your home directory:

```
$ cd ~
```

7. Now, open Visual Studio Code from within this folder:

```
$ code The-Ultimate-Docker-Container-Book-v4
```

   VS will start and open the `The-Ultimate-Docker-Container-Book-v4` folder, where you just downloaded the repository that contains the source code for this book, as the working folder.

Use VS Code to explore the code that you can see in the folder you just opened.

Installing VS Code extensions

Extensions are what make VS Code such a versatile editor. On all three platforms (macOS, Windows, and Linux), you can install VS Code extensions the same way:

1. Open a Bash console (or PowerShell in Windows) and execute the following group of commands to install the most essential extensions we are going to use in the upcoming examples in this book:

```
code --install-extension vscjava.vscode-java-pack
code --install-extension ms-dotnettools.csharp
```

```
code --install-extension ms-python.python
code --install-extension ms-azuretools.vscode-docker
code --install-extension eamodio.gitlens
```

We are installing extensions that enable us to work with Java, C#, .NET, and Python much more productively. We're also installing an extension built to enhance our experience with Docker.

2. After the preceding extensions have been installed successfully, restart VS Code to activate the extensions. You can now click the extensions icon in the activity pane on the left-hand side of VS Code to see all the installed extensions.
3. To get a list of all installed extensions in your VS Code, use this command:

```
$ code --list-extensions
```

Currently, AI is eating the world. This is specifically true in software development and associated fields. In this regard, we cannot miss discussing the installation of at least one popular AI-powered development environment.

Installing cursor.ai

As artificial intelligence continues to revolutionize software development, tools such as **cursor.ai** have emerged to help streamline your coding experience. cursor.ai is an intelligent assistant integrated directly into Visual Studio Code that provides real-time code suggestions, context-aware completions, and insightful recommendations—all designed to boost your productivity.

Follow these steps to install cursor.ai:

1. **Download the installer**: Visit `https://www.cursor.com` and click **Download**. The website detects your operating system and provides the appropriate installer.
2. **Run the installer**: Execute the downloaded file and follow the installation prompts (this process is similar to installing any standard application on Windows or macOS).

3. **Launch and configure**: Once installed, launch Cursor from your **Start** menu (Windows) or `Applications` folder (macOS). On first launch, you'll be prompted to configure settings (such as keyboard shortcuts, language, and code base indexing) and sign in with your account.

cursor.ai transforms your coding workflow by integrating advanced AI directly into a familiar, VS Code-based editor. It offers intelligent code completions, natural language editing, real-time debugging assistance, and context-aware code base insights to help you write, refactor, and troubleshoot code more efficiently. For more details, please visit `https://www.cursor.com/`.

Now that we have installed a proper code editor, let's focus on Docker and install Docker Desktop.

Installing Docker Desktop on macOS, Windows, or Linux

If you are using macOS or have Windows 10 or later installed on your laptop, then we strongly recommend that you install Docker Desktop. Since early 2022, Docker has also released a version of Docker Desktop for Linux. Docker Desktop gives you the best experience when working with containers. Follow these steps to install Docker Desktop for your system:

1. If you're working on Linux, please navigate to `https://docs.docker.com/desktop/install/linux/` and follow the instructions to install Docker Desktop. When done, skip to the "*Testing Docker Engine*" section.
2. No matter whether you're using Windows or macOS, navigate to the Docker start page at `https://www.docker.com/get-started`:

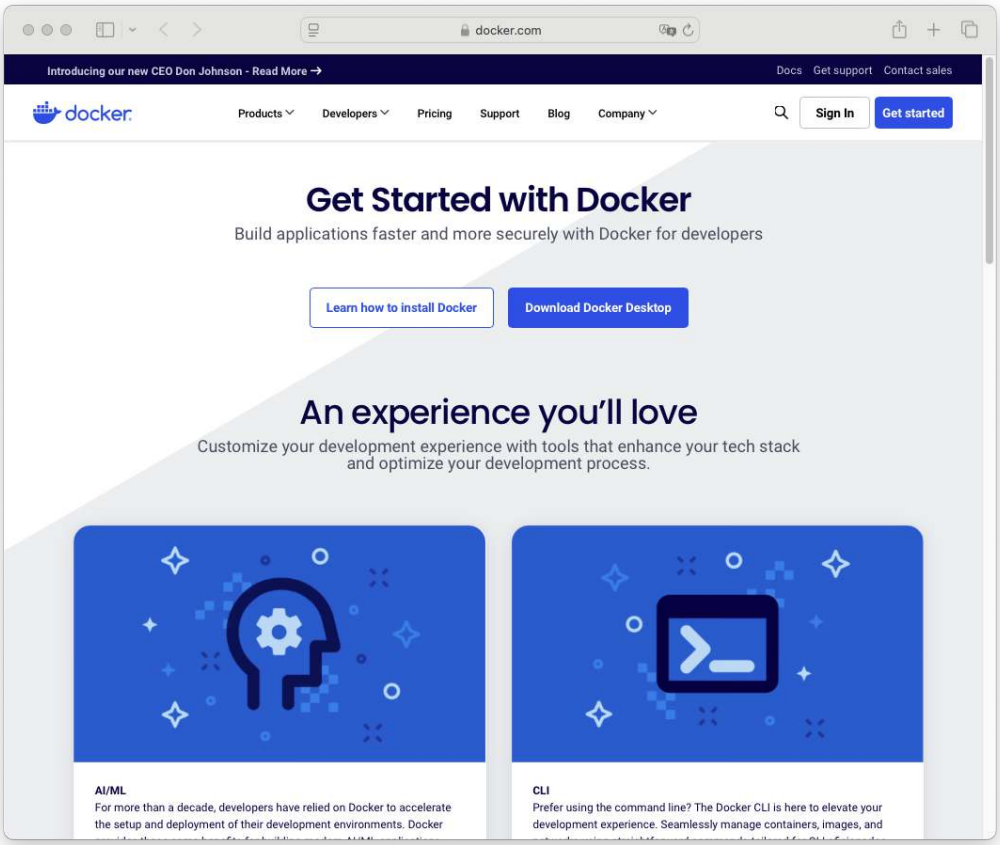


Figure 2.1: Get Started with Docker

3. In the upper-right corner of the view, you will find a **Sign In** button for Docker Hub. Click this button even if you don't yet have an account on Docker Hub, then follow the instructions to either log in or create an account. It is free, but you need an account to download the software.
4. In the previous screenshot, *Figure 2.1*, you will find a blue button called **Download Docker Desktop**. When you click it, a popup will appear, as shown in the following screenshot, containing the list of available downloads:

Figure 2.2: List of Docker Desktop targets

Select the one that is appropriate for you and observe the installation package being downloaded.

5. Once the package has been completely downloaded, proceed with the installation, usually by double-clicking on the download package.

Testing Docker Engine

Now that you have successfully installed Docker Desktop, let's test it. We will start by running a simple Docker container directly from the command line:

1. Open a Terminal window and execute the following command:

```
$ docker version
```

You should see something like this:

```
❯ docker version
Client:
 Version:           27.4.0
 API version:       1.47
 Go version:        go1.22.10
 Git commit:        bde2b89
 Built:             Sat Dec  7 10:35:43 2024
 OS/Arch:           darwin/arm64
 Context:           desktop-linux

Server: Docker Desktop 4.37.2 (179585)
 Engine:
  Version:          27.4.0
  API version:      1.47 (minimum version 1.24)
  Go version:       go1.22.10
  Git commit:       92a8393
  Built:            Sat Dec  7 10:38:33 2024
  OS/Arch:          linux/arm64
  Experimental:     false
 containerd:
  Version:          1.7.21
  GitCommit:        472731909fa34bd7bc9c087e4c27943f9835f111
 runc:
  Version:          1.1.13
  GitCommit:        v1.1.13-0-g58aa920
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0
```

Figure 2.3: Docker version of Docker Desktop

In the preceding output, we can see that it consists of two parts – a client and a server. Here, the server corresponds to Docker Engine, which is responsible for hosting and running containers. At the time of writing, the version of Docker Engine is `27.4.0`.

2. To see whether you can run containers, enter the following command into the Terminal window and hit *Enter*:

```
$ docker container run hello-world
```

If all goes well, your output should look something like the following:

```
❯ docker container run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c9c5fd25a1bd: Pull complete
Digest: sha256:7e1a4e2d11e2ac7a8c3f768d4166c2defeb09d2a750b010412b6ea13de1efb19
Status: Downloaded newer image for hello-world:latest

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

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

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

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

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

Figure 2.4: Running hello-world on Docker Desktop for macOS

If you read the preceding output carefully, you will have noticed that Docker didn't find an image called `hello-world:latest`, and thus decided to download it from a Docker image registry. Once downloaded, Docker Engine created a container from the image and ran it. The application runs inside the container and then outputs all the text, starting with **Hello from Docker!**

This is proof that Docker is installed and working correctly on your machine.

3. Let's try another funny test image that's usually used to check the Docker installation. Run the following command:

```
$ docker container run rancher/cowsay hello
```

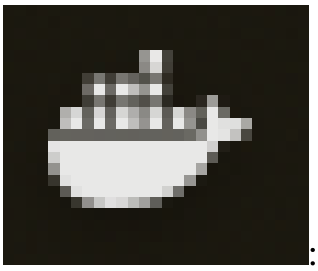
You should see this or a similar output:

```
● ➜  ~ docker container run rancher/cowsay Hello
Unable to find image 'rancher/cowsay:latest' locally
latest: Pulling from rancher/cowsay
cbdbe7a5bc2a: Pull complete
dd05e66d8cea: Pull complete
34d5e986f175: Pull complete
13eefd6dff68: Pull complete
Digest: sha256:5dab61268bc18daf56febb5a856b618961cd806dbc49a22a636128ca26f0bd94
Status: Downloaded newer image for rancher/cowsay:latest
WARNING: The requested image's platform (linux/amd64) does not match the detecte
d
 _______
< Hello >
 _______
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Figure 2.5: Running the cowsay image from Rancher

Great – we have confirmed that Docker Engine works on our local computer. Now, let's make sure the same is true for Docker Desktop.

Testing Docker Desktop

Depending on the operating system you are working with, be it Linux, Mac, or Windows, you can access the context menu for Docker Desktop in different areas. In any case, the symbol you are looking for is the little whale carrying containers. Here is the symbol as found on a Mac:  :

- **Mac**: 'You'll find the icon on the right-hand side of your menu bar at the top of the screen
- **Windows**: You'll find the icon in the Windows system tray
- **Linux**: *Here are the instructions for Ubuntu. On your distro, it may be different.* To start Docker Desktop for Linux, search for **Docker Desktop** via the **Applications** menu and open it. This will launch the Docker menu icon and open the Docker dashboard, reporting the status of Docker Desktop.

Once you have located the context menu for Docker Desktop on your computer, proceed with the following steps:

1. Click the *whale* icon to display the context menu of Docker Desktop. On the author's' Mac, it looks like this:

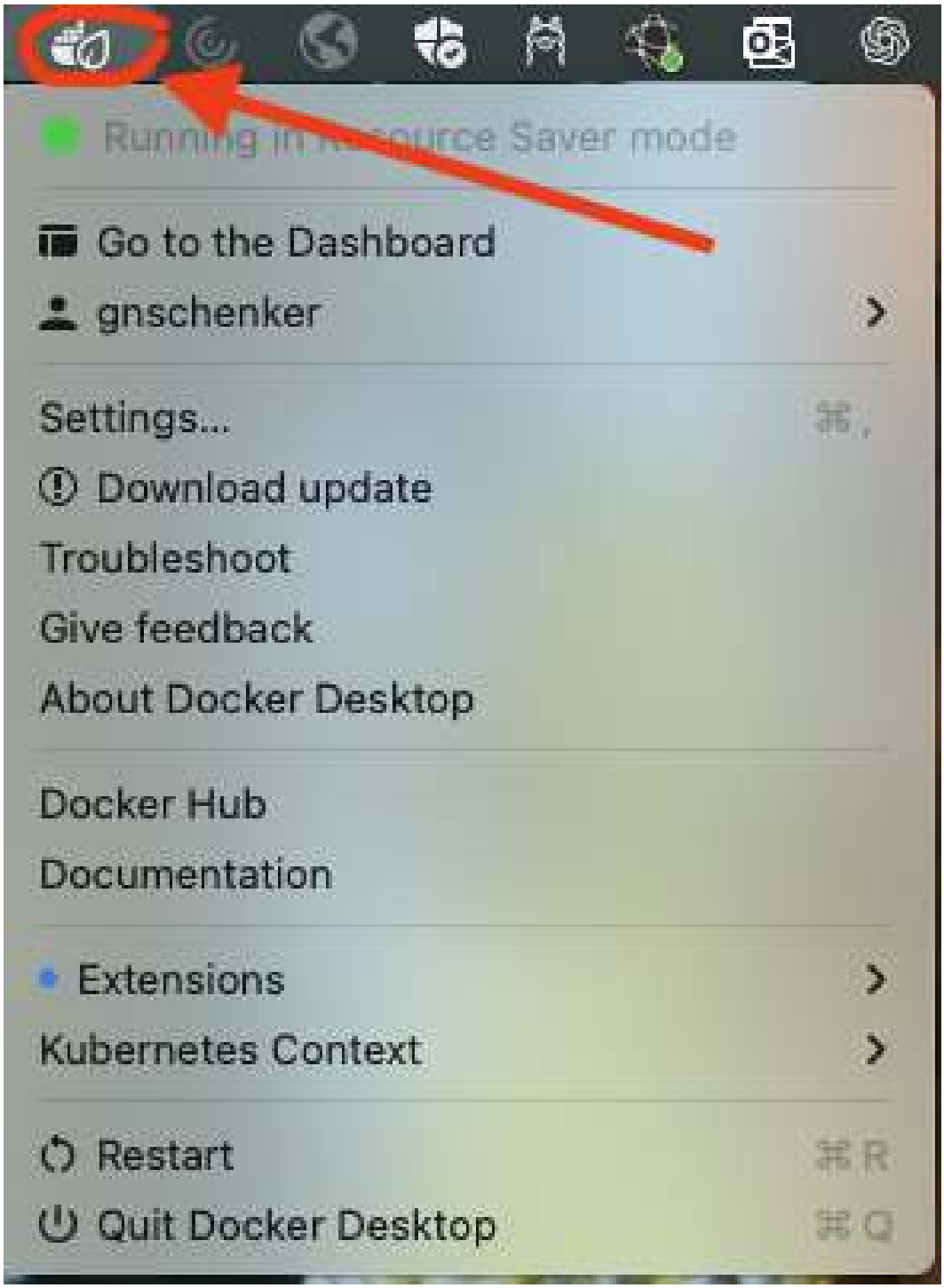


Figure 2.6: Context menu for Docker Desktop

2. From the menu, select **Dashboard**. The dashboard of Docker Desktop will open:

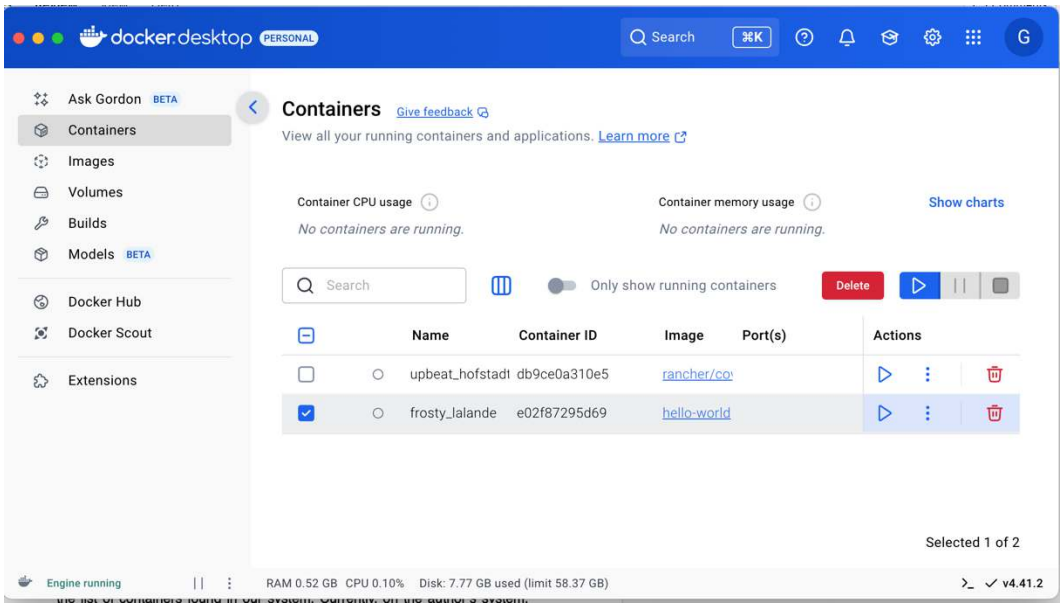


Figure 2.7: Dashboard of Docker Desktop

We can see that the dashboard has multiple tabs, indicated on the left-hand side of the view. Currently, the **Containers** tab is active. Consequently, we can see the list of containers found in our system. Currently, on the author's system, two have been found. If you inspect carefully, you will see that these are the containers that we previously created from the `hello-world` and `rancher/cowsay` Docker images. If you click on one of the entries, the details of this container will be displayed, and you will notice that they both have the status **Exited**.

Please take some time and explore this dashboard a bit. Don't worry if you get lost. It will all become much clearer as we proceed through the various chapters of this book.

1. When you're done exploring, close the dashboard window.

NOTE

Note

Closing the dashboard will not stop Docker Desktop. The application, as well as Docker Engine, will continue to run in the background. If, for some reason, you want to stop Docker on your system completely, you can select **Quit Docker Desktop** from the context menu shown in *Figure 2.6*.

Congratulations, you have successfully installed and tested Docker Desktop on your working computer! Now, let's continue with a few other useful tools.

Using Docker with WSL 2 on Windows

If you're a Windows 10 or 11 user, you can leverage the Windows Subsystem for Linux version 2 (WSL 2) to enjoy near-native Linux performance for containers. By default, Docker Desktop for Windows integrates tightly with WSL 2, eliminating the need for a separate Linux VM. To do so, please follow these steps:

1. **Enable WSL 2**: Make sure you have WSL 2 enabled on your system. You can install or upgrade WSL by following Microsoft's official documentation, typically involving enabling the **Windows Virtual Machine Platform** feature and installing a preferred Linux distribution from the Microsoft Store. You can find instructions here: `https://learn.microsoft.com/en-us/windows/wsl/install`
2. **Check Docker Desktop settings**: Once WSL 2 is enabled, open Docker Desktop and navigate to **Settings**. Under **General**, confirm that **Use the WSL 2 based engine** is switched on. This ensures Docker runs all containers via WSL 2 rather than Hyper-V or other backends.
3. **Run containers natively**: With WSL 2 activated, you can run Linux containers much more efficiently. Docker Desktop automatically manages resource allocation, filesystem mounting, and networking integration, giving you a smoother experience—similar to running Docker natively on a Linux machine.
4. **Advantages of WSL 2**:
    - **Improved performance**: Faster file I/O and near-native Linux speeds for Docker containers.
    - **Better resource management**: Lower overhead compared to older VM-based setups, and simpler memory/CPU balancing.
    - **Seamless filesystem integration**: Access your local Windows files or your Linux distribution's filesystem without complex sharing configurations.

5. **Troubleshooting and further details**: For deeper insights—such as customizing multiple WSL distributions or handling edge cases—you can refer to the Microsoft and Docker documentation. Since Docker Desktop and WSL 2 are jointly maintained by those teams, any platform-specific nuances are typically well documented in their respective guides.

By incorporating WSL 2, you will gain a more integrated and performant Docker workflow on Windows—without the extra complexity previously required by Docker Toolbox or dedicated Linux VMs.

Installing Docker Toolbox

Docker Toolbox has been available for developers for a few years. It precedes newer tools such as Docker Desktop. Toolbox allows a user to work very elegantly with containers on any macOS or Windows computer. Containers must run on a Linux host. Neither Windows nor macOS can run containers natively. Hence, we need to run a Linux VM on our laptop, where we can then run our containers. Docker Toolbox installs VirtualBox on our laptop, which is used to run the Linux VMs we need.

NOTE

Note

Docker Toolbox has been deprecated recently, and thus, we won't be discussing it further. For certain scenarios, it may still be of interest, though, which is why we are mentioning it here.

Enabling Kubernetes on Docker Desktop

Docker Desktop comes with integrated support for Kubernetes.

NOTE

What is Kubernetes?

Kubernetes is a powerful platform for automating the deployment, scaling, and management of containerized applications. Whether you're a developer, DevOps engineer, or system administrator, Kubernetes provides the tools and abstractions you need to manage your containers and applications in a scalable and efficient manner.

This support is turned off by default. But worry not – it is very easy to enable:

1. Open the dashboard of Docker Desktop.
2. In the top-left corner, select the *cog wheel* icon. This will open the settings page.
3. On the left-hand side, select the **Kubernetes** tab and then click the **Enable Kubernetes** toggle:

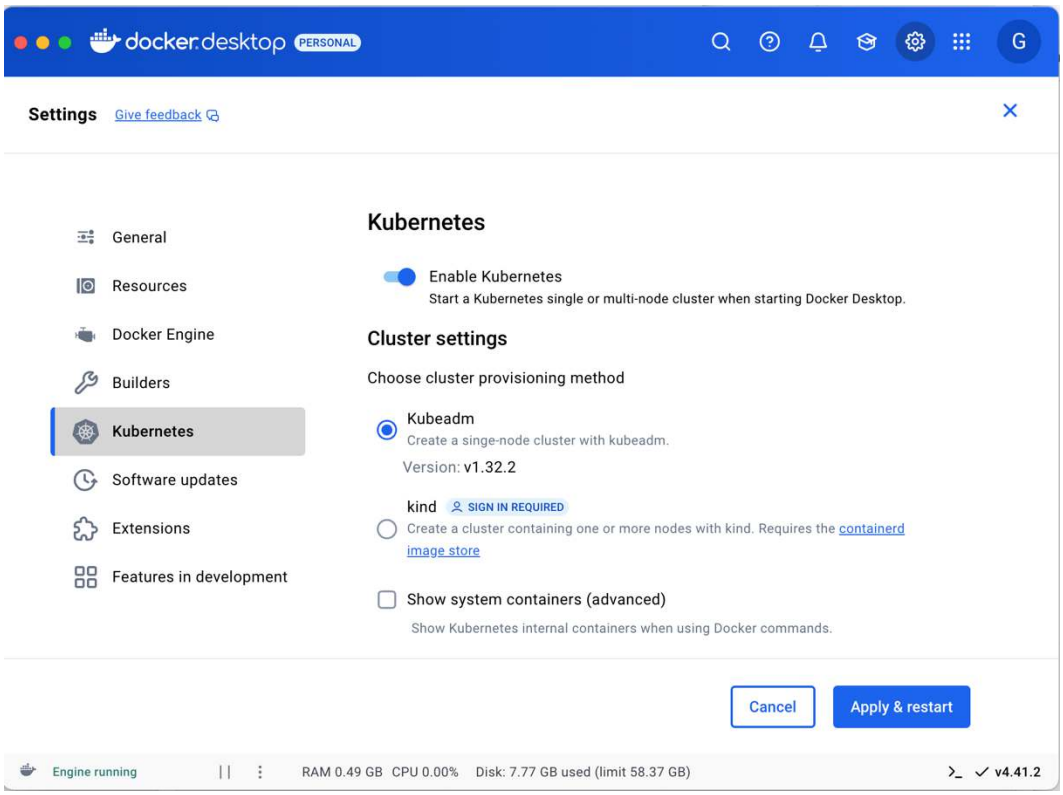


Figure 2.8: Enabling Kubernetes on Docker Desktop

4. Click the **Apply & restart** button.

Now, you will have to be patient since Docker will download all the supporting infrastructure and then start Kubernetes.

Once Docker has restarted, you are ready to use Kubernetes. Please refer to the *Installing minikube* section, later in this chapter, on how to test Kubernetes.

NOTE

Note

Only a single-node cluster is supported. For support for multi-node setups, you should use kind or minikube, as described later in this chapter.

Installing Podman

Podman is an open source, daemonless container engine that serves as an alternative to Docker. It is largely compatible with Docker's CLI, yet it offers some distinct advantages, such as running containers in rootless mode for improved security. Please follow these instructions to install Podman on your system:

Installing Podman on MacOS

To install Podman on a MacOS-based system, please follow these instructions:

1. **Install Podman via Homebrew**: In a Terminal, execute the following:

   ```
   $ brew install podman
   ```

2. **Initialize the Podman machine**: To set up a lightweight virtual machine for running containers, run this:

   ```
   $ podman machine init
   ```

3. **Start the Podman machine**: Launch your Podman-based container environment with the following command:

   ```
   $ podman machine start
   ```

4. **Verify the installation**: Confirm that Podman is installed by checking its version:

   ```
   $ podman --version
   ```

   Or, view the system details using the following:

   ```
   $ podman info
   ```

Installing Podman on Windows

On a Windows-based system, use the following instructions:

1. **Install Podman using Chocolatey**: In a terminal, execute the following:

   ```
   $ choco install podman -y
   ```

2. **Initialize the Podman machine**: To set up a lightweight virtual machine for running containers, run this:

```
$ podman machine init
```

3. **Start the Podman machine**: Launch your Podman-based container environment with the following command:

```
$ podman machine start
```

4. **Verify the installation**: Confirm that Podman is installed by checking its version:

```
$ podman --version
```

   Or, view the system details using the following:

```
$ podman info
```

Installing Podman on Linux

To install Podman on a Debian- or Ubuntu-based Linux machine, please follow these instructions:

1. **Update your package index**: Open a terminal and run the following:

```
$ sudo apt-get update
```

2. **Install Podman**: Execute the following command:

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

3. **Verify the installation**: Check the installed version to confirm Podman is ready:

```
$ podman --version
```

These instructions set up Podman's environment and verify that your system is ready to run containers. While Podman's rootless, daemonless design offers improved security and resource efficiency, remember that its integration on Windows may require additional configuration compared to Docker Desktop. For more information, please consult the Podman getting started page at https://podman.io/get-started.

After successfully installing Podman, let's compare it with Docker Desktop. Here are some of the pros and what are potential cons:

Pros of Podman:

- **Daemonless architecture**: Podman runs without a background daemon, reducing the attack surface and resource overhead
- **Rootless operation**: It allows running containers without root privileges, enhancing security
- **Docker CLI compatibility**: Most Docker commands work with Podman, making it easier to switch without a steep learning curve
- **Lightweight**: Podman typically consumes fewer system resources than Docker Desktop

Cons of Podman:

- **Limited GUI tools**: Unlike Docker Desktop, which provides a comprehensive graphical interface, Podman relies mainly on the CLI (although third-party GUIs exist)
- **Platform support**: Docker Desktop offers polished desktop applications for Windows and macOS, whereas Podman's support on non-Linux platforms may require additional configuration or workarounds
- **Ecosystem and integration**: Docker Desktop benefits from a mature ecosystem with broad third-party integrations and native support in many development tools

Now that we are able to run containers on our system, we also want to install some tooling for container orchestration.

Installing minikube

If you are using Docker Desktop, you may not need minikube at all since the former already provides out-of-the-box support for Kubernetes. If you cannot use Docker Desktop or, for some reason, you only have access to an older version of the tool that does not yet support Kubernetes, then it is a good idea to install minikube. **minikube** by default provisions a single-node Kubernetes cluster on your workstation and is accessible through **kubectl**, which is the command-line tool used to work with Kubernetes. Note that minikube is also able to provision multi-node clusters on your system.

Installing minikube on Linux, macOS, and Windows

To install minikube for Linux, macOS, or Windows, navigate to the following link: `https://kubernetes.io/docs/tasks/tools/install-minikube/`.

Follow the instructions carefully. Specifically, do the following:

1. Make sure you have a hypervisor installed, as described in the section marked inside the box in *Figure 2.9*:

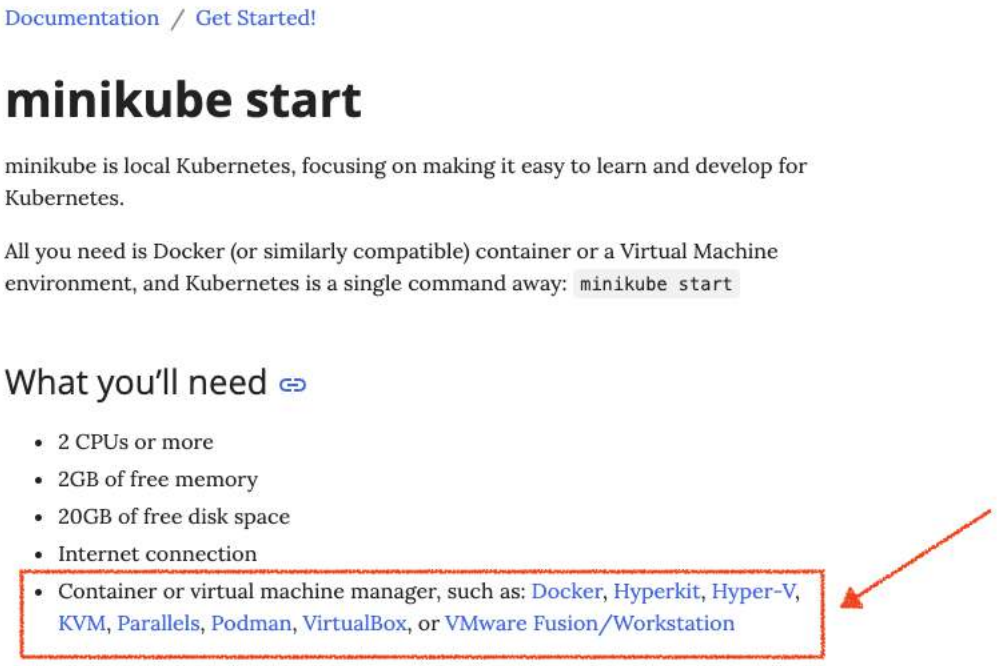


Figure 2.9: Prerequisites for minikube

2. Under **1 Installation**, select the combination that is valid for you. As an example, you can see the author's' selection for a *MacBook Pro M2 laptop* as the target machine:

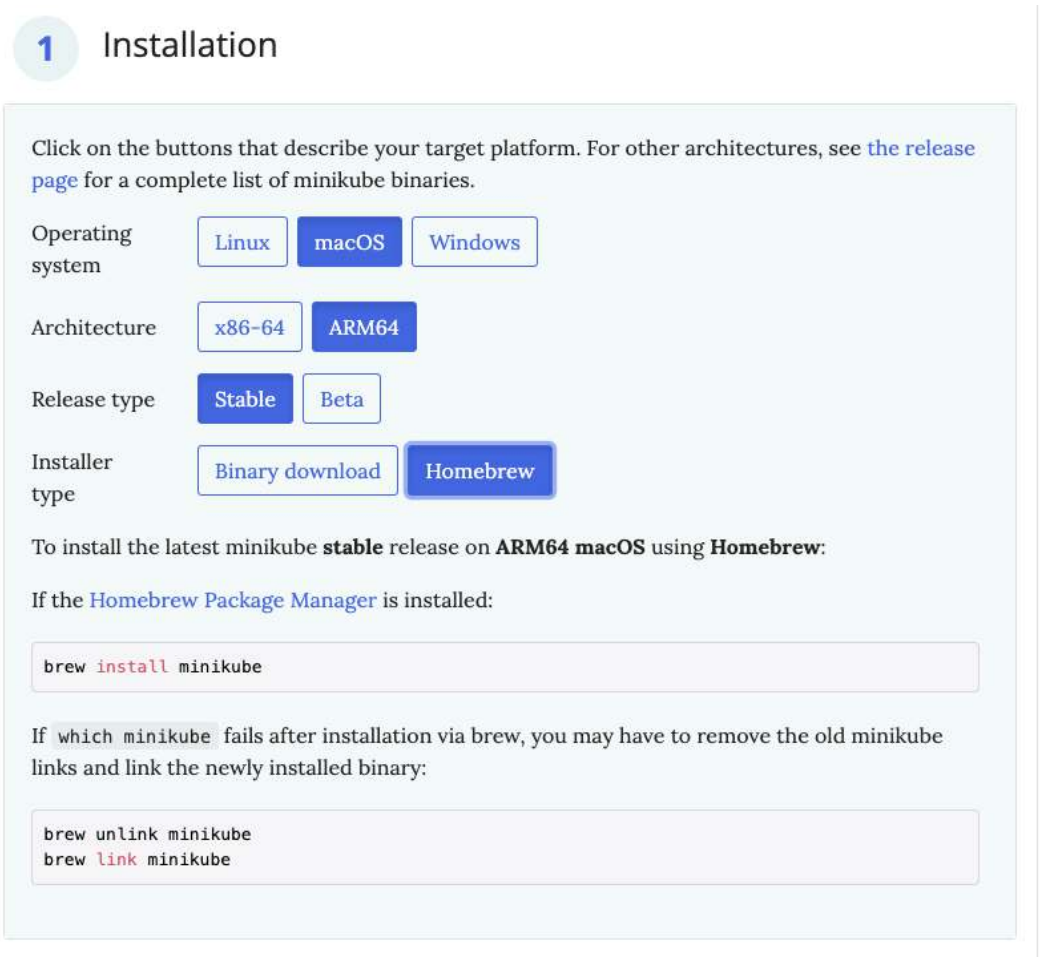


Figure 2.10: Selecting the correct installation for minikube

After preparing our system for the installation of minikube and selecting the appropriate method of installation, we will now demonstrate the actual installation on a MacBook Pro.

Installing minikube for MacBook Pro M2 using Homebrew

Follow these steps:

1. In a Terminal window, execute the steps shown previously, in *Figure 2.10*. In the author's' case, this is as follows:

```
$ brew install minikube
```

2. Test the installation with the following command:

```
$ minikube version
minikube version: v1.35.0
commit: dd5d320e41b5451cdf3c01891bc4e13d189586ed
```

3. Now, we're ready to start a cluster. Let's start with the default:

```
$ minikube start
```

This will output something like this:

```
minikube v1.35.0 on Darwin 15.3.1 (arm64)
Automatically selected the docker driver
Using Docker Desktop driver with root privileges
Starting "minikube" primary control-plane node in "minikube" cluster
Pulling base image v0.0.46 ...
Downloading Kubernetes v1.32.0 preload ...
  > gcr.io/k8s-minikube/kicbase...:  452.84 MiB / 452.84 MiB  100.00% 21.62 M
  > preloaded-images-k8s-v18-v1...:  303.97 MiB / 314.92 MiB  96.52% 14.01 Mi
```

NOTE

Note

minikube allows you to define single- and multi-node clusters.

1. The first time you do this, it will take a while since minikube needs to download all the Kubernetes binaries. When it's done, the last line of the output on your screen should be something like this:

```
Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

Great, we have successfully installed minikube on our system! Let's try to play with minikube a bit by creating a cluster and running our first application in a container on it. Don't worry if the following commands do not make a lot of sense to you at this time. We will discuss everything in this book in the coming chapters.

Testing minikube and kubectl

Let's start. Please follow these steps carefully:

1. Let's try to access our cluster using kubectl. First, we need to make sure we have the correct context selected for kubectl. If you have previously installed Docker Desktop and now minikube, you can use the following command:

```
$ kubectl config get-contexts
```

You should see something similar to this:

```
❯ kubectl config get-contexts
CURRENT   NAME             CLUSTER          AUTHINFO         NAMESPACE
          kind-my-cluster  kind-my-cluster  kind-my-cluster
*         minikube         minikube         minikube         default
```

Figure 2.11: List of contexts for kubectl after installing minikube

The asterisk next to the context called **minikube** tells us that this is the current context. Thus, when using kubectl, we will work with the new cluster created by minikube.

2. Now, let's see how many nodes our cluster has with this command:

```
$ kubectl get nodes
```

You should get something similar to this. Note that the version shown could differ in your case:

```
❯ kubectl get nodes
NAME       STATUS   ROLES           AGE    VERSION
minikube   Ready    control-plane   4m1s   v1.32.0
```

Figure 2.12: Showing the list of cluster nodes for the minikube cluster

Here, we have a single-node cluster. The node's role is that of the control plane, which means it is a master node. A typical Kubernetes cluster consists of a few master nodes and many worker nodes. The version of Kubernetes we're working with here is `1.32.0`.

3. Now, let's try to run something on this cluster. We will use Nginx, a popular web server, for this. If you have previously cloned the GitHub repository accompanying this book to the `The-Ultimate-Docker-Container-Book-v4` folder in your home directory (~), then you should find a folder setup inside this folder that contains a `.yaml` file that we're going to use for this test:
    1. Open a new Terminal window.
    2. Navigate to the `The-Ultimate-Docker-Container-Book-4` folder:

```
$ cd ~/The-Ultimate-Docker-Container-Book-v4
```

    3. Create a pod running Nginx with the following command:

```
$ kubectl apply -f setup/nginx.yaml
```

You should see this output:

```
pod/nginx created
```

4. We can double-check if the pod is running with kubectl:

```
$ kubectl get pods
```

We should see this:

```
NAME     READY   STATUS     RESTARTS   AGE
nginx    1/1      Running   0          29s
```

This indicates that we have 1 pod with Nginx running and that it has been restarted 0 times.

4. To access the Nginx server, we need to expose the application running in the pod with the following command:

```
$ kubectl expose pod nginx --type=NodePort --port=80
```

This is the only way can we access Nginx from our laptop – for example, via a browser. With the preceding command, we're creating a Kubernetes service, as indicated in the output generated for the command:

```
service/nginx exposed
```

5. We can use kubectl to list all the services defined in our cluster:

```
$ kubectl get services
```

We should see something similar to this:

```
❯ kubectl get services
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP        12m
nginx        NodePort    10.99.183.121   <none>        80:30706/TCP   36s
```

Figure 2.13: List of services on the minikube cluster

In the preceding output, we can see the second service, called **nginx**, which we just created. The service is of the **NodePort** type; port **80** of the pod had been mapped to port **30706** of the cluster node of our

Kubernetes cluster in minikube. Note that, in your case, the mapped port may be different!

6. Now, we can use minikube to make a tunnel to our cluster and open a browser with the correct URL to access the Nginx web server. Use this command:

```
$ minikube service nginx
```

The output in your Terminal window will be as follows:

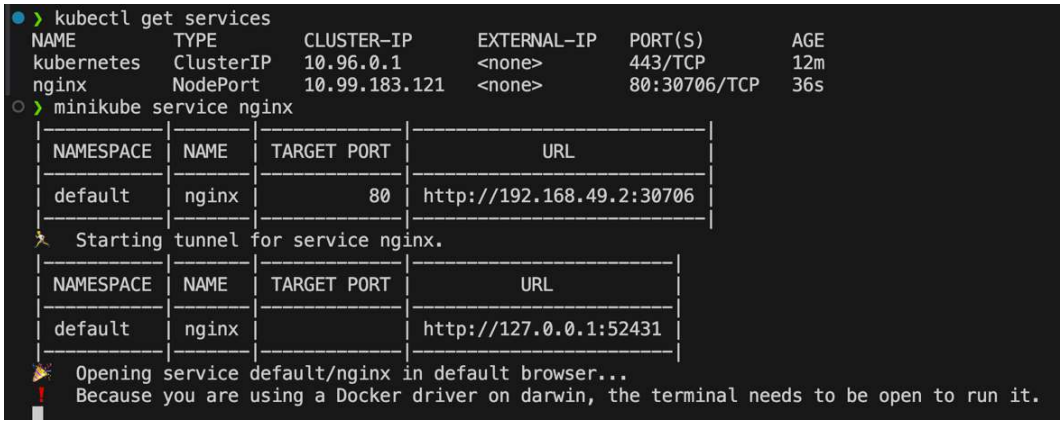


Figure 2.14: Opening access to the Kubernetes cluster on minikube

The preceding output shows that minikube created a tunnel for the Nginx service listening on node port `30706` to port `52431` on the host, which is on our laptop.

7. A new browser tab should have been opened automatically and should have navigated you to `http://127.0.0.1:52431`. You should see the welcome screen of Nginx:

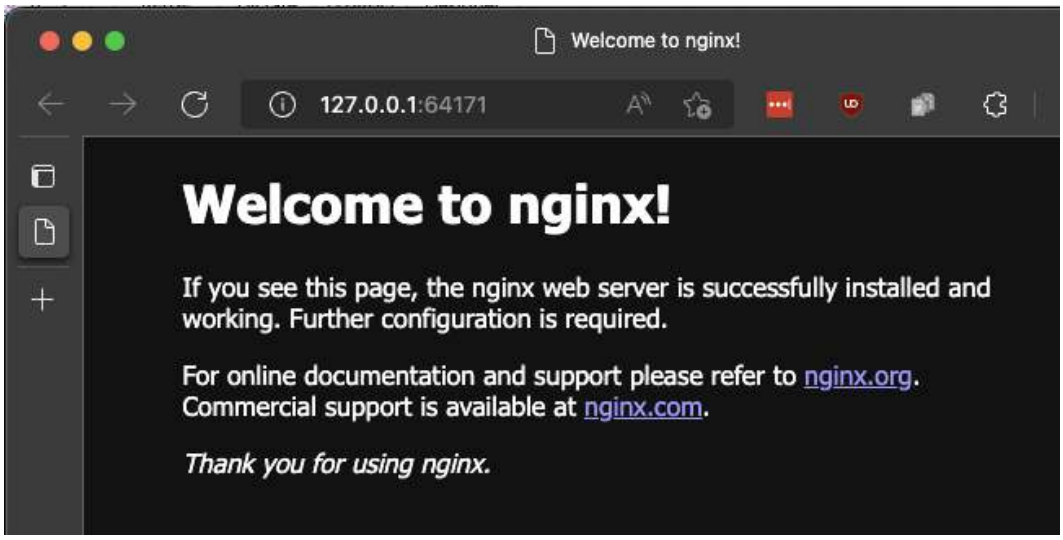


Figure 2.15: Welcome screen of Nginx running on a Kubernetes cluster on minikube

Wonderful, we have successfully run and accessed an Nginx web server on our little single-node Kubernetes cluster on minikube! Once you are done playing around, it is time to clean up:

1. Stop the tunnel to the cluster by pressing *Ctrl* + *C* inside your Terminal window.
2. Delete the nginx service and pod on the cluster:

```
$ kubectl delete service nginx
$ kubectl delete pod nginx
```

3. Stop the cluster with the following command:

```
$ minikube stop
```

You should see this:

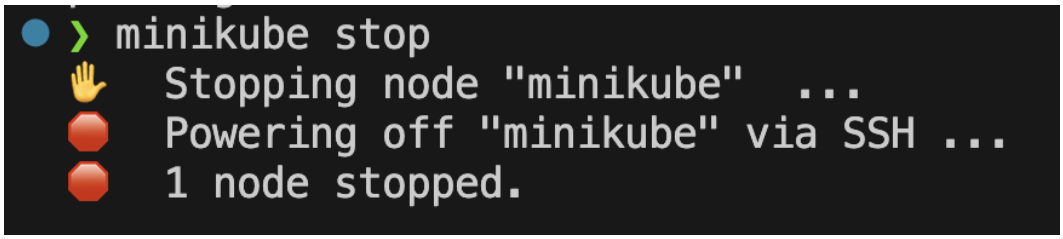


Figure 2.16: Stopping minikube

We have installed minikube and created and tested a single-node Kubernetes cluster with it. Now, let's demonstrate how we can use minikube to create a multi-node cluster.

Working with a multi-node minikube cluster

At times, testing with a single-node cluster is not enough. Worry not – minikube has you covered. Follow these instructions to create a true multi-node Kubernetes cluster in minikube:

1. If we want to work with a cluster consisting of multiple nodes in minikube, we can use this command:

   ```
   $ minikube start --nodes 3 -p demo
   ```

   The preceding command creates a cluster with three nodes and calls it `demo`.

2. Use kubectl to list all your cluster nodes:

   ```
   $ kubectl get nodes
   NAME            STATUS   ROLES                       AGE   VERSION
   demo            Ready    control-plane   84s   v1.32.0
   demo-m02   Ready     <none>                  45s   v1.32.0
   demo-m03   Ready     <none>                  22s   v1.32.0
   ```

   We have a three-node cluster where the `demo` node is a master node, and the two remaining nodes are work nodes.

3. We are not going to go any further with this example here, so use the following command to stop the cluster:

   ```
   $ minikube stop -p demo
   ```

4. Delete all the clusters on your system with this command:

   ```
   $ minikube delete --all
   ```

   This will delete the default cluster (called minikube) and the demo cluster in our case.

With this, we will move on to the next interesting tool that's useful when working with containers and Kubernetes. You should have this installed and readily available on your work computer.

Installing kind

kind is another popular tool that can be used to run a multi-node Kubernetes cluster locally on your machine. It is super easy to install and use. Let's go:

1. Use the appropriate package manager of your platform to install kind. You can find more detailed information about the installation process here: `https://kind.sigs.k8s.io/docs/user/quick-start/`:
    1. On MacOS, use Homebrew to install kind with the following command:

        ```
        $ brew install kind
        ```

    2. On a Windows machine, use Chocolatey to do the same with this command:

        ```
        $ choco install kind -y
        ```

    3. Finally, on a Linux machine, you can use the following script to install kind from its binaries:

        ```
        $ curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.17.0/kind-linux-amd64
        $ chmod +x ./kind
        $ sudo mv ./kind /usr/local/bin/kind
        ```

2. Once kind has been installed, test it with the following command:

    ```
    $ kind version
    ```

    If you're on a Mac, it should output something like this:

    ```
    kind v0.27.0 go1.24.0 darwin/arm64
    ```

3. Now, try to create a simple Kubernetes cluster consisting of one master node and two worker nodes. Use this command to accomplish this:

    ```
    $ kind create cluster
    ```

    After some time, you should see this output:

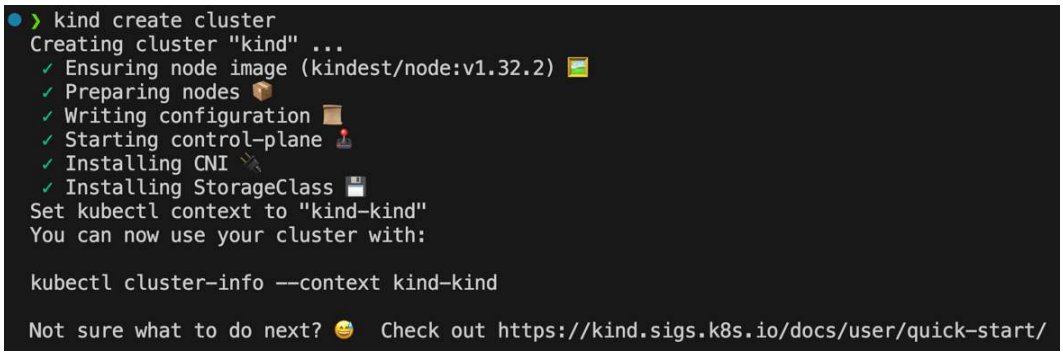


Figure 2.17: Creating a cluster with Kind

4. To verify that a cluster has been created, use this command:

```
$ kind get clusters
```

   The preceding output shows that there is exactly one cluster called **kind**, which is the default name.

5. We can create an additional cluster with a different name using the `--name` parameter, like so:

```
$ kind create cluster --name demo
```

6. Listing the clusters will then show this:

```
$ kind get clusters
demo
kind
```

   This works as expected.

   To clean up, run the following command:

```
$ kind delete clusters -all
```

With this, we have installed and tested a second version of a local Kubernetes orchestrator. Let's continue with some additional exercises involving minikube and kind.

Testing kind and minikube

Now that we have used kind to create two sample clusters, let's use kubectl to play with one of the clusters and run the first application on it. We will be using Nginx for this, similar to what we did with minikube:

1. Let's first create a cluster with minikube and one with kind:

```
$ minikube start -p minikube-demo
$ kind create cluster --name kind-demo
```

2. We can now use kubectl to access and work with the clusters we just created. While creating a cluster, kind also updated the configuration file for our kubectl. We can double-check this with the following command:

```
$ kubectl config get-contexts
```

   It should produce the following output:

```
❯ kubectl config get-contexts
CURRENT   NAME             CLUSTER          AUTHINFO         NAMESPACE
*         kind-kind-demo   kind-kind-demo   kind-kind-demo
          minikube-demo    minikube-demo    minikube-demo    default
```

Figure 2.18: List of contexts defined for kubectl

   You can see that the `minikube-demo` and `kind-kind-demo` clusters are part of the list of known clusters and that the `kind-kind-demo` cluster is the current context for kubectl.

3. Use the following command to make the `minikube-demo` cluster your current cluster if the asterisk indicates that another cluster is current:

```
$ kubectl config use-context minikube-demo
```

4. Let's list all the nodes of the `minikube-demo` cluster:

```
$ kubectl get nodes
```

   The output should be like this:

```
❯ kubectl get nodes
NAME             STATUS   ROLES           AGE    VERSION
minikube-demo    Ready    control-plane   7m4s   v1.32.0
```

Figure 2.19: Showing the list of nodes on the minikube cluster

5. Now, let's try to run the first container on this cluster. We will use our trusted Nginx web server, as we did earlier. Use the following command to run it:

```
$ kubectl apply -f setup/nginx.yaml
```

The output should be as follows:

```
pod/nginx created
```

6. To access the Nginx server, we need to do port forwarding using kubectl. Use this command to do so:

```
$ kubectl port-forward nginx 8080 80
```

The output should look like this:

```
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

7. Open a new browser tab and navigate to `http://localhost:8080`; you should see the welcome screen of Nginx:

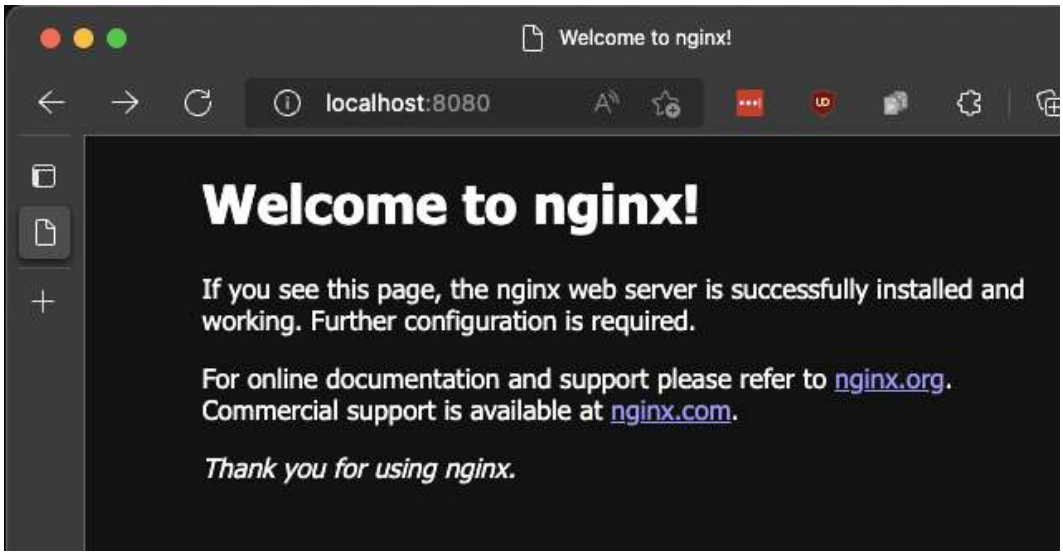


Figure 2.20: Welcome screen of Nginx running on a Kind cluster

8. Once you've finished playing with Nginx, use this command to delete the pod from the cluster:

```
$ kubectl delete -f setup/nginx.yaml
```

9. Before we continue, let's clean up and delete the two clusters we just created:

```
$ minikube delete -all
$ kind delete cluster --name kind-demo
```

With this, we have installed all the tools that we will need to successfully work with containers on our local machine.

Summary

In this chapter, we focused on establishing and configuring a robust working environment tailored for efficiently managing Docker containers—a setup that benefits developers, DevOps engineers, and operations professionals alike.

We began by emphasizing the value of a package manager, a fundamental tool for quickly installing and maintaining the many applications and utilities needed in a modern development workflow. Next, we stressed the importance of using a reliable shell for scripting (such as Bash or PowerShell) along with a powerful code editor such as Visual Studio Code, which was enhanced with essential extensions and even AI-powered development tools for smarter coding.

The chapter then guided you through installing container engines—primarily Docker for Desktop and Podman—providing the means to run and test containers natively on your machine. Finally, we delved into local Kubernetes orchestration by setting up and testing tools such as minikube and kind, which allowed us to simulate both single-node and multi-node clusters. This local setup empowered you to experiment with containerized applications in a controlled environment, laying the groundwork for more complex orchestration tasks in later chapters.

In the next chapter, we're going to learn important facts about containers. For example, we will explore how we can run, stop, list, and delete containers, but more than that, we will also dive deep into the anatomy of containers.

Further reading

Consider the following links for further reading:

- *Chocolatey – The Package Manager for Windows*: `https://chocolatey.org/`
- *Install Docker Toolbox on Windows*: `https://dockr.ly/2nuZUkU`
- *Run Docker on Hyper-V with Docker Machine*: `http://bit.ly/2HGMPiI`

- *Developing inside a Container*: https://code.visualstudio.com/docs/remote/containers

Questions

Based on what was covered in this chapter, please answer the following questions:

1. Why would we care about installing and using a package manager on our local computer?
2. With Docker Desktop, you can develop and run Linux containers.
    a. True
    b. False
3. Why are good scripting skills (such as Bash or PowerShell) essential for the productive use of containers?
4. Why is it critical to test your Docker installation using commands such as `docker version` and `docker container run hello-world`?
5. How do local Kubernetes tools such as minikube and kind benefit containerized application development?
6. What are the pros and cons of using Docker CLI, Docker Desktop, and Podman for container management?

Answers

The following are the answers to this chapter's questions:

1. Package managers such as `apk`, `apt`, or `yum` on Linux systems, Homebrew on MacOS, and Chocolatey on Windows make it easy to automate the installation of applications, tools, and libraries. It is a much more repeatable process when an installation happens interactively, and the user has to click through a series of views.
2. The answer is *True*. Yes, with Docker for Windows, you can develop and run Linux containers. It is also possible, but not discussed in this book, to develop and run native Windows containers with this edition of Docker Desktop. With the macOS and Linux editions, you can only develop and run Linux containers.

3. Scripts are used to automate processes and hence avoid human errors. Building, testing, sharing, and running Docker containers are tasks that should always be automated to increase their reliability and repeatability.
4. Running these tests confirms that Docker Engine is installed correctly and operational. The `docker version` command verifies that both the client and server components are communicating properly, while running `docker container run hello-world` ensures that your system can download images and execute containers successfully—serving as a practical check that your entire container environment is set up as expected.
5. Tools such as minikube and kind allow you to run a local Kubernetes cluster on your development machine. This enables testing of container orchestration, deployment strategies, and multi-node configurations without relying on remote cloud clusters. By simulating real-world Kubernetes environments locally, developers can experiment, troubleshoot, and refine their applications before moving to production setups.
6. Here are the pros and cons of using Docker CLI, Docker Desktop, and Podman for container management:
   Docker CLI:

   Pros:

   - Provides a direct and lightweight way to manage containers via commands
   - Highly scriptable, which is ideal for automating workflows and integrating into CI/CD pipelines

   Cons:

   - Has a steeper learning curve for beginners since it requires familiarity with command-line operations
   - Lacks a graphical interface, which might limit ease of use for visual management tasks

   Docker Desktop:

   Pros:

- Offers an integrated, user-friendly GUI that simplifies container management, including access to dashboards and Kubernetes integration
- Provides a complete environment (Docker Engine, CLI, and additional tools) in one package, easing setup on macOS and Windows

Cons:

- More resource-intensive compared to using just the CLI, which might impact performance on lower-spec machines
- Limited to certain operating systems (primarily macOS, Windows, and recently Linux) and may not suit all environments

Podman:

Pros:

- Operates in a daemonless mode and supports rootless container management, offering enhanced security and lower resource overhead
- Maintains a high degree of Docker CLI compatibility, easing the transition for Docker users

Cons:

- Lacks a mature, integrated GUI like Docker Desktop, potentially making it less accessible for those who prefer visual tools
- Ecosystem and community support might not be as extensive as Docker's, which can affect available third-party integrations and tooling

This comparison highlights that the choice among these tools depends on your specific needs—whether you prioritize simplicity and automation (Docker CLI), a full-featured graphical experience (Docker Desktop), or enhanced security and lightweight operation (Podman).

3

Mastering Containers

Join our book community on Discord:

https://packt.link/mqfS2

In the previous chapter, you learned how to optimally prepare your working environment for the productive and frictionless use of Docker. In this chapter, we are going to get our hands dirty and learn everything that is important to know when working with containers.

Here are the topics we're going to cover in this chapter:

- Running the first container
- Starting, stopping, and removing containers
- Inspecting containers
- Exec into a running container
- Attaching to a running container
- Retrieving container logs
- The anatomy of containers

After finishing this chapter, you will be able to do the following things:

- Run, stop, and delete a container based on an existing image, such as Nginx, BusyBox, or Alpine
- List all containers on the system
- Inspect the metadata of a running or stopped container
- Retrieve the logs produced by an application running inside a container
- Run a process such as `/bin/sh` in an already-running container
- Attach a terminal to an already-running container
- Explain in your own words, to an interested layperson, the underpinnings of a container

- Explain how Linux namespaces provide process isolation and how cgroups manage resource allocation, forming the foundation of containerization

Technical requirements

For this chapter, you should have Docker for Desktop installed on your Linux workstation, macOS, or Windows PC. On macOS, use the Terminal application, and on Windows, use the PowerShell console or Git Bash, to try out the commands you will be learning.

Running the first container

Before we start, we want to make sure that Docker is installed correctly on your system and ready to accept your commands. Open a new terminal window and type in the following command (note: do not type the `$` sign, as it is a placeholder for your prompt):

```
$ docker version
```

If everything works correctly, you should see the version of the Docker client and server installed on your laptop output in the terminal. At the time of writing, it looks like this:

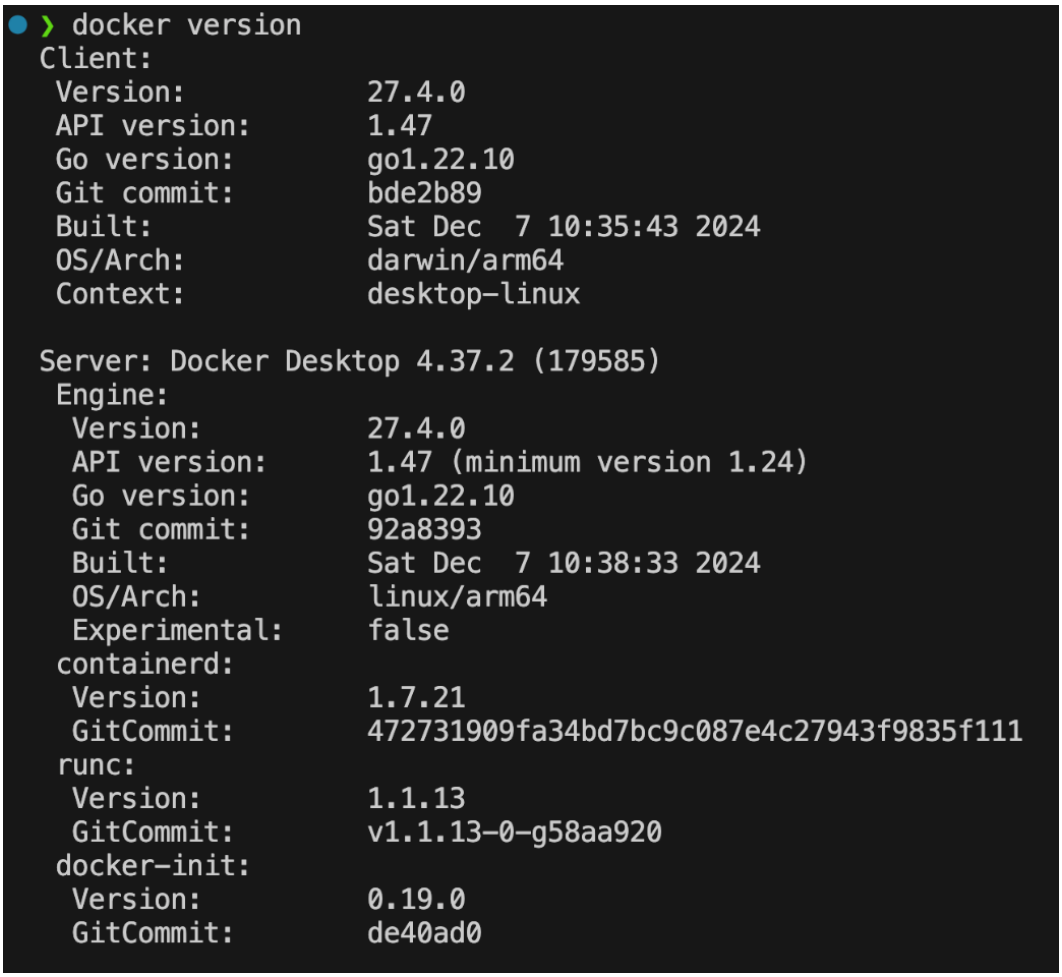


Figure 3.1 – Output of the docker version command

As you can see, I have version `27.4.0` installed on the author's MacBook Pro M2 laptop.

If this doesn't work for you, then something with your installation is not right. Please make sure that you have followed the instructions in the previous chapter on how to install Docker Desktop on your system.

So, you're ready to see some action. Please type the following command into your terminal window and hit the *Return* key:

```
$ docker container run alpine echo "Hello World"
```

When you run the preceding command the first time, you should see an output in your terminal window like this:

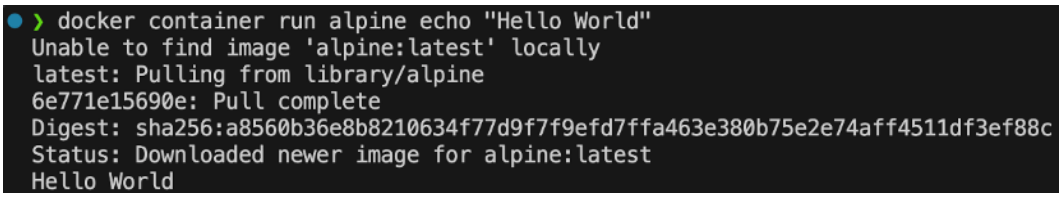


Figure 3.2 – Running an Alpine container for the first time

Now that was easy! Let's try to run the very same command again:

```
$ docker container run alpine echo "Hello World"
```

The second, third, or nth time you run the preceding command, you should see only this output in your terminal:

```
Hello World
```

Try to reason why the first time you run a command you see a different output than all of the subsequent times. But don't worry if you can't figure it out; we will explain the reasons in detail in the following sections of this chapter.

Starting, stopping, and removing containers

You successfully ran a container in the previous section. Now, we want to investigate in detail what exactly happened and why. Let's look again at the command we used:

```
$ docker container run alpine echo "Hello World"
```

This command contains multiple parts. First and foremost, we have the word `docker`. This is the name of the Docker **Command-Line Interface** (**CLI**) tool, which we are using to interact with the Docker engine that is responsible for running containers. Next, we have the word `container`, which indicates the context we are working with, such as `container`, `image`, or `volume`. As we want to run a container, our context is the word `container`. Next is the actual command we want to execute in the given context, which is `run`.

Let me recap – so far, we have `docker container run`, which means, "*hey Docker, we want to run a container*."

Now we also need to tell Docker which container to run. In this case, this is the so-called `alpine` container.

NOTE

> **Note**
>
> `alpine` is a minimal Docker image based on Alpine Linux with a complete package index and is only about 8 MB in size. It is an official image supported by the Alpine open source project and Docker.

Finally, we need to define what kind of process or task will be executed inside the container when it is running. In our case, this is the last part of the command, `echo "Hello World"`.

The following figure may help you to get a better idea of the whole thing:

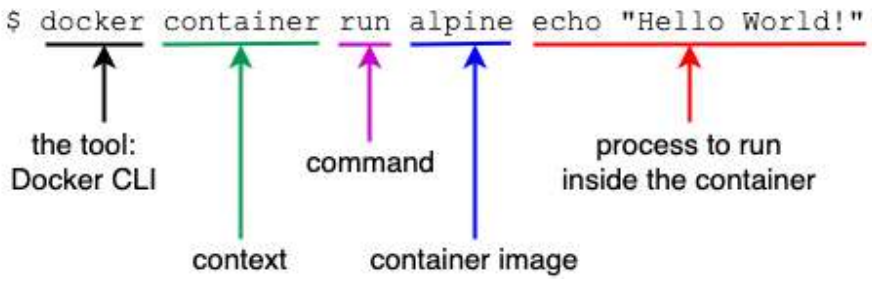


Figure 3.3 – Docker run command explained

Now that we have understood the various parts of a command to run a container, let's try to run another container with a different process executed inside it. Type the following command into your terminal:

```
$ docker container run quay.io/centos/centos echo "Hello from centos"
```

You should see output in your terminal window similar to the following:

```
❯ docker container run quay.io/centos/centos echo "Hello from centos"
Unable to find image 'quay.io/centos/centos:latest' locally
latest: Pulling from centos/centos
4ff8fa80ba5d: Pull complete
Digest: sha256:51ca701a9cd3b148b15b421e4bc75515108df15b333c4a61babc185e64744324
Status: Downloaded newer image for quay.io/centos/centos:latest
Hello from centos
```

Figure 3.4 – Running the echo command inside a CentOS container

What changed is that, this time, the container image we're using is `quay.io/centos/centos` and the process we're executing inside the `centos` container is `echo "Hello from centos"`.

> NOTE
>
> **Note**

> `centos` is the official Docker image for CentOS Linux, a community-supported distribution derived from sources freely provided to the public by Red Hat for **Red Hat Enterprise Linux** (**RHEL**). It has been deprecated, so we are now using the one from an alternative registry: **quay.io**.

Let's analyze the output in detail. The first line is as follows:

```
Unable to find image 'quay.io/centos/centos:latest' locally
```

This tells us that Docker didn't find an image named `quay.iy/centos/centos:latest` in the local cache of the system. So, Docker knows that it has to pull the image from some registry where container images are stored. By default, your Docker environment is configured so that images are pulled from Docker Hub at `docker.io`. But this time, we explicitly define that we want to pull from the registry at `quay.io`. This is expressed by the second line, as follows:

```
latest: Pulling from centos/centos
```

The next three lines of output are as follows:

```
4ff8fa80ba5d: Pull complete
Digest: sha256:51ca701a9cd3b148b15b421e4bc75515108df15b333c4a61babc185e64744324
Status: Downloaded newer image for quay.io/centos/centos:latest
```

This tells us that Docker has successfully pulled the `centos:latest` image from `quay.io`. The last lines of the output are generated by the process we ran inside the container, which is the `echo` tool in this case. If you have been attentive so far, then you might have noticed the `latest` keyword occurring a few times. Each image has a version (also called `tag`), and if we don't specify a version explicitly, then Docker automatically assumes it is `latest`.

If we run the preceding container again on our system, the first five lines of the output will be missing since, this time, Docker will find the container image cached locally and hence won't have to download it first. Try it out and verify what I just told you.

> NOTE
>
> **What exactly happens when you run a container?**

> When you execute the `docker container run` command, Docker performs several actions to create and start a new container from the specified image. First, Docker checks if the requested image is available locally; if not, it pulls the image from the configured registry. Once the image is available, Docker creates a new container by allocating a filesystem and setting up a network interface. It then assigns an IP address to the container and sets up port mappings as specified. After configuring the container's environment, Docker starts the container by executing the specified command. This process ensures that the application within the container runs in an isolated and consistent environment.
>
> Worry not, all this and more will be explained in detail in the coming chapters of this book.

Running a random trivia question container

For the subsequent sections of this chapter, we need a container that runs continuously in the background and produces some interesting output. That's why we have chosen an algorithm that produces random trivia questions. The API that produces free random trivia can be found at https://the-trivia-api.com.

Now, the goal is to have a process running inside a container that produces a new random trivia question every 2 seconds and outputs the question to `STDOUT`. The following script will do exactly that:

```
while :
do
    curl -s https://the-trivia-api.com/v2/questions\?limit\=1 | jq '.[0].question'
    sleep 2
done
```

If you are using PowerShell, the preceding command can be translated to the following:

```
while ($true) {
  Invoke-WebRequest -Uri "https://the-trivia-api.com/v2/questions\?limit\=1" -Method GET -
UseBasicParsing |
  Select-Object -ExpandProperty Content |
  ConvertFrom-Json |
  Select-Object -ExpandProperty 0 |
  Select-Object -ExpandProperty question
```

```
    Start-Sleep -Seconds 2
}
```

NOTE

Note

The `ConvertFrom-Json` cmdlet requires that the `Microsoft.PowerShell.Utility` module be imported. If it's not already imported, you'll need to run `Import-Module Microsoft.PowerShell.Utility` before running the script.

Try it in a terminal window. Stop the script by pressing *Ctrl* + *C*. The output should look similar to this:

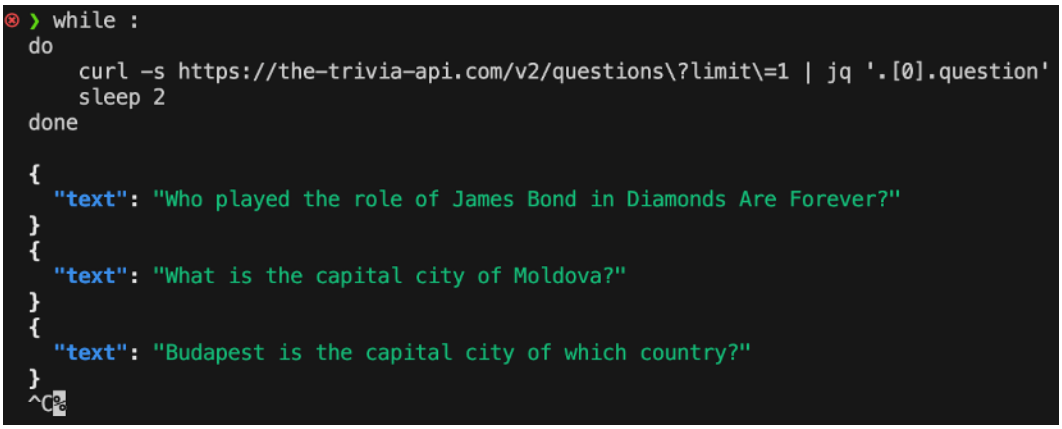


Figure 3.5 – Output random trivia

Each response is a different trivia question. You may need to install `jq` first on your Linux, macOS, or Windows computer. `jq` is a handy tool often used to nicely filter and format JSON output, which increases its readability onscreen. Use your package manager to install `jq` if needed. On Windows, using Chocolatey, the command would be as follows:

```
$ choco install jq
```

On a Mac using Homebrew, you would type the following:

```
$ brew install jq
```

Now, let's run this logic in an `alpine` container. Since this is not just a simple command, we want to wrap the preceding script in a script file and execute that one. To make things simpler, I have created a Docker image called

`fundamentalsofdocker/trivia` that contains all of the necessary logic so that we can just use it here. Later on, once we have introduced Docker images, we will analyze this container image further. For the moment, let's just use it as is. Execute the following command to run the container as a background service. In Linux, a background service is also called a daemon:

```
$ docker container run --detach \
    --name trivia fundamentalsofdocker/trivia:ed4
```

NOTE

Important note

We are using the `\` character to allow line breaks in a single logical command that does not fit on a single line. This is a feature of the shell script we use. In PowerShell, use the backtick (`) instead.

Also note that, on `zsh`, you may have to press *Shift* + *Enter* instead of only *Enter* after the `\` character to start a new line. Otherwise, you will get an error.

In the preceding expression, we have used two new command-line parameters, `--detach` and `--name`. Now, `--detach` tells Docker to run the process in the container as a Linux daemon.

The `--name` parameter, in turn, can be used to give the container an explicit name. In the preceding sample, the name we chose is `trivia`. If we don't specify an explicit container name when we run a container, then Docker will automatically assign the container a random but unique name. This name will be composed of the name of a famous scientist and an adjective. Such names could be `boring_borg` or `angry_goldberg`. They're quite humorous, the Docker engineers, aren't they?

Finally, the container we're running is derived from the `fundamentalsofdocker/trivia:ed4` image. Note how we are also using a tag, `ed4`, for the container. This tag just tells us that this image was originally created for the fourth edition of this book.

One important takeaway is that the container name has to be unique on the system. Let's make sure that the `trivia` container is up and running:

```
$ docker container ls -l
```

This should give us something like this:

```
❯ docker container ls -l
CONTAINER ID   IMAGE                                COMMAND          CREATED         STATUS         PORTS     NAMES
6ce8c46aba93   fundamentalsofdocker/trivia:ed4   "./trivia.sh"   9 seconds ago   Up 8 seconds             trivia
```

Figure 3.6 – Details of the last run container

An important part of the preceding output is the `STATUS` column, which in this case is `Up 8 seconds`. That is, the container has been up and running for 8 seconds now.

Don't worry if the previous Docker command is not yet familiar to you; we will come back to it in the next section.

To complete this section, let's stop and remove the `trivia` container with the following command:

```
$ docker rm --force trivia
```

The preceding command, while forcefully removing the `trivia` container from our system, will just output the name of the container, `trivia`, in the output.

Now it is time to learn how to list containers running or dangling on our system.

Listing containers

As we continue to run containers over time, we get a lot of them in our system. To prepare our system for the next command, let's run a few containers, as follows:

```
$ docker container run alpine echo "hello world"
$ docker container run --detach \
    quay.io/centos/centos:stream9 sleep 3600
$ docker container run --detach --name trivia fundamentalsofdocker/trivia:ed4
```

Now, to find out what is currently running on our host, we can use the `container ls` command, as follows:

```
$ docker container ls
```

This will list all currently running containers. Such a list might look similar to this:

```
❯ docker container ls
CONTAINER ID   IMAGE                                COMMAND          CREATED              STATUS              PORTS     NAMES
41ee1a094ccd   quay.io/centos/centos:stream9        "sleep 3600"     33 seconds ago       Up 32 seconds               optimistic_mcclintock
c6419e1070e3   fundamentalsofdocker/trivia:ed4   "./trivia.sh"   About a minute ago   Up About a minute           trivia
```

Figure 3.7 – List of all running containers on the system

Note, you can't see the `alpine` container in the preceding list. This is because the previous command only lists running containers, and the `alpine` container is in an `Exited` state. Later, we will learn how to also show stopped containers.

By default, Docker outputs seven columns with the following meanings:

Column	Description
Container ID	This is a short version of the unique ID of the container. It is an SHA-256, where SHA-256 (Secure Hash Algorithm 256-bit) is a widely used cryptographic hash function that takes an input and generates a fixed-size (256-bit) output, known as a hash. The full ID is 64 characters long.
Image	This is the name of the container image from which this container is instantiated.
Command	This is the command that is used to run the main process in the container.
Created	This is the date and time when the container was created.
Status	This is the status of the container (created, restarting, running, removing, paused, exited, or dead).
Ports	This is the list of container ports that have been mapped to the host.

Column	Description
Names	This is the name assigned to this container (note: multiple names for the same container are possible).

Table 3.1 – Description of the columns of the docker container ls command

If we want to list not just the currently running containers but all containers that are defined on our system, then we can use the `-a` or `--all` command-line parameter, as follows:

```
$ docker container ls --all
```

This will list containers in any state, such as Created, Running, or Exited.

Sometimes, we want to just list the IDs of all containers. For this, we have the `-q` or `--quiet` parameter:

```
$ docker container ls --quiet
```

You might wonder when this is useful. I will show you a command where it is very helpful right here:

```
$ docker container rm --force $(docker container ls --all --quiet)
```

Lean back and take a deep breath. Then, try to find out what the preceding command does. Don't read any further until you find the answer or give up.

Here is the solution: the preceding command forcefully deletes all containers that are currently defined on the system, including the stopped ones. The `rm` command stands for remove, and it will be explained soon.

There is also a `-l` parameter for the list command, that is, `docker container ls -l`. Try to use the `docker help` command to find out what the `-l` parameter stands for. You can invoke help for the list command as follows:

```
$ docker container ls --help
```

Now that you know how to list created, running, or stopped containers on your system, let's learn how to stop and restart containers.

Stopping and starting containers

Stopping and starting Docker containers are fundamental operations that allow us to manage the state of our applications effectively. Let's try this out with the `trivia` container we used previously:

1. Run the container again with this command:

```
$ docker container run -d --name trivia \
    fundamentalsofdocker/trivia:ed4
```

2. Now, if we want to stop this container, then we can do so by issuing this command:

```
$ docker container stop trivia
```

When you try to stop the `trivia` container, you will probably notice that it takes a while until this command is executed. To be precise, it takes about 10 seconds. *Why is this the case?*

Docker sends a Linux `SIGTERM` signal to the main process running inside the container. If the process doesn't react to this signal and terminate itself, Docker waits for 10 seconds and then sends `SIGKILL`, which will kill the process forcefully and terminate the container.

In the preceding command, we have used the name of the container to specify which container we want to stop. But we could have also used the container ID instead.

How do we get the ID of a container? There are several ways of doing so. The manual approach is to list all running containers and find the one that we're looking for in the list. From there, we copy its ID. A more automated way is to use some shell scripting and environment variables. If, for example, we want to get the ID of the `trivia` container, we can use this expression:

```
$ export CONTAINER_ID=$(docker container ls -a | \
    grep trivia | awk '{print $1}')
$ echo $CONTAINER_ID
```

The equivalent command in PowerShell would look like this:

```
$ $CONTAINER_ID = docker container ls -a | `
    Select-String "trivia" | `
```

```
    Select-Object -ExpandProperty Line | `
    ForEach-Object { $_ -split ' ' } | `
    Select-Object -First 1
$ Write-Output $CONTAINER_ID
```

Please note the back tricks (`) in PowerShell to denote a line break.

NOTE

Note

We are using the `-a` (or `--all`) parameter with the `docker container ls` command to list all containers, even the stopped ones. This is necessary in this case since we stopped the `trivia` container a moment ago.

Now, instead of using the container name, we can use the `$CONTAINER_ID` variable in our expression:

```
$ docker container stop $CONTAINER_ID
```

Once we have stopped the container, its status changes to `Exited`.

If a container is stopped, it can be started again using the `docker container start` command. Let's do this with our `trivia` container. It is good to have it running again, as we'll need it in the subsequent sections of this chapter:

```
$ docker container start $CONTAINER_ID
```

We can also start it by using the name of the container:

```
$ docker container start trivia
```

It is now time to discuss what to do with stopped containers that we don't need anymore.

Removing containers

When we run the `docker container ls -a` command, we can see quite a few containers that are in the `Exited` status. If we don't need these containers anymore, then it is a good thing to remove them from memory; otherwise, they unnecessarily occupy precious resources. The command to remove a container is as follows:

```
$ docker container rm <container ID>
```

Here, `<container ID>` stands for the ID of the container – a SHA-256 code – that we want to remove. Another way to remove a container is the following:

```
$ docker container rm <container name>
```

Here, we use the name of the container.

> NOTE
>
> **Challenge**
>
> Try to remove one of your exited containers using its ID.

Sometimes, removing a container will not work as it is still running. If we want to force a removal, no matter what the condition of the container currently is, we can use the `-f` or `--force` command-line parameter:

```
$ docker container rm <container ID> --force
```

Now that we have learned how to remove containers from our system, let's learn how to inspect containers present in the system.

Before you continue, make sure you have removed the `trivia` container with the following:

```
$ docker container rm -f trivia
```

Inspecting containers

Containers are runtime instances of an image and have a lot of associated data that characterizes their behavior. The `docker container inspect` command provides detailed, low-level information about a container in JSON format. It reveals everything from network settings and mount points to environment variables and the exact command used to start the container. This makes it a powerful tool for debugging and auditing, allowing you to understand how a container was configured and how it's currently behaving —without needing to access the container directly.

As usual, when executing the `inspect` command, we have to provide either the container ID or the name to identify the container for which we want to

obtain the data. So, let's inspect our sample container. First, we have to run it:

```
$ docker container run --detach --name trivia \
    fundamentalsofdocker/trivia:ed4
```

Then, use this command to inspect it:

```
$ docker container inspect trivia
```

The response is a big JSON object full of details. It looks similar to this:

Figure 3.8 – Inspecting the trivia container

Note that the preceding screenshot only shows the first part of a much longer output.

Please take a moment to analyze what you have. You should see information such as the following:

- The ID of the container
- The creation date and time of the container
- From which image the container is built

Many sections of the output, such as `Mounts` and `NetworkSettings`, don't make much sense right now, but we will discuss those in the upcoming chapters of this book. The data you're seeing here is also named the *metadata* of a container. We will be using the `inspect` command quite often in the remainder of this book as a source of information.

Sometimes, we need just a tiny bit of the overall information, and to achieve this, we can use either the grep tool or a filter. The former method does not always result in the expected answer, so let's look into the latter approach:

```
$ docker container inspect -f "{{json .State}}" trivia \
    | jq .
```

The `-f` or `--filter` parameter is used to define the `"{{json .State}}"` filter. The filter expression itself uses the Go template syntax. In this example, we only want to see the state part of the whole output in JSON format. To nicely format the output, we pipe the result into the `jq` tool:

```
❯ docker container inspect -f "{{json .State}}" trivia \
    | jq .

{
  "Status": "running",
  "Running": true,
  "Paused": false,
  "Restarting": false,
  "OOMKilled": false,
  "Dead": false,
  "Pid": 2125,
  "ExitCode": 0,
  "Error": "",
  "StartedAt": "2025-03-30T16:18:57.468353047Z",
  "FinishedAt": "0001-01-01T00:00:00Z"
}
```

Figure 3.9 – The state node of the inspect output

After we have learned how to retrieve loads of important and useful meta information about a container, we want to investigate how we can execute it in a running container.

Exec into a running container

The `docker container exec` command lets us run a new command inside an already running container without interrupting its main process. It's ideal for inspecting the container's state, troubleshooting problems, or performing administrative tasks—such as checking logs, testing connectivity, or restarting services. Unlike `docker container attach`, which we will describe in the next section, it doesn't connect us to the container's primary process but

starts a separate one, making it a safer and more flexible option for real-time diagnostics and maintenance.

How can we do this? First, we need to know either the ID or the name of the container, and then we can define which process we want to run and how we want it to run. Once again, we use our currently running `trivia` container, and we run a shell interactively inside it with the following command:

```
$ docker container exec -i -t trivia /bin/sh
```

The output on the screen will be as follows:

```
/app #
```

The `-i` (or `--interactive`) flag in the preceding command signifies that we want to run the additional process interactively, and `-t` (or `--tty`) tells Docker that we want it to provide us with a TTY (a terminal emulator) for the command. Finally, the process we run inside the container is `/bin/sh`.

If we execute the preceding command in our terminal, then we will be presented with a new prompt, `/app #`. We're now in a Bourne shell inside the `trivia` container. We can easily prove that by, for example, executing the `ps` command, which will list all running processes in the context:

```
/app # ps
```

The result should look somewhat similar to this:

```
❯ docker container exec -i -t trivia /bin/sh
/app # ps
PID   USER     TIME  COMMAND
    1 root      0:00 {trivia.sh} /bin/sh ./trivia.sh
  237 root      0:00 /bin/sh
  350 root      0:00 sleep 2
  351 root      0:00 ps
/app #
```

Figure 3.10 – Executing into the running trivia container

We can clearly see that the process with PID `1` is the command that we have defined to run inside the `trivia` container. The process with PID `1` is also named the main process.

Exit the container by pressing *Ctrl* + *D*.

We not only execute additional processes interactively in a container but also execute them in an automated way. Please consider the following command:

```
$ docker container exec trivia ps
```

The output evidently looks very similar to the preceding output:

```
❯ docker container exec trivia ps
PID   USER     TIME  COMMAND
    1 root      0:00 {trivia.sh} /bin/sh ./trivia.sh
  480 root      0:00 sleep 2
  481 root      0:00 ps
```

Figure 3.11 – List of processes running inside the trivia container

The difference is that we did not use an extra process to run a shell, but executed the `ps` command directly. We can even run processes as a daemon using the `-d` flag and define environment variables valid inside the container, using the `-e` or `--env` flag variables, as follows:

1. Run the following command to start a shell inside a `trivia` container and define an environment variable named `MY_VAR` that is valid inside this container:

```
$ docker container exec -it \
    -e MY_VAR="Hello World" \
    trivia /bin/sh
```

2. You'll find yourself inside the `trivia` container. Output the content of the `MY_VAR` environment variable, as follows:

```
/app # echo $MY_VAR
```

3. You should see the **Hello World** output in the terminal, as follows:

```
❯ docker container exec -it \
    -e MY_VAR="Hello World" \
    trivia /bin/sh

/app # echo $MY_VAR
Hello World
/app #
```

Figure 3.12 – Running a trivia container and defining an environment variable

4. To exit the `trivia` container, press *Ctrl* + *D*:

```
/app # <CTRL-d>
```

Before you continue to the next section, make sure to remove the `trivia` container.

```
$ docker container rm --force trivia
```

Great, we have learned how to execute into a running container and run additional processes. But there is another important way to work with a running container.

Attaching to a running container

Attaching to a running Docker container allows us to interact directly with the process inside it, which is especially useful for debugging, monitoring output, or manually executing commands in an interactive shell. It gives us a live view of the container's standard input, output, and error streams—essentially placing us *inside* the container as if we were running the application locally. This can be invaluable when diagnosing issues or exploring container behavior in real time.

We can use the `attach` command to attach our terminal's standard input, output, and error (or any combination of the three) to a running container using the ID or name of the container. Let's do this for our `trivia` container:

1. Open a new terminal window.

NOTE

Tip

You may want to use a terminal other than the integrated terminal of VS Code for this exercise, as it seems to cause problems with the key combinations that we are going to use. On Mac, use the Terminal app, as an example.

1. Run a new instance of the `trivia` Docker image in interactive mode:

```
$ docker container run -it \
    --name trivia fundamentalsofdocker/trivia:ed4
```

2. Open yet another terminal window and use this command to attach it to the container:

```
$ docker container attach trivia
```

   In this case, we will see, every two seconds or so, a new quote appearing in the output.

3. To quit the container without stopping or killing it, we can use the *Ctrl* + *P* and *Ctrl* + *Q* key combination. This detaches us from the container while leaving it running in the background.
4. Back in the first terminal window, hit *Ctrl* + *C* to stop the `trivia` container.
5. Stop and remove the container forcefully:

```
$ docker container rm --force trivia
```

NOTE

Tip

If you are using the *Ctrl* + *P* and *Ctrl* + *Q* key combination in a terminal of VS Code, it won't work as the key combination is intercepted by VS Code. Use a standalone terminal instead.

Let's run another container – this time, an Nginx web server:

1. Run the Nginx web server as follows:

```
$ docker run -d --name nginx -p 8080:80 nginx:alpine
```

NOTE

Tip

Here, we run the Alpine version of Nginx as a daemon in a container named `nginx`. The `-p 8080:80` command-line parameter opens port `8080` on the host (that is, the user's machine) for access to the Nginx web server running inside the container. Don't worry about the syntax here, as we will explain this feature in more detail in *Chapter 10*, *Single-Host Networking*.

On Windows, you'll need to approve a prompt that Windows Firewall will pop up. You have to allow Docker Desktop on the firewall.

1. Let's see whether we can access Nginx using the `curl` tool by running this command:

```
$ curl -4 localhost:8080
```

   If all works correctly, you should be greeted by the welcome page of Nginx:

```
❯ curl -4 localhost:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

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

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

Figure 3.13 – Welcome message of the Nginx web server

2. Now, let's attach our terminal to the Nginx container to observe what's happening:

```
$ docker container attach nginx
```

3. Once you are attached to the container, you will not see anything at first. But now, open another terminal, and in this new terminal window, repeat the `curl` command a few times, for example, using the following script:

```
$ for n in {1..10} do; curl -4 localhost:8080 done;
```

   Or, in PowerShell, use the following:

```
PS> for ($n = 1; $n -le 10; $n++) {
    curl -4 http://localhost:8080
```

```
}
```

You should see the logging output of Nginx, which looks similar to this:

```
docker container attach nginx
172.17.0.1 - - [30/Mar/2025:16:38:42 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/8.7.1" "-"
172.17.0.1 - - [30/Mar/2025:16:38:47 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/8.7.1" "-"
172.17.0.1 - - [30/Mar/2025:16:39:27 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/8.7.1" "-"
172.17.0.1 - - [30/Mar/2025:16:39:27 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/8.7.1" "-"
172.17.0.1 - - [30/Mar/2025:16:39:27 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/8.7.1" "-"
172.17.0.1 - - [30/Mar/2025:16:39:27 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/8.7.1" "-"
172.17.0.1 - - [30/Mar/2025:16:39:27 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/8.7.1" "-"
172.17.0.1 - - [30/Mar/2025:16:39:27 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/8.7.1" "-"
172.17.0.1 - - [30/Mar/2025:16:39:27 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/8.7.1" "-"
172.17.0.1 - - [30/Mar/2025:16:39:27 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/8.7.1" "-"
172.17.0.1 - - [30/Mar/2025:16:39:28 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/8.7.1" "-"
172.17.0.1 - - [30/Mar/2025:16:39:28 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/8.7.1" "-"
172.17.0.1 - - [30/Mar/2025:16:39:28 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/8.7.1" "-"
```

Figure 3.14 – Output of Nginx

4. Quit the container by pressing *Ctrl* + *C*. This will detach your terminal and, at the same time, stop the Nginx container.
5. To clean up, remove the Nginx container with the following command:

```
$ docker container rm nginx
```

In the next section, we're going to learn how to work with container logs.

Retrieving container logs

It is a best practice for any good application to generate some logging information that developers and operators alike can use to find out what the application is doing at a given time, and whether there are any problems to help to pinpoint the root cause of the issue.

When running inside a container, the application should preferably output the log items to `STDOUT` and `STDERR` and not into a file. If the logging output is directed to `STDOUT` and `STDERR`, then Docker can collect this information and keep it ready for consumption by a user or any other external system:

1. Run a `trivia` container in detach mode:

```
$ docker container run --detach \
    --name trivia fundamentalsofdocker/trivia:ed4
```

Let it run for a minute or so to give it time to generate a few trivia questions.

2. To access the logs of a given container, we can use the `docker container logs` command. If, for example, we want to retrieve the logs of our `trivia` container, we can use the following expression:

```
$ docker container logs trivia
```

   This will retrieve the whole log produced by the application from the very beginning of its existence.

NOTE

Note

Stop, wait a second – this is not quite true, what I just said. By default, Docker uses the so-called `json-file` logging driver. This driver stores logging information in a file. If there is a file rolling policy defined, then `docker container logs` only retrieves what is in the currently active log file and not what is in previous rolled files that might still be available on the host.

1. If we want to only get a few of the latest entries, we can use the `-t` or `--tail` parameter, as follows:

```
$ docker container logs --tail 5 trivia
```

   This will retrieve only the last five lines of the log that the process running inside the container produced.

2. Sometimes, we want to follow the log that is produced by a container. This is possible when using the `-f` or `--follow` parameter. The following expression will output the last five log items and then follow the log as it is produced by the containerized process:

```
$ docker container logs --tail 5 --follow trivia
```

3. Press *Ctrl* + *C* to stop following the logs.
4. Clean up your environment and remove the `trivia` container with the following:

```
$ docker container rm --force trivia
```

Often, using the default mechanism for container logging is not enough. We need a different way of logging. This is discussed in the following section.

Logging drivers

Docker includes multiple logging mechanisms to help us to get information from running containers. These mechanisms are named logging drivers. Which logging driver is used can be configured at the Docker daemon level. The default logging driver is `json-file`. Some of the drivers that are currently supported natively are as follows:

Driver	Description
`none`	No log output for the specific container is produced.
`json-file`	This is the default driver. The logging information is stored in files, formatted as JSON.
`journald`	If the `journald` daemon is running on the host machine, we can use this driver. It forwards logging to the `journald` daemon.
`syslog`	If the `syslog` daemon is running on the host machine, we can configure this driver, which will forward the log messages to the `syslog` daemon.
`gelf`	When using this driver, log messages are written to a **Graylog Extended Log Format** (**GELF**) endpoint. Popular examples of such endpoints are Graylog and Logstash.

Driver	Description
`fluentd`	Assuming that the `fluentd` daemon is installed on the host system, this driver writes log messages to it.
`awslogs`	The `awslogs` logging driver for Docker is a logging driver that allows Docker to send log data to Amazon CloudWatch Logs.
`splunk`	The Splunk logging driver for Docker allows Docker to send log data to Splunk, a popular platform for log management and analysis.

Table 3.2 – List of logging drivers

NOTE

Note

If you change the logging driver, please be aware that the `docker container logs` command is only available for the `json-file` and `journald` drivers. Docker 20.10 and up introduce *dual logging*, which uses a local buffer that allows you to use the `docker container logs` command for any logging driver.

Using a container-specific logging driver

The logging driver can be set globally in the Docker daemon configuration file. But we can also define the logging driver on a container-by-container basis. In the following example, we are running a `busybox` container and use the `--logdriver` parameter to configure the `none` logging driver:

1. Run an instance of the `busybox` Docker image and execute a simple script in it, outputting a hello message three times:

```
$ docker container run --name test -it \
    --log-driver none \
```

```
    busybox sh -c \
      'for N in 1 2 3; do echo "Hello $N"; done'
```

We should see the following:

```
Hello 1
Hello 2
Hello 3
```

2. Now, let's try to get the logs of the preceding container:

```
$ docker container logs test
```

The output is as follows:

```
Error response from daemon: configured logging driver does not support reading
```

This is to be expected since the `none` driver does not produce any logging output.

3. Let's clean up and remove the test container:

```
$ docker container rm test
```

To end this section about logging, we want to discuss a somewhat advanced topic, namely, how to change the default logging driver.

Advanced topic – changing the default logging driver

Let's change the default logging driver of a Linux host. The easiest way to do this is on a real Linux host. For this purpose, we're going to use Vagrant with an Ubuntu image. Vagrant is an open source tool developed by HashiCorp that is often used to build and maintain portable virtual software development environments. Please follow these instructions:

1. Open a new terminal window.
2. If you haven't done so before, on your Mac or Windows machine, you may need to install a hypervisor such as VirtualBox first. If you're using a Pro version of Windows, you can also use Hyper-V instead:
    - To install VirtualBox on a Mac with an Intel CPU, use Homebrew as follows:

```
$ brew install --cask virtualbox
```

- On Windows, with Chocolatey, use the following:

```
$ choco install -y virtualbox
```

NOTE

Note

On a Mac with an M1/M2 CPU, at the time of writing this, you need to install the developer preview of VirtualBox. Please follow the instructions here: https://www.virtualbox.org/wiki/Downloads.

3. Install Vagrant on your computer using your package manager, such as Chocolatey on Windows or Homebrew on Mac. On the author's MacBook Pro M2, the command looks like this:

```
$ brew install --cask vagrant
```

On a Windows machine, the corresponding command would be the following:

```
$ choco install -y vagrant
```

4. Once successfully installed, make sure Vagrant is available with the following command:

```
$ vagrant --version
```

At the time of writing this, Vagrant replies with the following:

```
Vagrant 2.4.3
```

5. In your terminal, execute the following command to initialize an Ubuntu 22.04 VM with Vagrant:

```
$ vagrant init bento/ubuntu-24.04
```

Here is the generated output:

```
❯ vagrant init bento/ubuntu-24.04
A `Vagrantfile` has been placed in this directory. You are now
ready to `vagrant up` your first virtual environment! Please read
the comments in the Vagrantfile as well as documentation on
`vagrantup.com` for more information on using Vagrant.
```

Figure 3.15 – Initializing a Vagrant VM based on Ubuntu 22.04

Vagrant will create a file called `Vagrantfile` in the current folder. Optionally, you can use your editor to analyze the content of this file.

6. Now, start this VM using Vagrant:

```
$ vagrant up
```

7. Connect from your laptop to the VM using a secure shell (`ssh`):

```
$ vagrant ssh
```

After this, you will find yourself inside the VM and can start working with Docker inside this VM.

```
❯ vagrant ssh
Welcome to Ubuntu 24.04.2 LTS (GNU/Linux 6.8.0-53-generic aarch64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

 System information as of Sun Mar 30 05:01:56 PM UTC 2025

  System load:           0.0
  Usage of /:            16.9% of 29.82GB
  Memory usage:          5%
  Swap usage:            0%
  Processes:             129
  Users logged in:       0
  IPv4 address for eth0: 10.0.2.15
  IPv6 address for eth0: fd00::a00:27ff:fe71:19d8


This system is built by the Bento project by Chef Software
More information can be found at https://github.com/chef/bento

Use of this system is acceptance of the OS vendor EULA and License Agreements.
vagrant@vagrant:~$
```

Figure 3.16 – Inside the Vagrant Ubuntu 24.04 box

8. Once inside the Ubuntu VM, install Docker using the following steps:
    a. Update the package list:

```
$ sudo apt-get update
```

b. Install the required dependencies:

```
$ sudo apt-get update
$ sudo apt-get install -y ca-certificates \
    curl gnupg lsb-release
```

c. Add Docker's official GPG key:

```
$ sudo mkdir -p /etc/apt/keyrings
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/etc/apt/keyrings/docker.gpg
```

d. Add a Docker repository:

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

e. Update the package lists again:

```
$ sudo apt update
```

f. Install Docker Engine and components:

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

g. Verify the Docker installation:

```
$ sudo docker run hello-world
```

h. Add user to the Docker group:

```
$ sudo usermod -aG docker $USER
```

9. Log out and log in again to apply the changes.
10. Now, we want to edit the Docker daemon configuration file and trigger the Docker daemon to reload the configuration file thereafter:

a. Navigate to the `/etc/docker` folder:

```
$ cd /etc/docker
```

b. Run `vi` as follows:

```
$ sudo vi daemon.json
```

c. Enter the following content:

```
{
    "Log-driver": "json-log",
    "log-opts": {
        "max-size": "10m",
        "max-file": 3
    }
}
```

d. The preceding definition tells the Docker daemon to use the `json-log` driver with a maximum log file size of 10 MB before it is rolled, and the maximum number of log files that can be present on the system is three before the oldest file gets purged.

e. Save and exit `vi` by first pressing *Esc*, then typing `:w:q` (which means *write and quit*), and finally hitting the *Enter* key.

f. Now, we must send a `SIGHUP` signal to the Docker daemon so that it picks up the changes in the configuration file:

```
$ sudo kill -SIGHUP $(pidof dockerd)
```

g. Note that the preceding command only reloads the config file and does not restart the daemon.

11. Test your configuration by running a few containers and analyzing the log output.
12. Clean up your system once you are done experimenting with the following:

```
$ vagrant box list
$ vagrant destroy [name|id]
```

Great! The previous section was an advanced topic and showed how you can change the log driver on a system level. Let's now talk a bit about the anatomy of containers.

The anatomy of containers

Many people wrongly compare containers to VMs. However, this is a questionable comparison. Containers are not just lightweight VMs. OK then,

what is the correct description of a container?

Containers are specially encapsulated and secured processes running on the host system. Containers leverage a lot of features and primitives available in the Linux operating system. The most important ones are **namespaces** and **control groups** (**cgroups** for short). All processes running in containers only share the same Linux kernel of the underlying host operating system. This is fundamentally different compared with VMs, as each VM contains its own full-blown operating system.

The startup times of a typical container can be measured in milliseconds, while a VM normally needs several seconds to minutes to start up. VMs are meant to be long-living. It is a primary goal of each operations engineer to maximize the uptime of their VMs. Contrary to that, containers are meant to be ephemeral. They come and go relatively quickly.

Let's first get a high-level overview of the architecture that enables us to run containers.

Architecture

Here, we have an architectural diagram of Docker and how this all fits together:

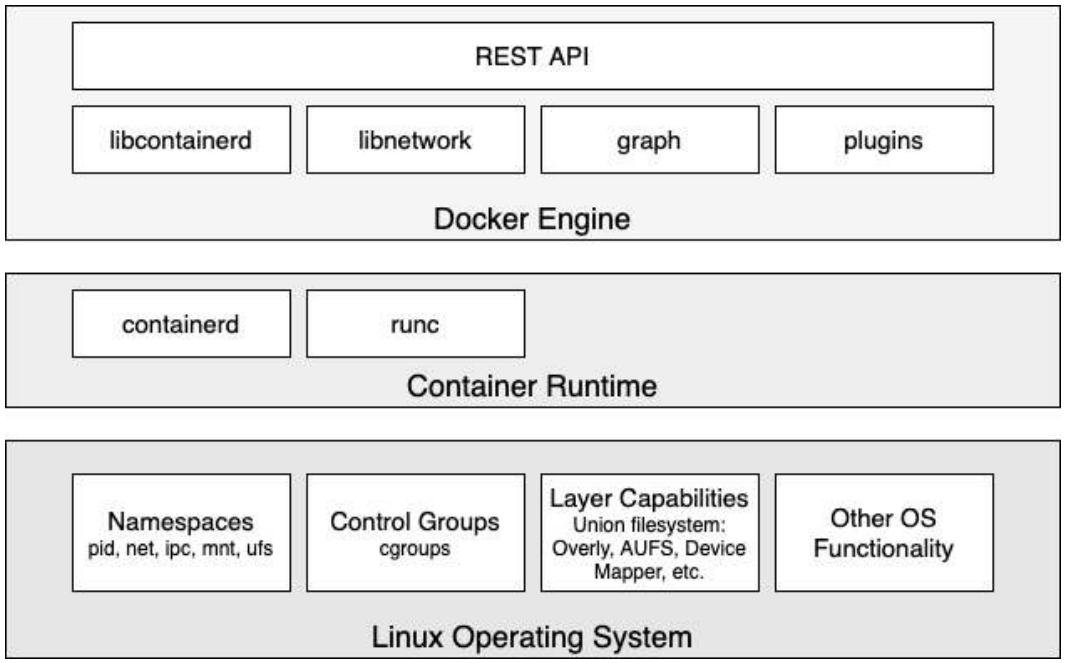


Figure 3.16 – High-level architecture of Docker

In the lower part of the preceding diagram, we have the Linux operating system with its cgroups, namespaces, and layer capabilities, as well as other operating system functionality that we do not need to explicitly mention here. Then, there is an intermediary layer composed of `containerd` and `runc`. On top of all that sits Docker Engine. Docker Engine offers a RESTful interface to the outside world that can be accessed by any tool, such as the Docker CLI, Docker Desktop, or Kubernetes, to name just a few.

Let's now describe the main building blocks in a bit more detail.

Namespaces

Linux namespaces were around for years before they were leveraged by Docker for its containers. A **namespace** is an abstraction of global resources such as filesystems, network access, and process trees (also named PID namespaces) or the system group IDs and user IDs. A Linux system is initialized with a single instance of each namespace type. After initialization, additional namespaces can be created or joined.

The Linux namespaces originated in 2002 in the 2.4.19 kernel. In kernel version 3.8, user namespaces were introduced, and with this, namespaces were ready to be used by containers.

If we wrap a running process, say, in a filesystem namespace, then this provides the illusion that the process owns its own complete filesystem. This, of course, is not true; it is only a virtual filesystem. From the perspective of the host, the contained process gets a shielded subsection of the overall filesystem. It is like a filesystem in a filesystem:

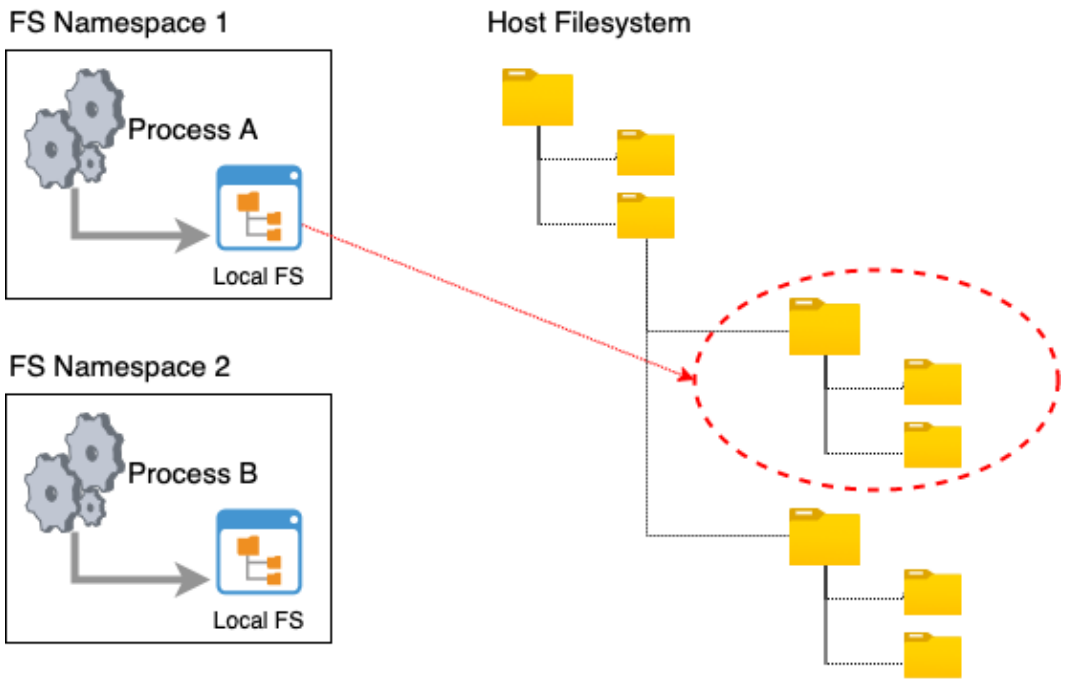


Figure 3.17 – Namespaces explained

The same applies to all of the other global resources for which namespaces exist. The user ID namespace is another example. Now that we have a user namespace, we can define a `jdoe` user many times on the system as long as it is living in its own namespace.

The PID namespace is what keeps processes in one container from seeing or interacting with processes in another container. A process might have the apparent PID 1 inside a container, but if we examine it from the host system, it will have an ordinary PID, say, 334:

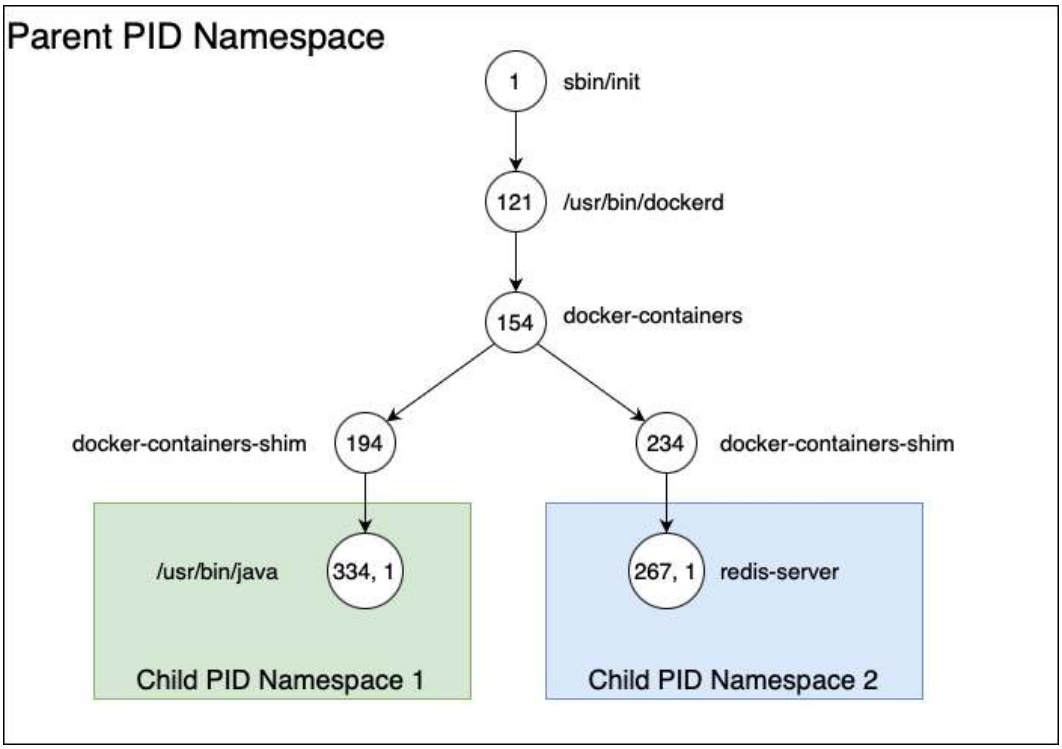


Figure 3.18 – Process tree on a Docker host

In each namespace, we can run one-to-many processes. That is important when we talk about containers, which we already experienced when we executed another process in an already-running container.

Control groups

Linux cgroups are used to limit, manage, and isolate the resource usage of collections of processes running on a system. Resources are CPU time, system memory, network bandwidth, or combinations of these resources.

Engineers at Google originally implemented this feature in 2006. The cgroups functionality was merged into the Linux kernel mainline in kernel version 2.6.24, which was released in January 2008.

Using cgroups, administrators can limit the resources that containers can consume. With this, we can avoid, for example, the classic noisy neighbor problem, where a rogue process running in a container consumes all CPU time or reserves massive amounts of RAM and, as such, starves all the other processes running on the host, whether they're containerized or not.

Union filesystem

Union filesystem (**unionfs**) forms the backbone of what is known as container images. We will discuss container images in detail in the next chapter. Currently, we want to just understand what unionfs is and how it works a bit better. unionfs is mainly used on Linux and allows files and directories of distinct filesystems to be overlaid to form a single coherent filesystem. In this context, the individual filesystems are called branches. Contents of directories that have the same path within the merged branches will be seen together in a single merged directory, within the new virtual filesystem. When merging branches, the priority between the branches is specified. In that way, when two branches contain the same file, the one with the higher priority is seen in the final filesystem.

Container plumbing

The foundation on which Docker Engine is built is formed of two components, `runc` and `containerd`.

Originally, Docker was built in a monolithic way and contained all of the functionality necessary to run containers. Over time, this became too rigid, and Docker started to break out parts of the functionality into their own components. Let's explain in more detail what `runc` and `containerd` are.

runc

runc is a lightweight, portable container runtime. It provides full support for Linux namespaces, as well as native support for all security features available on Linux, such as SELinux, AppArmor, seccomp, and cgroups.

runC is a tool for spawning and running containers according to the **Open Container Initiative** (**OCI**) specification. It is a formally specified configuration format governed by the **Open Container Project** (**OCP**) under the auspices of the Linux Foundation.

Containerd

runC is a low-level implementation of a container runtime; containerd builds on top of it and adds higher-level features, such as image transfer and storage, container execution, and supervision, as well as network and storage attachments. With this, it manages the complete life cycle of containers. Containerd is the reference implementation of the OCI specifications and is by far the most popular and widely used container runtime.

Containerd was donated to and accepted by the CNCF in 2017. There are alternative implementations of the OCI specification. Some of them are rkt by CoreOS, CRI-O by Red Hat, and LXD by Linux Containers. However, containerd is currently by far the most popular container runtime and is the default runtime of Kubernetes 1.8 or later and the Docker platform.

This concludes our introduction to the anatomy of containers. Let's recap the chapter.

Summary

In this chapter, you learned how to work with containers that are based on existing images. We showed how to run, stop, start, and remove a container. Then, we inspected the metadata of a container, extracted its logs, and learned how to run an arbitrary process in an already-running container. Last but not least, we dug a bit deeper and investigated how containers work and what features of the underlying Linux operating system they leverage.

In the next chapter, you're going to learn what container images are and how we can build and share our own custom images. We'll also be discussing the best practices commonly used when building custom images, such as minimizing their size and leveraging the image cache. Stay tuned!

Further reading

The following articles give you some more information related to the topics we discussed in this chapter:

- Get started with containers at `https://docs.docker.com/get-started/`
- Get an overview of Docker container commands at `http://dockr.ly/2iLBV2I`
- Learn about isolating containers with a user namespace at `http://dockr.ly/2gmyKdf`
- Learn about limiting a container's resources at `http://dockr.ly/2wqN5Nn`

Questions

To assess your learning progress, please answer the following questions:

1. Which two core Linux features enable containerization by providing process isolation and resource management?
2. What are the possible states of a Docker container?
3. Which command is used to display all currently running containers on your Docker host?
4. How can you list only the container IDs of all Docker containers?
5. What is the difference between `docker container exec` and `docker container attach`?
6. How do you run a Docker container in detached mode, and why would you choose to do so?

Answers

Here are sample answers to the questions presented in this chapter:

1. Linux namespaces and control groups (cgroups) are the two essential features. Namespaces create isolated environments for processes by giving each container its own view of the system (for example, process trees, network interfaces, file systems), while cgroups manage and limit the resources (CPU, memory, I/O, and so on) that processes within each container can consume.
2. The possible states of a Docker container are as follows:
    - **Created**: The container that has been created but not yet started
    - **Restarting**: The container is in the process of being restarted
    - **Running**: The container is actively executing its main process
    - **Paused**: All processes within the container have been temporarily suspended
    - **Exited**: The container has finished running and its main process has stopped
    - **Dead**: Docker attempted to stop the container, but it could not be terminated properly
3. We can use the following (or the old, shorter version, `docker ps`):

```
$ docker container ls
```

This is used to list all containers that are currently running on our Docker host. Note that this will *not* list the stopped containers, for which you need the extra `--all` (or `-a`) parameter.

4. To list all IDs of containers, running or stopped, we can use the following:

```
$ docker container ls -a -q
```

   Here, `-q` stands for output ID only, and `-a` tells Docker that we want to see all containers, including stopped ones.

5. The difference between `docker container exec` and `docker container attach` is as follows:
   - `docker container exec`: This command starts a **new process** inside an already running container. For example, you can launch an interactive shell (using `/bin/sh` or `/bin/bash`) without affecting the container's main process.
   - `docker container attach`: This command connects your terminal directly to the **main process** of the container, attaching to its standard input, output, and error streams. This is useful for viewing real-time logs or interacting with the primary application running in the container.
6. To run a container in the background, use the `--detach` (or `-d`) flag with the `docker container run` command. Here's an example:

```
docker container run -d --name my_container my_image
```

   Running in detached mode is useful when you want the container to operate as a background service—such as a web server or database—without tying up your terminal session.

4

Creating and Managing Container Images

Join our book community on Discord:

https://packt.link/mqfS2

In the previous chapter, we learned what containers are and how to run, stop, remove, list, and inspect them. We extracted the logging information of some containers, ran other processes inside an already running container, and finally, we dove deep into the anatomy of containers. Whenever we ran a container, we created it using a container image. In this chapter, we will familiarize ourselves with these container images. We will learn what they are, how to create them, and how to distribute them.

This chapter will cover the following topics:

- What are Docker images?
- Creating Docker images
- Lift and shift: containerizing a legacy application
- Sharing or shipping images
- Supply chain security practices

After completing this chapter, you will be able to do the following:

- Build custom Docker images using Dockerfiles, applying best practices for efficiency and security
- Create a custom image by interactively changing the container layer and committing it
- Author a simple Dockerfile using keywords such as `FROM`, `COPY`, `RUN`, `CMD`, and `ENTRYPOINT` to generate a custom image
- Export an existing image using `docker image save` and import it into another Docker host

- Write a multi-step Dockerfile that minimizes the size of the resulting image by only including the resulting binaries in the final image
- Create a Dockerfile for an existing legacy application
- Utilize Docker registries to store, share, and version-control images, demonstrating this by pushing and pulling images from a registry

What are images?

In Linux, everything is a file. The whole operating system is a filesystem with files and folders stored on the local disk. This is an important fact to remember when looking at what container images are. As we will see, an image is a big tarball containing a filesystem. More specifically, it contains a layered filesystem.

NOTE

tarball

A tarball (also known as a `.tar` archive) is a single file that contains multiple files or directories. It is a common archive format that is used to distribute software packages and other collections of files. The `.tar` archive is usually compressed using gzip or another compression format to reduce its size. Tarballs are commonly used in Unix-like operating systems, including Linux and macOS, and can be unpacked using the `tar` command.

The layered filesystem

Container images are templates from which containers are created. These images are not made up of just one monolithic block but are composed of many layers. The first layer in the image is also called the **base layer**. We can see this in the following figure:

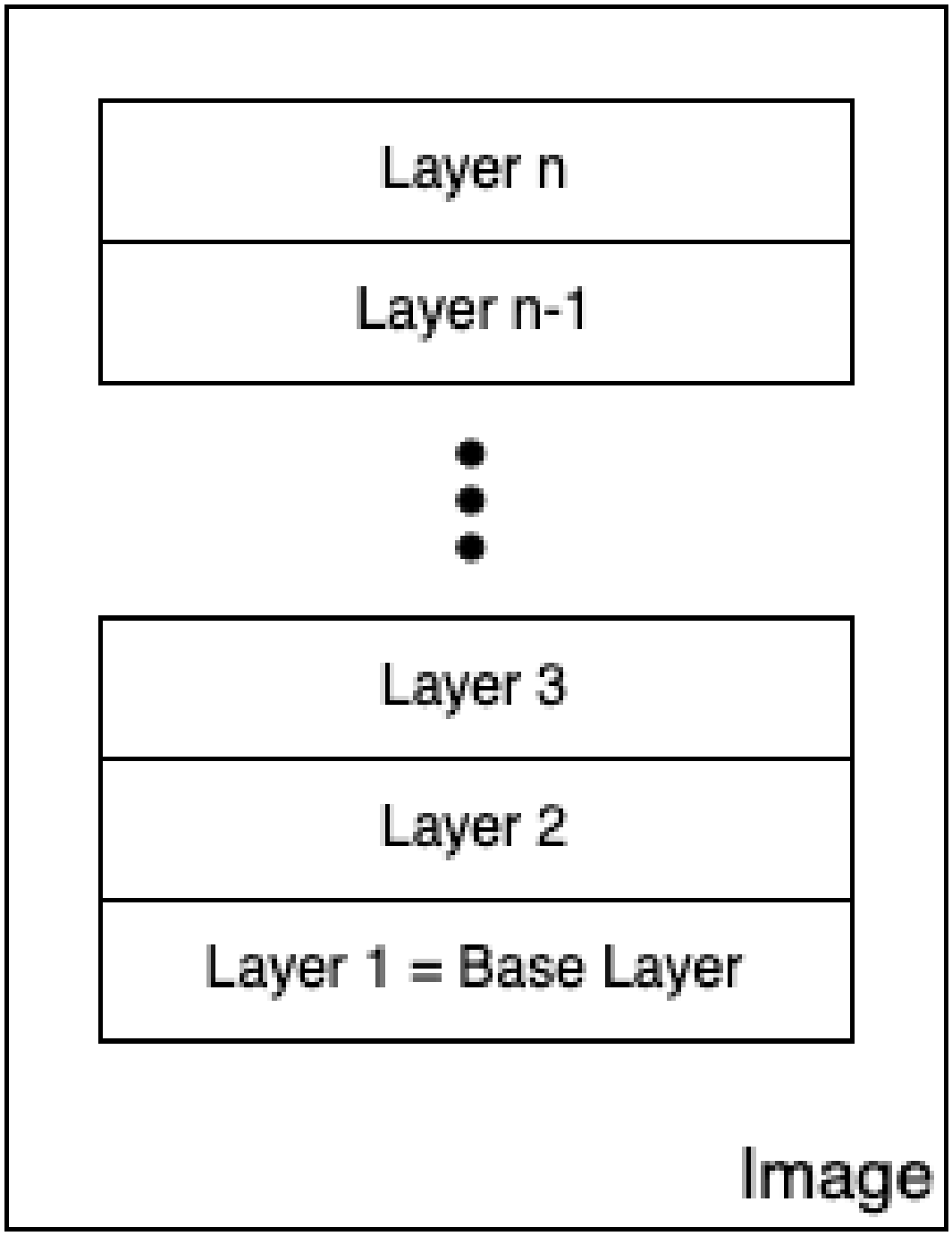


Figure 4.1: The image as a stack of layers

Each layer contains files and folders. Each layer only contains the changes to the filesystem concerning the underlying layers. Docker uses a Union filesystem – as discussed in *Chapter 3*, *Mastering Containers* – to create a virtual filesystem out of the set of layers. A storage driver handles the details regarding the way these layers interact with each other. Various storage

drivers are available that each have advantages and disadvantages in different situations.

The layers of a container image are all immutable. Immutable means that, once generated, the layer cannot ever be changed. The only possible operation affecting the layer is its physical deletion. This immutability of layers is important because it opens up a tremendous number of opportunities, as we will see later in this chapter, more precisely in the *Dockerfile best practices* section.

In the following figure, we can see what a custom image for a web application, using Nginx as a web server, could look like:

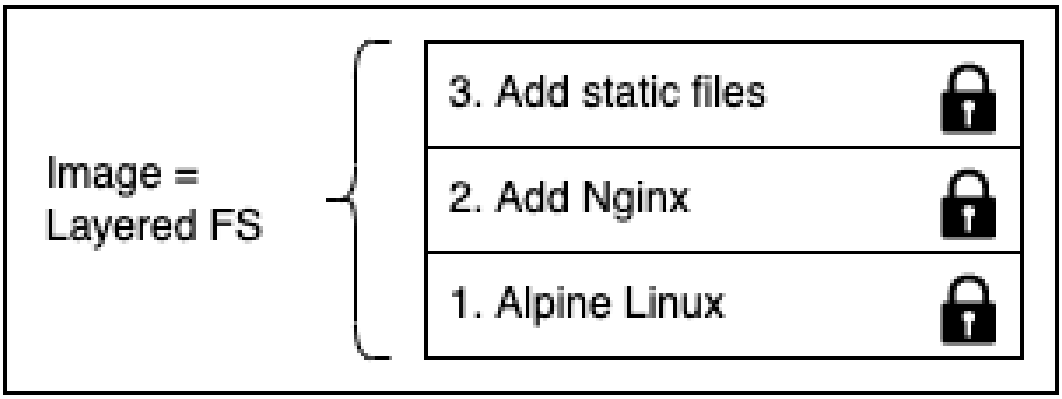


Figure 4.2: A sample custom image based on Alpine and Nginx

Our base layer here consists of the Alpine Linux distribution. Then, on top of that, we have an **Add Nginx** layer where Nginx is added on top of Alpine. Finally, the third layer contains all the files that make up the web application, such as HTML, CSS, and JavaScript files.

As has been said previously, each image starts with a base image. Typically, this base image is one of the official images found on Docker Hub, such as a Linux distro, such as Alpine, Ubuntu, or CentOS. However, it is also possible to create an image from scratch.

NOTE

Note

Docker Hub is a public registry for container images. It is a central hub ideally suited for sharing public container images. The registry can be found here: `https://hub.docker.com/`.

Each layer only contains the delta of changes regarding the previous set of layers. The content of each layer is mapped to a special folder on the host system, which is usually a subfolder of `/var/lib/docker/`.

Since layers are immutable, they can be cached without ever becoming stale. This is a big advantage, as we will see in the *Dockerfile best practices* section.

The writable container layer

As we have discussed, a container image is made of a stack of immutable or read-only layers. When Docker Engine creates a container from such an image, it adds a writable container layer on top of this stack of immutable layers. Our stack now looks as follows:

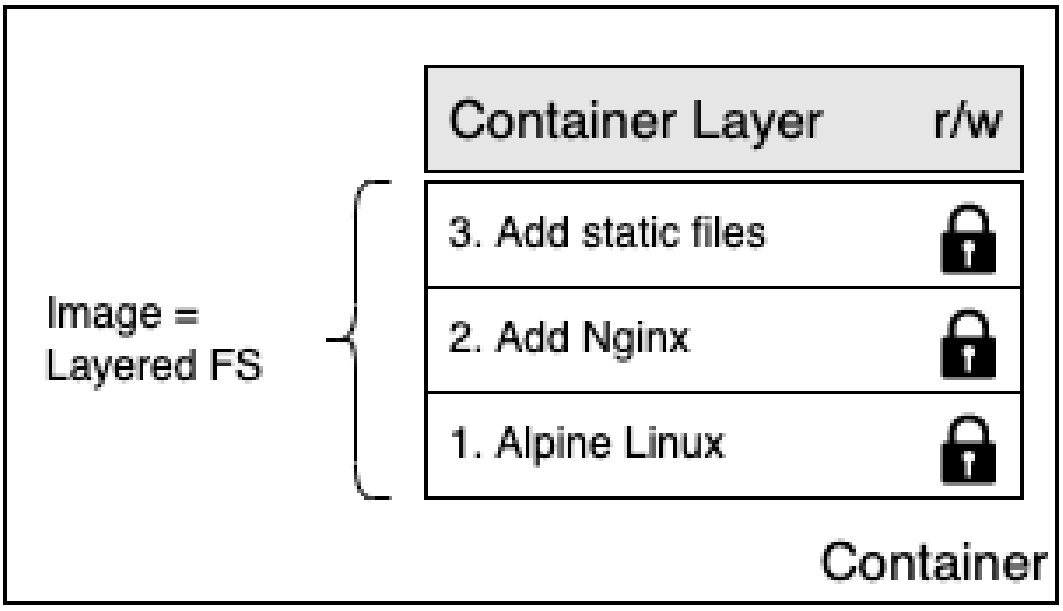


Figure 4.3: The writable container layer

The container layer is marked as **read/write** (**r/w**). Another advantage of the immutability of image layers is that they can be shared among many containers created from this image. All that is needed is a thin, writable container layer for each container, as shown in the following figure:

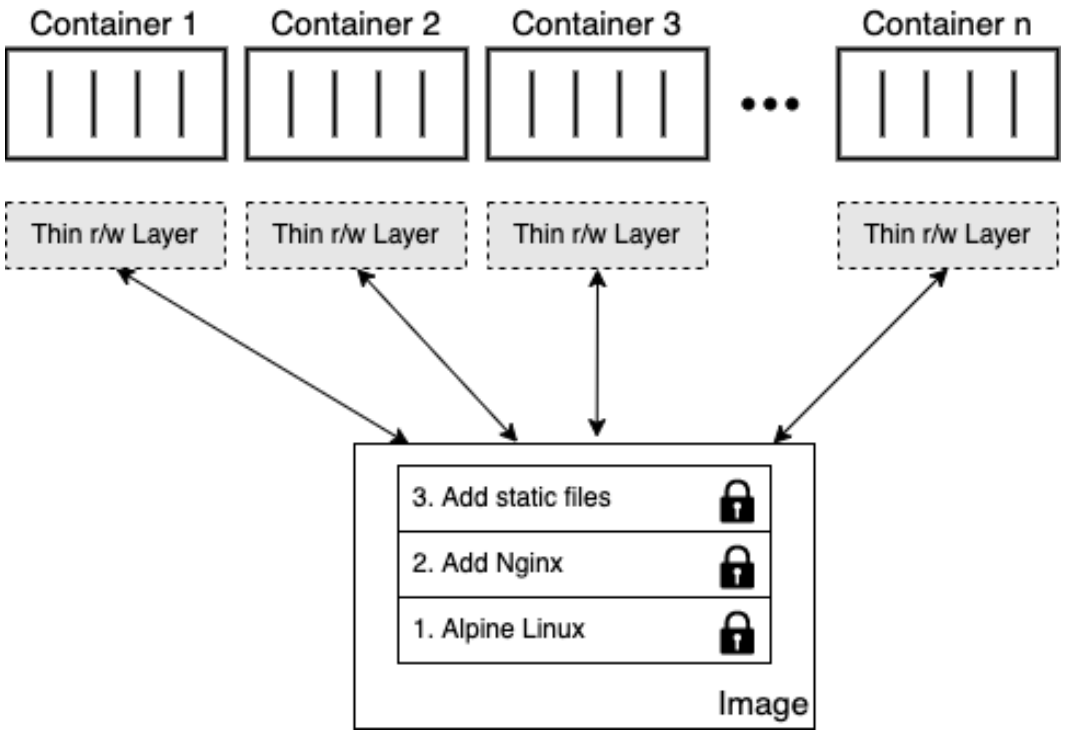


Figure 4.4: Multiple containers sharing the same image layers

This technique, of course, results in a tremendous reduction in the resources that are consumed. Furthermore, this helps decrease the loading time of a container since only a thin container layer has to be created once the image layers have been loaded into memory, which only happens for the first container.

Copy-on-write

Docker uses the copy-on-write technique when dealing with images. Copy-on-write is a strategy for sharing and copying files for maximum efficiency. If a layer uses a file or folder that is available in one of the lower-lying layers, then it just uses it. If, on the other hand, a layer wants to modify, say, a file from a lower-lying layer, then it first copies this file up to the target layer and then modifies it. In the following figure, we can see what this means:

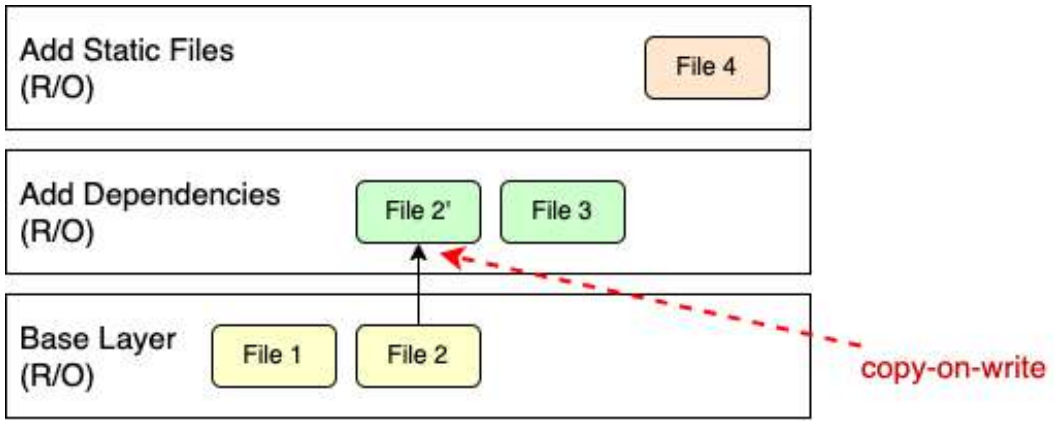


Figure 4.5: Docker image using copy-on-write

The second layer wants to modify **File 2**, which is present in the base layer. Thus, it copies it up and then modifies it; this is indicated by the apostrophe. Now, let's say that we're sitting in the top layer of the preceding graphic. This layer will use **File 1** from the base layer and **File 2** and **File 3** from the second layer.

Graph drivers

Graph drivers, also known as storage drivers, play a crucial role in Docker's layered architecture. They enable the Union filesystem, which allows Docker to efficiently manage and store layered container images.

Essentially, a graph driver merges multiple image layers into a single, coherent root filesystem. This unified view becomes the root filesystem within a container's mount namespace, dictating how the container accesses and interacts with stored data.

Docker uses a flexible, pluggable architecture that supports various graph drivers. The recommended and most widely used driver today is **overlay2**, due to its performance and efficiency advantages. Docker also supports the original **overlay** driver, although it has largely been superseded by **overlay2**.

Now that we understand what images are, we will learn how we can create a Docker image ourselves.

Creating Docker images

There are three ways to create a new container image on your system. The first one is by interactively building a container that contains all the additions and

changes you desire, and then committing those changes into a new image. The second, and most important, way is to use a Dockerfile to describe what's in the new image, and then build the image using that Dockerfile as a manifest. Finally, the third way of creating an image is by importing it into the system from a tarball.

Now, let's look at these three ways in detail.

Interactive image creation

The first way we can create a custom image is by interactively building a container. That is, we start with a base image that we want to use as a template and run a container of it interactively. Let's say that this is the Alpine image:

1. The command to run the container would be as follows:

```
$ docker container run -it \
    --name sample \
    alpine:3.21 /bin/sh
```

   The preceding command runs a container based on the `alpine:3.21` image.

2. We run the container interactively with an attached **teletypewriter** (**TTY**) using the `-it` parameter, name it `sample` with the `--name` parameter, and finally run a shell inside the container using `/bin/sh`.
   In the Terminal window where you ran the preceding command, you should see something like this:

```
❯ docker container run -it \
    --name sample \
    alpine:3.21 /bin/sh

Unable to find image 'alpine:3.21' locally
3.21: Pulling from library/alpine
Digest: sha256:a8560b36e8b8210634f77d9f7f9efd7ffa463e380b75e2e74aff4511df3ef88c
Status: Downloaded newer image for alpine:3.21
/ #
```

Figure 4.6: Alpine container in interactive mode

   By default, the Alpine container does not have the `curl` tool installed. Let's assume we want to create a new custom image that has `curl` installed.

3. Inside the container, we can then run the following command:

```
/ # apk update && apk add curl
```

The preceding command first updates the Alpine package manager, `apk`, and then it installs the `curl` tool. The output of the preceding command should look approximately like this:

```
/ # apk update && apk add curl
fetch https://dl-cdn.alpinelinux.org/alpine/v3.21/main/aarch64/APKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.21/community/aarch64/APKINDEX.tar.gz
v3.21.3-340-g8659f68d269 [https://dl-cdn.alpinelinux.org/alpine/v3.21/main]
v3.21.3-346-gd79468e34c0 [https://dl-cdn.alpinelinux.org/alpine/v3.21/community]
OK: 25249 distinct packages available
(1/9) Installing brotli-libs (1.1.0-r2)
(2/9) Installing c-ares (1.34.5-r0)
(3/9) Installing libunistring (1.2-r0)
(4/9) Installing libidn2 (2.3.7-r0)
(5/9) Installing nghttp2-libs (1.64.0-r0)
(6/9) Installing libpsl (0.21.5-r3)
(7/9) Installing zstd-libs (1.5.6-r2)
(8/9) Installing libcurl (8.12.1-r1)
(9/9) Installing curl (8.12.1-r1)
Executing busybox-1.37.0-r12.trigger
OK: 12 MiB in 24 packages
/ #
```

Figure 4.7: Installing curl on Alpine

4. Now, we can indeed use `curl`, for example, to access Google at `https://google.com`, as the following code snippet shows:

```
/ # curl -I https://google.com
HTTP/2 301
location: https://www.google.com/
content-type: text/html; charset=UTF-8
content-security-policy-report-only: object-src 'none';base-uri 'self';script-src 'nonce-zrl-heITqnKMzwl8zALLsg' 'strict-dynamic'
ample' 'unsafe-eval' 'unsafe-inline' https: http:;report-uri https://csp.withgoogle.com/csp/gws/other-hp
date: Sun, 20 Apr 2025 08:15:36 GMT
expires: Tue, 20 May 2025 08:15:36 GMT
cache-control: public, max-age=2592000
server: gws
content-length: 220
x-xss-protection: 0
x-frame-options: SAMEORIGIN
alt-svc: h3=":443"; ma=2592000,h3-29=":443"; ma=2592000

/ #
```

Figure 4.8: Using curl from within the container

With the preceding command, we have contacted the Google home page, and with the `-I` parameter, we have told `curl` to only output the response headers.

5. Once we have finished our customization, we can quit the container by typing `exit` at the prompt or hitting *Ctrl* + *D*.
6. Now, if we list all containers with the `docker container ls -a` command, we will see that our sample container has a status of `Exited`, but still

exists on the system, as shown in the following code block:

```
$ docker container ls -a | grep sample
```

7. This should output something similar to this:

```
❯ docker container ls -a | grep sample
8e3e4b0d5cc8   alpine:3.21   "/bin/sh"   8 minutes ago   Exited (0) About a minute ago   sample
```

Figure 4.9: The customized Docker container

8. If we want to see what has changed in our container concerning the base image, we can use the `docker container diff` command, as follows:

```
$ docker container diff sample
```

9. The output should present a list of all modifications done on the filesystem of the container, as follows:

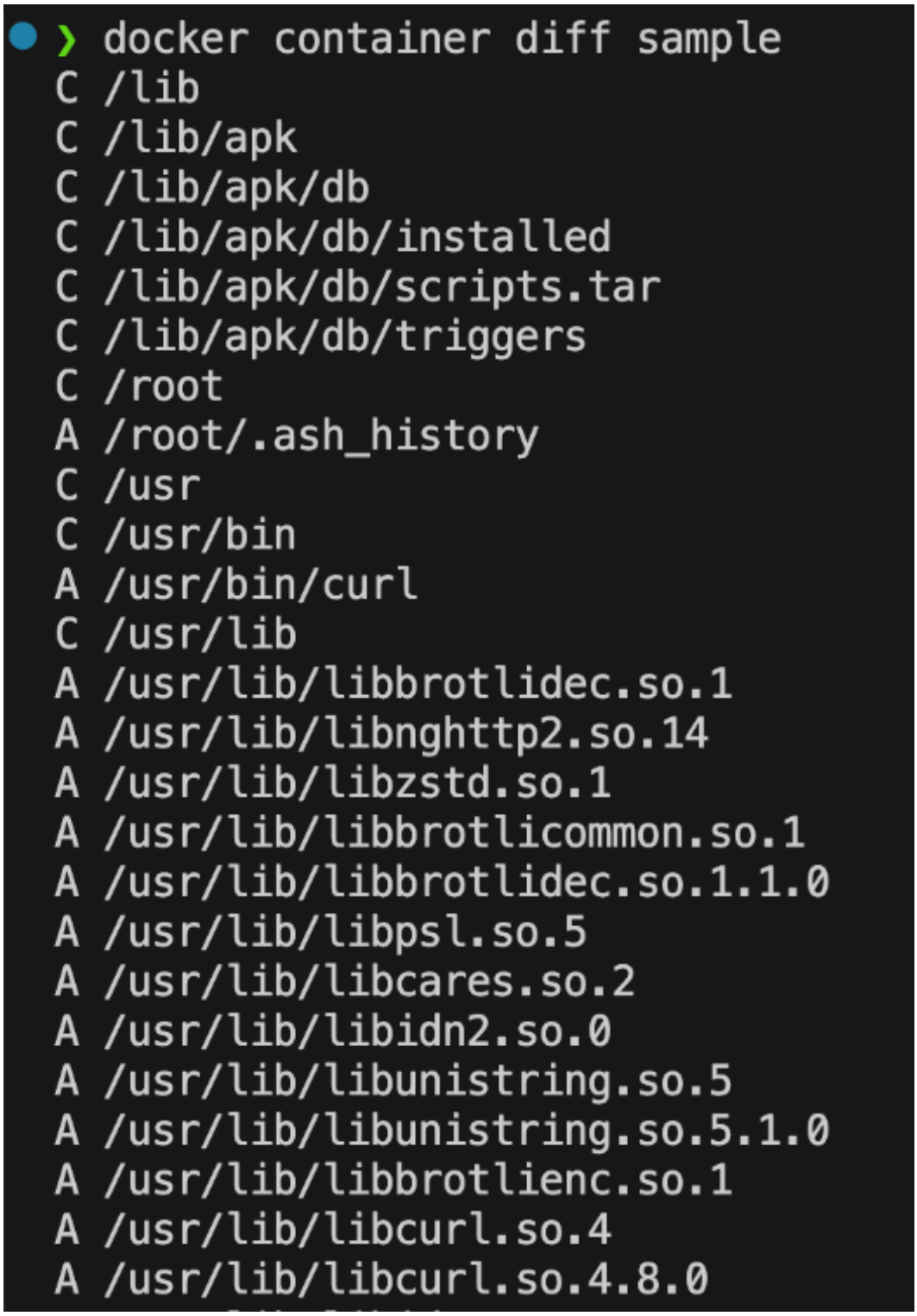


Figure 4.10: Output of the docker diff command (truncated)

We have shortened the preceding output for better readability. In the list, `A` stands for added (file or folder), and `C` stands for changed. If we

had any deleted files, then those would be prefixed with a `D`.

10. We can now use the `docker container commit` command to persist our modifications and create a new image from them, like this:

```
$ docker container commit sample my-alpine
```

11. The output generated by the preceding command on the author's computer is as follows:

```
sha256:78a1a06bef41e0cbe9d2228d9715a1dbb87...
```

12. With the preceding command, we have specified that the new image will be called `my-alpine`. The output generated by the preceding command corresponds to the ID of the newly generated image.
13. We can verify this by listing all the images on our system, as follows:

```
$ docker image ls
```

14. We can see this image ID as follows:

```
❯ docker image ls
REPOSITORY   TAG       IMAGE ID       CREATED              SIZE
my-alpine    latest    78a1a06bef41   About a minute ago   15.7MB
alpine       3.21      8d591b0b7dea   2 months ago         8.17MB
alpine       3.21.3    8d591b0b7dea   2 months ago         8.17MB
alpine       latest    8d591b0b7dea   2 months ago         8.17MB
nginx        alpine    cedb667e1a7b   2 months ago         49.4MB
busybox      latest    82c58a5c5b73   6 months ago         4.04MB
```

Figure 4.11: Listing all Docker images

We can see that the image named `my-alpine` has the expected ID of `78a1a06bef41` (corresponding to the first part of the full hash code) and automatically got a tag of `latest` assigned. This happened since we did not explicitly define a tag ourselves. In this case, Docker always defaults to the `latest` tag.

15. If we want to see how our custom image has been built, we can use the `history` command, as follows:

```
$ docker image history my-alpine
```

16. This will print a list of the layers our image consists of, as follows:

```
❯ docker image history my-alpine
IMAGE          CREATED         CREATED BY                                      SIZE      COMMENT
78a1a06bef41   4 minutes ago   /bin/sh                                         7.57MB
8d591b0b7dea   2 months ago    CMD ["/bin/sh"]                                 0B        buildkit.dockerfile.v0
<missing>      2 months ago    ADD alpine-minirootfs-3.21.3-aarch64.tar.gz …   8.17MB    buildkit.dockerfile.v0
```

Figure 4.12: History of the my-alpine Docker image

The top layer – marked in red – in the preceding output is the one that we just created by adding the `curl` package. The other two lines stem from the original build of the Alpine 3.21 Docker image. It was created and uploaded 2 months ago.

Now that we have seen how we can interactively create a Docker image, let's look into how we can do the same declaratively using a Dockerfile.

Using Dockerfiles

Manually creating custom images, as shown in the previous section of this chapter, is very helpful when doing exploration, creating prototypes, or authoring feasibility studies. But it has a serious drawback: it is a manual process and thus is not repeatable or scalable. It is also error-prone, just like any other task executed manually by humans. There must be a better way.

This is where the so-called Dockerfile comes into play. A **Dockerfile** is a text file that, by default, is called Dockerfile. It contains instructions on how to build a custom container image. It is a declarative way of building images.

NOTE

Declarative versus imperative

In computer science, in general, and with Docker specifically, you often use a declarative way of defining a task. You describe the expected outcome and let the system figure out how to achieve this goal, rather than giving step-by-step instructions to the system on how to achieve this desired outcome. The latter is an imperative approach.

Let's look at a sample Dockerfile, as follows:

```
FROM python:3.12
RUN mkdir -p /app
WORKDIR /app
COPY ./requirements.txt /app/
RUN pip install -r requirements.txt
CMD ["python", "main.py"]
```

This is a Dockerfile used to containerize a Python version 3.12 application. As we can see, the file has six lines, each starting with a keyword such as `FROM`, `RUN`, or `COPY`.

NOTE

Note

It is a convention to write the keywords in all caps, but that is not a must.

Each line of the Dockerfile results in a layer in the resulting image. In the following figure, the image is drawn upside down compared to the previous figures in this chapter, showing an image as a stack of layers. Here, the base layer is shown on top. Don't let yourself be confused by this. In reality, the base layer is always the lowest in the stack:

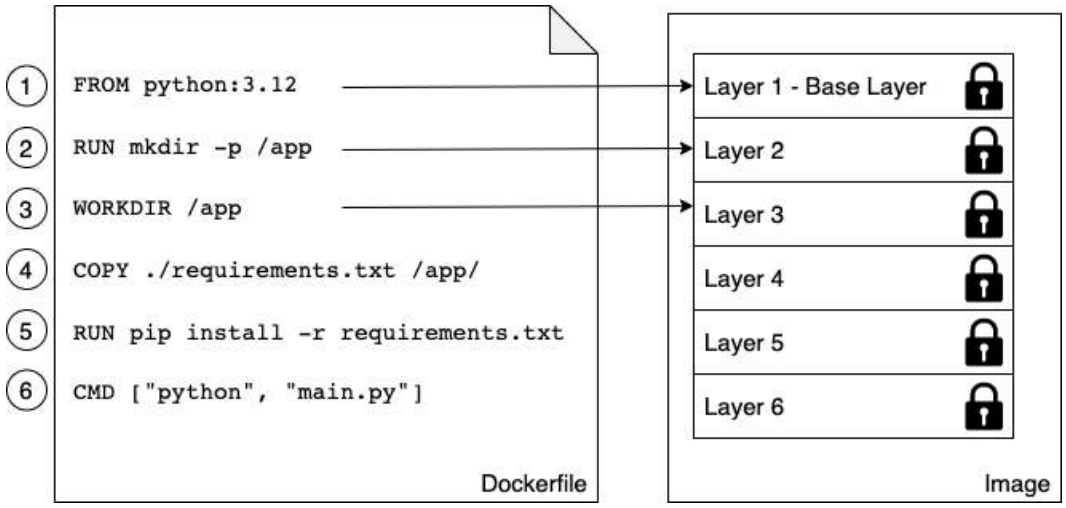


Figure 4.13: The relationship between a Dockerfile and the layers in an image

Now, let's look at the individual keywords in more detail.

The FROM keyword

Every Dockerfile starts with the `FROM` keyword. With it, we define which base image we want to start building our custom image from. If we want to build starting with Ubuntu 24.10, for example, we will have the following line in the Dockerfile:

```
FROM ubuntu:24.10
```

On Docker Hub, there are curated or official images for all major Linux distros, as well as for all important development frameworks or languages, such as Python, Node.js, Ruby, Go, and many more. Depending on our needs, we should select the most appropriate base image.

For example, if I want to containerize a Python 3.12 application, I might want to select the relevant official `python:3.12` image.

If we want to start from scratch, we can also use the following statement:

```
FROM scratch
```

This is useful in the context of building super-minimal images that only – for example – contain a single binary: the actual statically linked executable, such as `Hello-World`. The scratch image is an empty base image.

`FROM scratch`, in reality, is a no-op in the Dockerfile, and as such does not generate a layer in the resulting container image.

The RUN keyword

The next important keyword is `RUN`. The argument for `RUN` is any valid Linux command, such as the following:

```
RUN yum install -y wget
```

The preceding command is using the `yum` CentOS package manager to install the `wget` package in the running container. This assumes that our base image is CentOS or **Red Hat Enterprise Linux** (**RHEL**). If we had Ubuntu as our base image, then the command would look similar to the following:

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

It would look like this because Ubuntu uses `apt-get` as a package manager.

Similarly, we could define a line with `RUN`, like this:

```
RUN mkdir -p /app && cd /app
```

We could also do this:

```
RUN tar -xJC /usr/src/python --strip-components=1 \
    -f python.tar.xz
```

Here, the former creates an `/app` folder in the container and navigates to it, and the latter un-tars a file to a given location. It is completely fine, and even recommended, for you to format a Linux command using more than one physical line, such as this:

```
RUN apt-get update \
  && apt-get install -y --no-install-recommends \
    ca-certificates \
    libexpat1 \
    libffi6 \
    libgdbm3 \
    libreadline7 \
    libsqlite3-0 \
    libssl1.1 \
  && rm -rf /var/lib/apt/lists/*
```

If we use more than one line, we need to put a backslash (\) at the end of the lines to indicate to the shell that the command continues on the next line.

NOTE

Tip

Try to find out what the preceding command does.

The COPY and ADD keywords

The `COPY` and `ADD` keywords are very important since, in the end, we want to add some content to an existing base image to make it a custom image. Most of the time, these are a few source files of, say, a web application, or a few binaries of a compiled application.

These two keywords are used to copy files and folders from the host into the image that we're building. The two keywords are very similar, with the exception that the `ADD` keyword also lets us copy and unpack `TAR` files, and provides a URL as a source for the files and folders to copy.

Let's look at a few examples of how these two keywords can be used, as follows:

```
COPY . /app
COPY ./web /app/web
COPY sample.txt /data/my-sample.txt
ADD sample.tar /app/bin/
ADD http://example.com/sample.txt /data/
```

In the preceding lines of code, the following applies:

- The first line copies all files and folders from the current directory recursively to the `app` folder inside the container image
- The second line copies everything in the `web` subfolder to the target folder, `/app/web`
- The third line copies a single file, `sample.txt`, into the target folder, `/data`, and at the same time, renames it `my-sample.txt`
- The fourth statement unpacks the `sample.tar` file into the target folder, `/app/bin`
- Finally, the last statement copies the remote file, `sample.txt`, into the target file, `/data`

Wildcards are allowed in the source path. For example, the following statement copies all files starting with `sample` to the `mydir` folder inside the image:

```
COPY ./sample* /mydir/
```

From a security perspective, it is important to know that, by default, all files and folders inside the image will have a **user ID** (**UID**) and a **group ID** (**GID**) of 0. The good thing is that for both `ADD` and `COPY`, we can change the ownership that the files will have inside the image using the optional `--chown` flag, as follows:

```
ADD --chown=11:22 ./data/web* /app/data/
```

The preceding statement will copy all files starting with `web` and put them into the `/app/data` folder in the image, and at the same time assign user 11 and group 22 to these files.

Instead of numbers, we could also use names for the user and group, but then these entities would have to be already defined in the root filesystem of the image at `/etc/passwd` and `/etc/group`, respectively; otherwise, the build of the image would fail.

The WORKDIR keyword

The `WORKDIR` keyword defines the working directory or context that is used when a container is run from our custom image. So, if I want to set the context

to the `/app/bin` folder inside the image, my expression in the Dockerfile would have to look as follows:

```
WORKDIR /app/bin
```

All activity that happens inside the image after the preceding line will use this directory as the working directory. It is very important to note that the following two snippets from a Dockerfile are not the same:

```
RUN cd /app/bin
RUN touch sample.txt
```

Compare the preceding code with the following code:

```
WORKDIR /app/bin
RUN touch sample.txt
```

The former will create the file in the root of the image filesystem, while the latter will create the file at the expected location in the `/app/bin` folder. Only the `WORKDIR` keyword sets the context across the layers of the image. The `cd` command alone is not persisted across layers.

NOTE

Note

It is completely fine to change the current working directory multiple times in a Dockerfile.

The CMD and ENTRYPOINT keywords

The `CMD` and `ENTRYPOINT` keywords are special. While all other keywords defined for a Dockerfile are executed at the time the image is built by the Docker builder, these two are definitions of what will happen when a container is started from the image we define. When the container runtime starts a container, it needs to know what the process or application will be that has to run inside that container. That is exactly what `CMD` and `ENTRYPOINT` are used for – to tell Docker what the start process is and how to start that process.

Now, the differences between `CMD` and `ENTRYPOINT` are subtle, and honestly, most users don't fully understand them or use them in the intended way. Luckily, in most cases, this is not a problem, and the container will run

anyway; it's just that handling them is not always as straightforward as it could be.

To better understand how to use these two keywords, let's analyze what a typical Linux command or expression looks like. Let's take the `ping` utility as an example, as follows:

```
$ ping -c 3 8.8.8.8
```

In the preceding expression, `ping` is the command, and `-c 3 8.8.8.8` are the parameters of this command. Let's look at another expression here:

```
$ wget -O - http://example.com/downloads/script.sh
```

Again, in the preceding expression, `wget` is the command, and `-O - http://example.com/downloads/script.sh` are the parameters.

Now that we have dealt with this, we can get back to `CMD` and `ENTRYPOINT`. `ENTRYPOINT` is used to define the command of the expression, while `CMD` is used to define the parameters for the command. Thus, a Dockerfile using Alpine as the base image and defining `ping` as the process to run in the container could look like this:

```
FROM alpine:3.21
RUN apk update && apk add curl
ENTRYPOINT ["ping"]
CMD ["-c","3","8.8.8.8"]
```

For both `ENTRYPOINT` and `CMD`, the values are formatted as a JSON array of strings, where the individual items correspond to the tokens of the expression that are separated by whitespace. This is the preferred way of defining `CMD` and `ENTRYPOINT`. It is also called the **exec** form.

Alternatively, we can use what's called the shell form, as shown here:

```
CMD command param1 param2
```

NOTE

Note

You can find the preceding Dockerfile in the sample code for *Chapter 04*, subfolder `solutions/pinger`.

We can now build an image called pinger from the preceding Dockerfile, as follows:

```
$ docker image build -t pinger .
```

Here is the output generated by the preceding command:

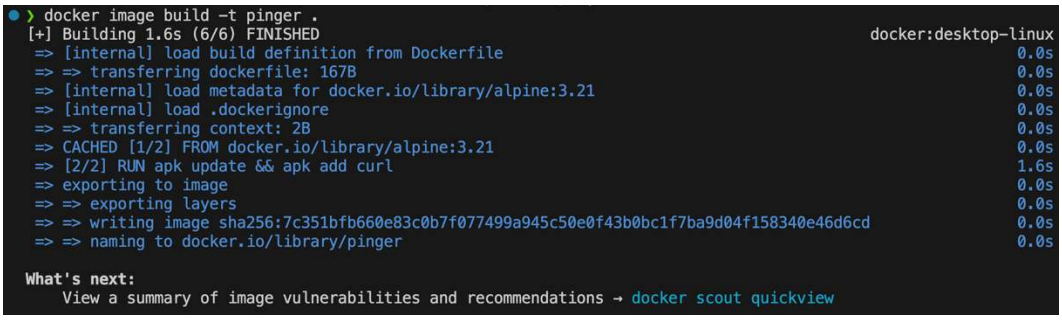


Figure 4.14: Building the pinger Docker image

Then, we can run a container from the pinger image we just created, like this:

```
$ docker container run --rm -it pinger
```

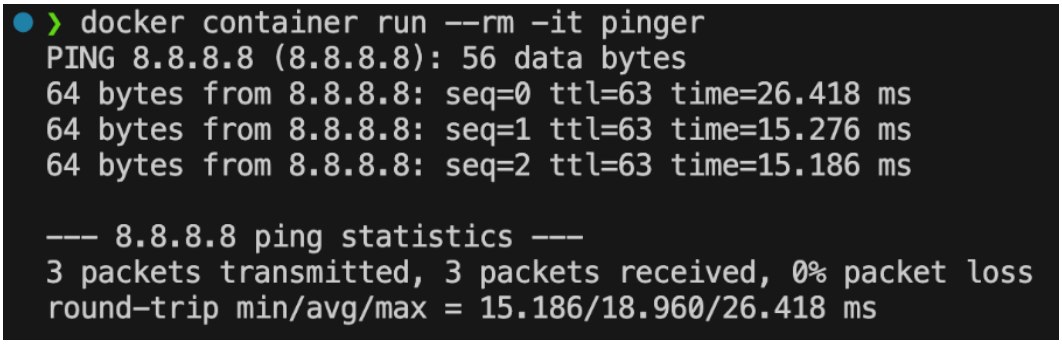


Figure 4.15: Output of the pinger container

In the preceding command, we are using the `--rm` parameter, which defines that the container is automatically removed once the applications inside the container end.

The beauty of this is that we can now override the CMD part that we have defined in the Dockerfile (remember, it was `["-c", "3","8.8.8.8"]`) when we create a new container by adding the new values at the end of the `docker container run` expression, like this:

```
$ docker container run --rm -it pinger -w 5 127.0.0.1
```

This will cause the container to ping the loopback IP address (`127.0.0.1`) for 5 seconds.

If we want to override what's defined in `ENTRYPOINT` in the Dockerfile, we need to use the `--entrypoint` parameter in the `docker container run` expression. Let's say we want to execute a shell (`ash` for Alpine shell) in the container instead of the `ping` command. We could do so by using the following command:

```
$ docker container run --rm -it --entrypoint ash pinger
```

We will then find ourselves inside the container. Type `exit` or press *Ctrl* + *D* to leave the container.

As I already mentioned, we do not necessarily have to follow best practices and define the command through `ENTRYPOINT` and the parameters through `CMD`; instead, we can enter the whole expression as a value of `CMD` and it will work, as shown in the following code block:

```
FROM alpine:3.21
CMD wget -O - http://www.google.com
```

Here, I have even used the shell form to define the CMD. But what happens in this situation if `ENTRYPOINT` is undefined? If you leave `ENTRYPOINT` undefined, then it will have the default value of `/bin/sh -c`, and whatever the value of `CMD` is will be passed as a string to the shell command. The preceding definition would thereby result in entering the following code to run the process inside the container:

```
/bin/sh -c "wget -O - http://www.google.com"
```

Consequently, `/bin/sh` is the main process running inside the container, and it will start a new child process to run the `wget` utility.

A complex Dockerfile

So far, we have discussed the most important keywords commonly used in Dockerfiles. Now, let's look at a realistic and somewhat complex example of a Dockerfile. Those of you who are interested might note that it looks very similar to the first Dockerfile that we presented in this chapter. Here is its content:

```
FROM node:23-bookworm
RUN mkdir -p /app
WORKDIR /app
COPY package.json /app/
RUN npm install
COPY . /app
ENTRYPOINT ["npm"]
CMD ["start"]
```

OK, so what is happening here? This is a Dockerfile that is used to build an image for a Node.js application; we can deduce this from the fact that the `node:23-bookworm` base image is used. Then, the second line is an instruction to create an `/app` folder in the filesystem of the image. The third line defines the working directory or context in the image to be this new `/app` folder. Then, on line four, we copy a `package.json` file into the `/app` folder inside the image. After this, on line five, we execute the `npm install` command inside the container; remember, our context is the `/app` folder, so `npm` will find the `package.json` file there that we copied on line four.

Once all the Node.js dependencies have been installed, we copy the rest of the application files from the current folder of the host into the `/app` folder of the image.

Finally, in the last two lines, we define what the startup command will be when a container is run from this image. In our case, it is `npm start`, which will start the Node.js application.

NOTE

Note

You'll find the preceding Dockerfile and a trivial Node.js application in the sample code for *Chapter 4*, subfolder `solutions/node-sample`.

Building an image

Let's look at a concrete example and build a simple Docker image, as follows:

1. Navigate to the sample code repository. Normally, this should be located in your home folder:

```
$ cd ~/The-Ultimate-Docker-Container-Book-Fourth-Edition
```

2. If it doesn't already exist, create a new subfolder for *Chapter 4* and navigate to it:

```
$ mkdir chapter-04 && cd chapter-04
```

3. In the preceding folder, create a `sample-1` subfolder and navigate to it, like this:

```
$ mkdir sample-1 && cd sample-1
```

4. Use your favorite editor to create a file called `Dockerfile` inside this sample folder, with the following content:

```
FROM ubuntu:25.04
RUN apt-get update && apt-get install -y wget
```

5. Save the file and exit your editor.
6. Back in the Terminal window, we can now build a new container image using the preceding Dockerfile as a manifest or construction plan, like this:

```
$ docker image build -t my-ubuntu .
```

Please note that there is a period (`.`) at the end of the preceding command. The following screenshot shows the command in action:

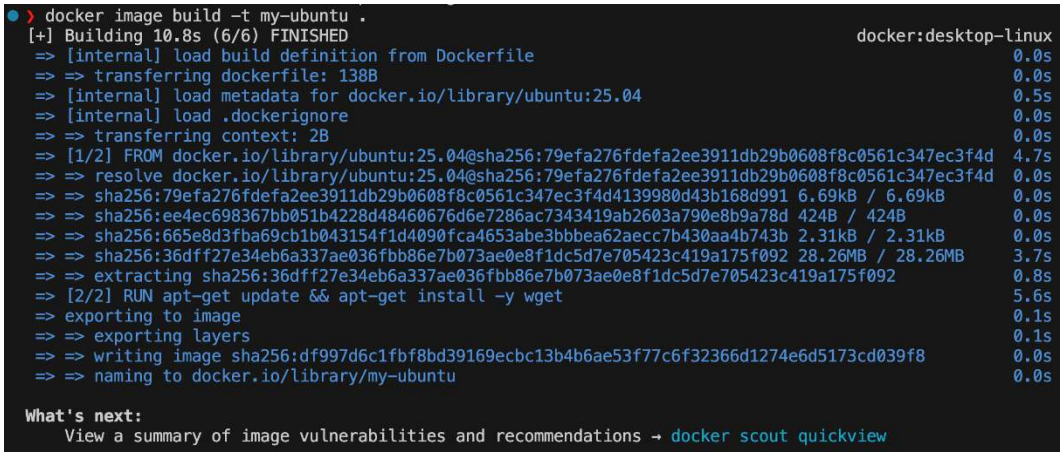


Figure 4.16: Building our first custom image from Ubuntu 25.04

The previous command means that the Docker builder is creating a new image called `my-ubuntu` using the Dockerfile that is present in the current directory. Here, the period at the end of the command specifies the

current directory. We could also write the preceding command as follows, with the same result:

```
$ docker image build -t my-ubuntu -f Dockerfile .
```

Here, we can omit the `-f` parameter since the builder assumes that the Dockerfile is called `Dockerfile`. We only ever need the `-f` parameter if our Dockerfile has a different name or is not located in the current directory.

Let's analyze the output shown in *Figure 4.16*. This output is created by the Docker build kit:

1. First, we have the following line:

```
[+] Building 10.8s (6/6) FINISHED
```

This line is generated at the end of the build process, although it appears as the first line. It tells us that the building took approximately 11 seconds and was executed in 6 steps.

2. Now, let's skip the next few lines until we reach this one:

```
=> [1/2] FROM docker.io/library/ubuntu:25.04@sha256...
```

This line tells us which line of the Dockerfile the builder is currently executing (1 of 2). We can see that this is the `FROM ubuntu:25.04` statement in our Dockerfile. This is the declaration of the base image, on top of which we want to build our custom image. What the builder then does is pull this image from Docker Hub if it is not already available in the local cache.

3. Now, follow the next step. I have shortened it even more than the preceding one to concentrate on the essential part:

```
=> [2/2] RUN apt-get update && apt-get install -y wget
```

This is our second line in the Dockerfile, where we want to use the `apt-get` package manager to install the `wget` utility.

4. The last few lines are as follows:

```
=> exporting to image                          0.1s
=> => exporting layers                         0.1s
```

```
 => => writing image sha256:df997d6c1fb...        0.0s
 => => naming to docker.io/library/my-ubuntu.     0.0s
```

5. Here, the builder finalizes building the image and provides the image with the sha256 code of `df997d6c1fb...`.
   This tells us that the resulting custom image has been given an ID of `df997d6c1fb...` and has been tagged with the name `my-ubuntu:latest`.

Now that we have analyzed how the build process of a Docker image works and what steps are involved, let's talk about how to further improve this by introducing multi-step builds.

Working with multi-step builds

To demonstrate why a Dockerfile with multiple build steps is useful, let's make an example Dockerfile. Let's take a **Hello World** application written in C:

1. Open a new Terminal window and navigate to this chapter's folder:

```
$ cd The-Ultimate-Docker-Container-Book-Fourth-Edition/chapter-04
```

2. Create a new folder called `multi-step-build` in your chapter folder:

```
$ mkdir multi-step-build
```

3. Open VS Code for this folder:

```
$ code multi-step-build
```

4. Create a file called `hello.c` in this folder and add the following code to it:

```
#include <stdio.h>
int main (void)
{
    printf ("Hello, world!\n");
    return 0;
}
```

5. Now, we want to containerize this application and write a Dockerfile in the same folder with this content:

```
FROM alpine:3.12
RUN apk update && \
    apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
```

```
COPY . /app
RUN mkdir bin
RUN gcc -Wall hello.c -o bin/hello
ENTRYPOINT ["/app/bin/hello"]
```

6. Next, let's build this image:

```
$ docker image build -t hello-world .
```

This gives us a fairly long output since the builder must install the Alpine **Software Development Kit** (**SDK**), which, among other tools, contains the C++ compiler we need to build the application.

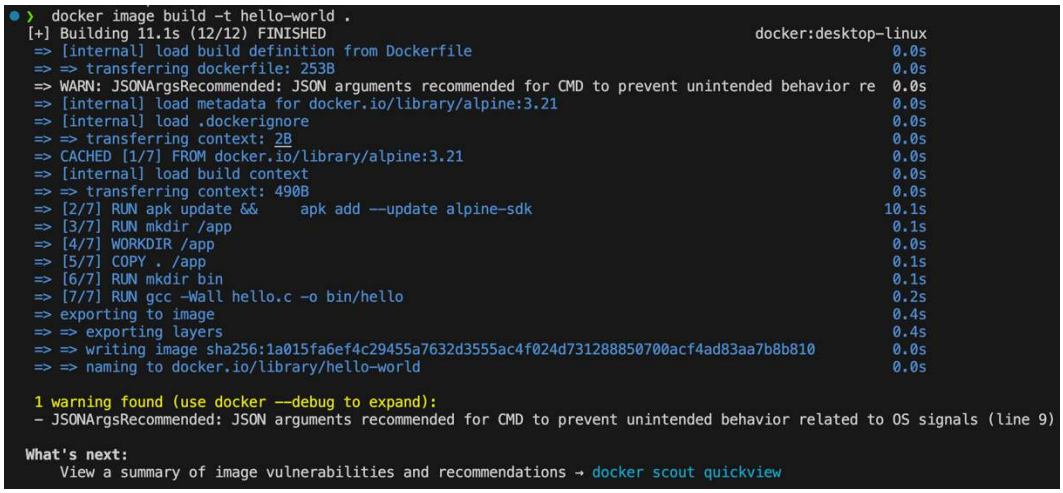


Figure 4.17: Building the Docker image for the C application

7. Once the build is done, we can list the image and see its size that's been shown, as follows:

```
$ docker image ls | grep hello-world
```

In the author's case, the output is as follows:

Figure 4.18: Size of the unoptimized Docker image

With a size of 260 MB, the resulting image is way too big. In the end, it is just a **Hello World** application. The reason for it being so big is that the image not only contains the Hello World binary but also all the tools to compile and link the application from the source code. But this is not desirable when running

the application, say, in production. Ideally, we only want to have the resulting binary in the image and not a whole SDK.

It is precisely for this reason that we should define Dockerfiles as multi-stage. We have some stages that are used to build the final artifacts, and then a final stage, where we use the minimal necessary base image and copy the artifacts into it. This results in very small Docker images. Let's do this:

1. Create a new Dockerfile in your folder called `Dockerfile.multi-step` with this content:

```
FROM alpine:3.21 AS build
RUN apk update && \
    apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
COPY . /app
RUN mkdir bin
# Compile statically so runtime has no dependencies
RUN gcc -static -O2 hello.c -o bin/hello

FROM scratch
COPY --from=build /app/bin/hello /app/hello
ENTRYPOINT ["/app/hello"]
```

   Here, we have the first stage with an alias called `build` that is used to compile the application; then, the second stage uses the `scratch` base image and does not install the SDK, but only copies the binary from the `build` stage, using the `--from` parameter, into this final image.

2. Let's build the image again, as follows:

```
$ docker image build -t hello-world-small \
    -f Dockerfile.multi-step .
```

3. Let's compare the sizes of the images with this command:

```
$ docker image ls | grep hello-world
```

   Here, we get the following output:

```
❯ docker image ls | grep hello-world
hello-world         latest    5620e48864bd   50 seconds ago      260MB
hello-world-small   latest    95e5dfaf6dab   4 minutes ago       136kB
```

Figure 4.19: Comparing sizes of Docker images

We have been able to reduce the size from 260 MB to a mere 136 KB. This is a reduction in size by more than 3 magnitudes! A smaller image has many advantages, such as a smaller attack surface area for hackers, reduced memory and disk consumption, faster startup times of the corresponding containers, and a reduction of the bandwidth needed to download the image from a registry, such as Docker Hub.

Dockerfile best practices

In this section, we will list down a few recommended best practices to consider when authoring a Dockerfile, which are as follows:

Containers are ephemeral

First and foremost, we need to consider that containers are meant to be ephemeral. By ephemeral, we mean that a container can be stopped and destroyed, and a new one built and put in place with an absolute minimum of setup and configuration. That means that we should try hard to keep the time that is needed to initialize the application running inside the container to a minimum, as well as the time needed to terminate or clean up the application.

Leverage the immutability of container image layers

The next best practice tells us that we should order the individual commands in the Dockerfile so that we leverage caching as much as possible. Building a layer of an image can take a considerable amount of time – sometimes many seconds, or even minutes. While developing an application, we will have to build the container image for our application multiple times. We want to keep the build times to a minimum.

When we're rebuilding a previously built image, the only layers that are rebuilt are the ones that have changed, but if one layer needs to be rebuilt, all subsequent layers also need to be rebuilt. This is very important to remember. Consider the following example:

```
FROM node:23-bookworm
RUN mkdir -p /app
WORKIR /app
COPY . /app
RUN npm install
CMD ["npm", "start"]
```

In this example, the `npm install` command on line five of the Dockerfile usually takes the longest. A classical Node.js application has many external dependencies, all of which are downloaded and installed during this step. It can take minutes until it is done. To save time, we want to avoid running `npm install` each time we rebuild the image; however, developers often make changes to their source code during the development of an application. This means that line four, the result of the `COPY` command, changes every time, and thus, this layer has to be rebuilt. But as we discussed previously, that also means that all subsequent layers have to be rebuilt, which – in this case – includes the `npm install` command. To avoid this, we can slightly modify the Dockerfile and have the following:

```
FROM node:23-bookworm
RUN mkdir -p /app
WORKIR /app
COPY package.json /app/
RUN npm install
COPY . /app
CMD ["npm", "start"]
```

Here, on line four, we only copied the single file that the `npm install` command needs as a source, which is the `package.json` file. This file rarely changes in a typical development process. As a consequence, the `npm install` command also has to be executed only when the `package.json` file changes. All the remaining frequently changed content is added to the image after the `npm install` command.

Minimize the number of layers

A further best practice is to keep the number of layers that make up your image relatively small. The more layers an image has, the more the graph driver needs to work to consolidate the layers into a single root filesystem for the corresponding container. Of course, this takes time, and thus, the fewer layers an image has, the faster the startup time for the container can be.

But how can we keep our number of layers low? Remember that, in a Dockerfile, each line that starts with a keyword such as `FROM`, `COPY`, or `RUN` creates a new layer. The easiest way to reduce the number of layers is to combine multiple individual `RUN` commands into a single one. For example, say that we had the following in a Dockerfile:

```
...
RUN apt-get update
RUN apt-get install -y ca-certificates
RUN rm -rf /var/lib/apt/lists/*
...
```

We could combine these into a single concatenated expression, as follows:

```
...
RUN apt-get update \
    && apt-get install -y ca-certificates \
    && rm -rf /var/lib/apt/lists/*
...
```

The former will generate three layers in the resulting image, while the latter will only create a single layer.

Keeping container image sizes minimal

The next three best practices all result in smaller images. Why is this important? Smaller images reduce the time and bandwidth needed to download the image from a registry. They also reduce the amount of disk space needed to store a copy locally on the Docker host and the memory needed to load the image. Finally, smaller images also mean a smaller attack surface for hackers. Here are the best practices mentioned:

- The first best practice that helps reduce the image size is to use a `.dockerignore` file. We want to avoid copying unnecessary files and folders into an image, to keep it as lean as possible. A `.dockerignore` file works in the same way as a `.gitignore` file, for those who are familiar with Git. In a `.dockerignore` file, we can configure patterns to exclude certain files or folders from being included in the context when building the image.
- The next best practice is to avoid installing unnecessary packages in the filesystem of the image. Once again, this is to keep the image as lean as possible.
- Last but not least, it is recommended that you use multi-stage builds so that the resulting image is as small as possible and only contains the absolute minimum needed to run your application or application service.

In the next section, we are going to learn how to create a Docker image from a previously saved image. In fact, it may look like restoring an image.

Saving and loading images

The third way to create a new container image is by importing or loading it from a file. A container image is nothing more than a tarball. To demonstrate this, we can use the `docker image save` command to export an existing image to a tarball, like this:

```
$ mkdir backup
$ docker image save -o ./backup/my-alpine.tar my-alpine
```

The preceding command takes our `my-alpine` image that we previously built and exports it into a file called `./backup/my-alpine.tar`:

```
❯ mkdir backup
❯ docker image save -o ./backup/my-alpine.tar my-alpine
❯ ls -al backup
total 31392
drwxr-xr-x@ 3 gabrielschenker  staff        96 Apr 20 11:57 .
drwxr-xr-x@ 7 gabrielschenker  staff       224 Apr 20 11:57 ..
-rw-------@ 1 gabrielschenker  staff  16072704 Apr 20 11:57 my-alpine.tar
```

Figure 4.20: Exporting an image as a tarball

If, on the other hand, we have an existing tarball and want to import it as an image into our system, we can use the `docker image load` command, as follows:

```
$ docker image load -i ./backup/my-alpine.tar
```

The output of the preceding command should be as follows:

```
Loaded image: my-alpine:latest
```

With this, we have learned how to build a Docker image in three different ways. We can do so interactively, by defining a Dockerfile, or by importing it into our system from a tarball.

In the next section, we will discuss how we can create Docker images for existing legacy applications, and thus run them in a container, and profit from this.

Containerizing a legacy app using the lift and shift approach

We can't always start from scratch and develop a brand-new application. More often than not, we find ourselves with a huge portfolio of traditional applications that are up and running in production and provide mission-critical value to the company or the customers of the company. Often, those applications are organically grown and very complex. Documentation is sparse, and nobody wants to touch such an application. Often, the saying *Never touch a running system* applies. Yet, market needs change, and with that arises the need to update or rewrite those apps. Often, a complete rewrite is not possible due to the lack of resources and time, or due to the excessive cost. What are we going to do about those applications? Could we possibly Dockerize them and profit from the benefits introduced by containers?

It turns out we can. In 2017, Docker introduced a program called **Modernize Traditional Apps** (**MTA**) to their enterprise customers, which in essence promised to help those customers take their existing or traditional Java and .NET applications and containerize them, without the need to change a single line of code. The focus of MTA was on Java and .NET applications since those made up the lion's share of the traditional applications in a typical enterprise. But the same is possible for any application that was written in, say, C, C++, Python, Node.js, Ruby, PHP, or Go, to name just a few other languages and platforms.

Let's imagine such a legacy application for a moment. Let's assume we have an old Java application that was written 10 years ago, and that was continuously updated during the following 5 years. The application is based on Java SE 6, which came out in December 2006. It uses environment variables and property files for configuration. Secrets such as usernames and passwords used in the database connection strings are pulled from a secrets keystore, such as HashiCorp's Vault.

Now, let's describe each of the required steps to lift and shift a legacy application in more detail.

Analyzing external dependencies

One of the first steps in the modernization process is to discover and list all external dependencies of the legacy application:

- Does it use a database? If so, which one? What does the connection string look like?
- Does it use external APIs such as credit card approval or geo-mapping APIs? What are the API keys and key secrets?
- Is it consuming from or publishing to an **Enterprise Service Bus** (**ESB**)?

These are just a few possible dependencies that come to mind. Many more exist. These are the seams of the application to the outer world, and we need to be aware of them and create an inventory.

Preparing source code and build instructions

The next step is to locate all the source code and other assets, such as images, CSS, and HTML files, that are part of the application. Ideally, they should be located in a single folder. This folder will be the root of our project and can have as many subfolders as needed. This project root folder will be the context during the build of the container image we want to create for our legacy application. Remember, the Docker builder only includes files in the build that are part of that context; in our case, that is the root project folder.

There is, though, an option to download or copy files during the build from different locations, using the `COPY` or `ADD` commands. Please refer to the online documentation for the exact details on how to use these two commands. This option is useful if the sources for your legacy application cannot be easily contained in a single, local folder.

Once we are aware of all the parts that are contributing to the final application, we need to investigate how the application is built and packaged. In our case, this is most probably done by using **Maven**. Maven is the most popular build automation tool for Java, and has been – and still is – used in most enterprises that develop Java applications. In the case of a legacy .NET application, it is most probably done by using the MSBuild tool, and in the case of a C/C++ application, `make` would most likely be used.

Once again, let's extend our inventory and write down the exact build commands used. We will need this information later on, when authoring the Dockerfile.

Configuration

Applications need to be configured. Information provided during configuration could be – for example – the type of application logging to use, connection strings to databases, and hostnames to services such as ESBs or URIs to external APIs, to name just a few.

We can differentiate a few types of configurations, as follows:

- **Build time**: This is the information needed during the build of the application and/or its Docker image. It needs to be available when we create the Docker images.
- **Environment**: This is configuration information that varies with the environment in which the application is running – for example, *development*, *staging*, or *production*. This kind of configuration is applied to the application when a container with the app starts – for example, in production.
- **Runtime**: This is information that the application retrieves during runtime, such as secrets to access an external API.

Secrets

Every mission-critical enterprise application needs to deal with secrets in some form or another. The most familiar secrets are part of the connection information needed to access databases that are used to persist the data produced by or used by the application. Other secrets include the credentials needed to access external APIs, such as a credit score lookup API. It is important to note that, here, we are talking about secrets that have to be provided by the application itself to the service providers it uses or depends on, and not to secrets provided by the users of the application. The actor here is our own application, which needs to be authenticated and authorized by external authorities and service providers.

There are various ways traditional applications get their secrets. The worst and most insecure way of providing secrets is by hardcoding them or reading them from configuration files or environment variables, where they are available in cleartext. A much better way is to read the secrets during runtime from a special secret store that persists the secrets encrypted and provides

them to the application over a secure connection, such as **Transport Layer Security** (**TLS**).

Once again, we need to create an inventory of all secrets that our application uses and the way it procures them. Thus, we need to ask ourselves where we can get our secrets from: is it through environment variables or configuration files, or is it by accessing an external keystore, such as HashiCorp's Vault, AWS Secrets Manager, or Azure's Secrets Manager?

Authoring the Dockerfile

Once we have a complete inventory of all the items we discussed in the previous few sections, we are ready to author our Dockerfile. But I want to warn you: don't expect this to be a one-shot-and-go task. You may need several iterations until you have crafted your final Dockerfile. The Dockerfile may be rather long and ugly looking, but that's not a problem, as long as we get a working Docker image. We can always fine-tune the Dockerfile once we have a working version.

The base image

Let's start by identifying the base image we want to use and build our image from. Is there an official Java image available that is compatible with our requirements? Remember that our application is based on Java SE 6. If such a base image is available, then we should use that one. Otherwise, we will want to start with a Linux distro such as Red Hat, Oracle, or Ubuntu. In the latter case, we will use the appropriate package manager of the distro (`yum`, `apt`, or another) to install the desired versions of Java and Maven. For this, we can use the `RUN` keyword in the Dockerfile. Remember, `RUN` allows us to execute any valid Linux command in the image during the build process.

Assembling the sources

In this step, we make sure all the source files and other artifacts needed to successfully build the application are part of the image. Here, we mainly use the two keywords of the Dockerfile: `COPY` and `ADD`. Initially, the structure of the source inside the image should look the same as on the host, to avoid any build problems. Ideally, you would have a single `COPY` command that copies all of the root project folders from the host into the image. The corresponding Dockerfile snippet could then look as simple as this:

```
WORKDIR /app
COPY . .
```

NOTE

Note

Don't forget to also provide a `.dockerignore` file, which is located in the project root folder, which lists all the files and (sub) folders of the project root folder that should *not* be part of the build context.

As mentioned earlier, you can also use the `ADD` keyword to download sources and other artifacts into the Docker image that are not located in the build context but somewhere reachable by a URI, as shown here:

```
ADD http://example.com/foobar ./
```

This would create a `foobar` folder in the image's working folder and copy all the contents from the URI.

Building the application

In this step, we make sure to create the final artifacts that make up our executable legacy application. Often, this is a `JAR` or `WAR` file, with or without some satellite JARs. This part of the Dockerfile should mimic the way you traditionally used to build an application before containerizing it. Thus, if you're using Maven as your build automation tool, the corresponding snippet of the Dockerfile could look as simple as this:

```
RUN mvn --clean install
```

In this step, we may also want to list the environment variables the application uses and provide sensible defaults. But never provide default values for environment variables that provide secrets to the application, such as the database connection string! Use the `ENV` keyword to define your variables, like this:

```
ENV foo=bar
ENV baz=123
```

Also, declare all ports that the application is listening on and that need to be accessible from outside of the container via the `EXPOSE` keyword, like this:

```
EXPOSE 5000
EXPOSE 15672/tcp
```

Next, we will explain the `start` command.

Defining the start command

Usually, a Java application is started with a command such as `java -jar <mainapplication jar>` if it is a standalone application. If it is a WAR file, then the `start` command may look a bit different. Therefore, we can either define `ENTRYPOINT` or `CMD` to use this command. Thus, the final statement in our Dockerfile could look like this:

```
ENTRYPOINT java -jar pet-shop.war
```

Often, though, this is too simplistic, and we need to execute a few pre-run tasks. In this case, we can craft a script file that contains the series of commands that need to be executed to prepare the environment and run the application. Such a file is often called `docker-entrypoint.sh`, but you are free to name it however you want. Make sure the file is executable – for example, run the following command on the host:

```
chmod +x ./docker-entrypoint.sh
```

The last line of the Dockerfile would then look like this:

```
ENTRYPOINT ./docker-entrypoint.sh
```

Now that you have been given hints on how to containerize a legacy application, it is time to recap and ask ourselves, is it worth the effort?

Why bother?

At this point, I can see you scratching your head and asking yourself: Why bother? Why should you take on this seemingly huge effort just to containerize a legacy application? What are the benefits?

It turns out that the **return on investment** (**ROI**) is huge. Enterprise customers of Docker have publicly disclosed at conferences such as DockerCon 2018 and 2019 that they are seeing these two main benefits of Dockerizing traditional applications:

- More than a 50% saving in maintenance costs

- Up to a 90% reduction in the time between the deployments of new releases

The costs saved by reducing the maintenance overhead can be directly reinvested and used to develop new features and products. The time saved during new releases of traditional applications makes a business more agile and able to react to changing customer or market needs more quickly.

Now that we have discussed how to build Docker images at length, it is time to learn how we can ship those images through the various stages of the software delivery pipeline.

Sharing or shipping images

To be able to ship our custom image to other environments, we need to give it a globally unique name. This action is often called **tagging an image**. We then need to publish the image to a central location from which other interested or entitled parties can pull it. These central locations are called **image registries**.

In the following sections, we will describe how this works in more detail.

Tagging an image

Each image has a so-called **tag**. A tag is often used to version images, but it has a broader reach than just being a version number. If we do not explicitly specify a tag when working with images, then Docker automatically assumes we're referring to the *latest* tag. This is relevant when pulling an image from Docker Hub, as shown in the following example:

```
$ docker image pull alpine
```

The preceding command will pull the `alpine:latest` image from Docker Hub. If we want to explicitly specify a tag, we can do so like this:

```
$ docker image pull alpine:3.21
```

This will pull the Alpine image that has been tagged with `3.21`.

Demystifying image namespaces

So far, we have been pulling various images and haven't been worrying so much about where those images originated from. Your Docker environment is configured so that, by default, all images are pulled from Docker Hub. We also

only pulled so-called **official images** from Docker Hub, such as `alpine` or `busybox`.

Now, it is time to widen our horizons a bit and learn about how images are namespaced. The most generic way to define an image is by its fully qualified name, which looks as follows:

```
<registry URL>/<User or Org>/<name>:<tag>
```

Let's look at this in a bit more detail:

Namespace part	Description
`<registry URL>`	This is the URL to the registry from which we want to pull the image. By default, this is docker.io. More generally, this could be `https://registry.acme.com`. Other than Docker Hub, there are quite a few public registries out there that you could pull images from. The following is a list of some of them, in no particular order: Google, at `https://cloud.google.com/container-registry` Amazon AWS Amazon Elastic Container Registry (ECR), at `https://aws.amazon.com/ecr/` Microsoft Azure, at `https://azure.microsoft.com/en-us/services/container-registry/` Red Hat, at `https://access.redhat.com/containers/` Artifactory, at

Namespace part	Description
`<User or Org>`	This is the private Docker ID of either an individual or an organization defined on Docker Hub – or any other registry, for that matter, such as Microsoft or Oracle.
`<name>`	This is the name of the image, which is often also called a repository.
`<tag>`	This is the tag of the image.

Table 4.1: Docker image namespace elements

Let's look at an example, as follows:

```
https://registry.acme.com/engineering/web-app:1.0
```

Here, we have an image, `web-app`, that is tagged with version `1.0` and belongs to the `engineering` organization on the private registry at `https://registry.acme.com`.

Now, there are some special conventions:

- If we omit the registry URL, then Docker Hub is automatically taken
- If we omit the tag, then the `latest` tag is taken
- If it is an official image on Docker Hub, then no user or organization namespace is needed

Here are a few samples in tabular form:

Image	Description
`alpine`	The official `alpine` image on Docker Hub with the `latest` tag.
`ubuntu:22.04`	The official `ubuntu` image on Docker Hub with the `22.04` tag or version.

Image	Description
`hashicorp/vault`	The `vault` image of an organization called `hashicorp` on Docker Hub with the `latest` tag.
`acme/web-api:12.0`	The `web-api` image version of `12.0` that's associated with the `acme` org. The image is on Docker Hub.
`gcr.io/jdoe/sample-app:1.1`	The `sample-app` image with the `1.1` tag, belonging to an individual with the `jdoe` ID on Google's container registry.

Table 4.2: Examples of valid Docker image names

Now that we know how the fully qualified name of a Docker image is defined and what its parts are, let's talk about some special images we can find on Docker Hub.

Explaining official images

In the preceding table, we mentioned "*official image*" a few times. This needs an explanation.

Images are stored in repositories on the Docker Hub registry. Official repositories are a set of repositories hosted on Docker Hub that are curated by individuals or organizations that are also responsible for the software packaged inside the image. Let's look at an example of what that means. There is an official organization behind the Ubuntu Linux distro. This team also provides official versions of Docker images that contain their Ubuntu distros.

Official images are meant to provide essential base OS repositories, images for popular programming language runtimes, frequently used data storage, and other important services.

Docker sponsors a team whose task is to review and publish all those curated images in public repositories on Docker Hub. Furthermore, Docker scans all official images for vulnerabilities.

Pushing images to a registry

Creating custom images is all well and good, but at some point, we want to share or ship our images to a target environment, such as a test, **quality assurance** (**QA**), or production system. For this, we typically use a container registry. One of the most popular public registries out there is Docker Hub. It is configured as a default registry in your Docker environment, and it is the registry from which we have pulled all our images so far.

On a registry, we can usually create personal or organizational accounts. For example, the author's account at Docker Hub is `gnschenker`. Personal accounts are good for personal use. If we want to use the registry professionally, then we'll probably want to create an organizational account, such as `acme`, on Docker Hub. The advantage of the latter is that organizations can have multiple teams. Teams can have differing permissions.

To be able to push an image to my account on Docker Hub, I need to tag it accordingly. Let's say I want to push the latest version of the Alpine image to my account and give it a tag of `1.0`. I can do this in the following way:

1. Tag the existing image, `alpine:latest`, with this command:

   ```
   $ docker image tag alpine:latest gnschenker/alpine:1.0
   ```

   Here, Docker does not create a new image but creates a new reference to the existing image, `alpine:latest`, and names it `gnschenker/alpine:1.0`.

2. Now, to be able to push the image, I have to log in to my account, as follows:

   ```
   $ docker login -u gnschenker -p <my secret password>
   ```

3. Make sure to replace `gnschenker` with your own Docker Hub username and `<my secret password>` with your password.
4. After a successful login, I can then push the image, like this:

   ```
   $ docker image push gnschenker/alpine:1.0
   ```

5. I will see something similar to this in the Terminal window:

   ```
   The push refers to repository [docker.io/gnschenker/alpine]
   04a094fe844e: Mounted from library/alpine
   1.0: digest: sha256:5cb04fce... size: 528
   ```

For each image that we push to Docker Hub, we automatically create a repository. A repository can be private or public. Everyone can pull an image from a public repository. From a private repository, an image can only be pulled if you are logged in to the registry and have the necessary permissions configured.

Supply chain security practices

Supply chain security is a critical aspect of managing Docker images, especially in environments where security compliance and resilience against vulnerabilities are mandatory. Implementing robust supply chain security practices ensures that container images remain trustworthy and free from vulnerabilities throughout their lifecycle.

Key practices to enhance supply chain security include the following:

- **Using official and verified images**: Always prefer official Docker images or those verified by trusted vendors. These images are regularly updated and scanned for vulnerabilities.
- **Image scanning**: Regularly scan container images using vulnerability scanning tools such as Trivy, Clair, or Docker Scout. These tools help detect known vulnerabilities and suggest remediation strategies.
- **Image signing and verification**: Use digital signatures (such as Docker Content Trust, implemented via Notary) to ensure the integrity and authenticity of your images. Signed images help verify that the image has not been tampered with and originates from a trusted source.
- **Least privilege principle**: Run containers using minimal permissions necessary. Avoid running containers as the root user to reduce potential attack surfaces.
- **Regular updates and patches**: Continuously monitor and update images to integrate the latest security patches and fixes provided by the image maintainers.

By incorporating these practices into our containerization workflow, we can significantly reduce the risk of security breaches and ensure our containerized applications run securely in production. We will dive into more details in *Chapter 8*, *Docker Tips and Tricks*.

Summary

In this chapter, we explored the fundamental concepts and practices of creating and managing Docker container images. We started by examining the layered architecture of Docker images and the crucial role of graph drivers (also known as storage drivers). We saw how these graph drivers, particularly the widely recommended overlay2, efficiently merge multiple immutable layers into a unified root filesystem used by containers.

Next, we walked through practical approaches to building container images, including interactive image creation, Dockerfile-driven builds, and methods for saving and loading images. Additionally, we introduced a pragmatic "lift and shift" approach to help modernize legacy applications, outlining a structured process involving dependency analysis, configuration management, and secure handling of secrets.

Finally, we highlighted the importance of supply chain security practices, emphasizing key strategies for securing Docker images. We discussed best practices such as using official and verified images, regularly scanning for vulnerabilities, employing image signing to ensure authenticity, applying the principle of least privilege to containers, and consistently integrating security updates and patches.

By following these guidelines, we reinforced how containerized applications can be securely and efficiently prepared for robust production deployments.

In the next chapter, we will delve into managing data in Docker containers, focusing on data volumes, environment variables, and the use of configuration files. We will learn practical approaches to persist data and configure applications securely and effectively.

Questions

Please try to answer the following questions to assess your learning progress:

1. What is the primary function of Docker graph drivers?
2. What is a Dockerfile used for?
3. How can you create a Docker image interactively?
4. What are two important benefits of using multi-stage builds in Dockerfiles?

5. What are three best practices for securing the Docker image supply chain?
6. How would you create a Dockerfile that inherits from Ubuntu version 25.04, and that installs `ping` and runs `ping` when a container starts? The default address used to ping should be `127.0.0.1`.
7. How would you create a new container image that uses `alpine:latest` as a base image and installs `curl` on top of it? Name the new image `my-alpine:1.0`.
8. Create a Dockerfile that uses multiple steps to create an image of a Hello World app of minimal size, written in C or Go.
9. Name three essential characteristics of a Docker container image.
10. What command can you use to export a Docker image as a tarball?
11. You want to push an image named `foo:1.0` to your `jdoe` personal account on Docker Hub. Which of the following is the right solution?
    a. `$ docker container push foo:1.0`
    b. `$ docker image tag foo:1.0 jdoe/foo:1.0`
    c. `$ docker image push jdoe/foo:1.0`
    d. `$ docker login -u jdoe -p <your password>`
    e. `$ docker image tag foo:1.0 jdoe/foo:1.0`
    f. `$ docker image push jdoe/foo:1.0`
    g. `$ docker login -u jdoe -p <your password>`
    h. `$ docker container tag foo:1.0 jdoe/foo:1.0`
    i. `$ docker container push jdoe/foo:1.0`
    j. `$ docker login -u jdoe -p <your password>`
    k. `$ docker image push foo:1.0 jdoe/foo:1.0`

Answers

Here are possible answers to this chapter's questions:

1. Graph drivers merge multiple image layers into a single, coherent root filesystem used by containers.
2. A Dockerfile is used as a declarative method to define and build Docker container images consistently and repeatably.

3. You can create a Docker image interactively by running a container from an existing base image, making manual changes, and then committing those changes into a new image using the `docker commit` command.
4. Multi-stage builds help create significantly smaller container images by including only essential runtime components and improve security by minimizing the attack surface.
5. Three best practices to secure the Docker image supply chain are:
    a. Regularly scanning images for vulnerabilities.
    b. Using official or verified images from trusted sources.
    c. Implementing image signing and verification to ensure authenticity.
6. The Dockerfile could look like this:

```
FROM ubuntu:25.04
RUN apt-get update && \
    apt-get install -y iputils-ping
ENTRYPOINT ["ping"]
CMD ["127.0.0.1"]
```

   Note that in Ubuntu, the `ping` tool is part of the `iputils-ping` package. You can build the image called pinger – for example – with the following command:

```
$ docker image build -t mypinger .
```

7. The Dockerfile could look like this:

```
FROM alpine:latest
RUN apk update && \
    apk add curl
```

   Build the image with the following command:

```
$ docker image build -t my-alpine:1.0 .
```

8. The Dockerfile for a Go application could look like this:

```
FROM golang:1.23 AS builder
WORKDIR /app
# Disable modules so no go.mod is needed
ENV GO111MODULE=off
COPY main.go .
```

```
RUN CGO_ENABLED=0 GOOS=linux \
    go build -ldflags="-s -w" -o hello
FROM scratch
COPY --from=builder /app/hello /hello
ENTRYPOINT ["/hello"]
```

You can find the full solution in the `~/The-Ultimate-Docker-Container-Book-Fourth-Edition/chapter-04/solutions/answer-08` folder.

9. A Docker image has the following characteristics:
    - It is immutable
    - It consists of one-to-many layers
    - It contains the files and folders needed for the packaged application to run
10. Use the `docker image save` command to export an image as a tarball, for example: `docker image save -o image.tar <image-name>`.
11. The correct answer is C. First, you need to log in to Docker Hub; then, you must tag your image correctly with the username. Finally, you must push the image.

5

Data Volumes and Configuration

Join our book community on Discord:

https://packt.link/mqfS2

In the previous chapter, we learned how to build and share our container images. Focus was placed on how to build images that are as small as possible by only containing artifacts that are needed by the containerized application.

In this chapter, we are going to learn how we can work with stateful containers – that is, containers that consume and produce data. We will also learn how to configure our containers at runtime and at image build time, using environment variables and config files.

Here is a list of the topics we're going to discuss:

- Creating and mounting data volumes
- Sharing data between containers
- Using host volumes
- Defining volumes in images
- Configuring containers
- Persistent storage and stateful container patterns

After working through this chapter, you will be able to do the following:

- Create, delete, and list data volumes
- Mount an existing data volume into a container
- Create durable data from within a container using a data volume
- Share data between multiple containers using data volumes
- Mount any host folder into a container using data volumes
- Define the access mode (read/write or read-only) for a container when accessing data in a data volume
- Configure environment variables for applications running in a container

- Parameterize a Dockerfile by using build arguments

Technical requirements

For this chapter, you need to have Docker Desktop installed on your machine. There is no code accompanying this chapter.

Before we start, we need to create a folder for *Chapter 5* inside our code repository:

1. Use this command to navigate to the folder where you checked out the code from GitHub:

```
$ cd ~/The-Ultimate-Docker-Container-Book-v4
```

NOTE

Note

If you did not check out the GitHub repository at the default location, the preceding command may vary for you.

1. Create a sub-folder for *Chapter 5* and navigate to it:

```
$ mkdir chapter-05 && cd chapter-05
```

Let's get started!

Creating and mounting data volumes

All meaningful applications consume or produce data. Yet containers are, preferably, meant to be stateless. How are we going to deal with this? One way is to use Docker volumes. Volumes allow containers to consume, produce, and modify a state. Volumes have a life cycle that goes beyond the life cycle of containers. When a container that uses a volume dies, the volume continues to exist. This is great for the durability of the state.

Modifying the container layer

Before we dive into volumes, let's first discuss what happens if an application in a container changes something in the filesystem of the container. In this case, the changes are all happening in the writable container layer that we

introduced in *Chapter 4*, *Creating and Managing Container Images*. Let's quickly demonstrate this:

1. Run a container and execute a script in it that creates a new file, like this:

```
$ docker container run --name demo alpine \
  /bin/sh -c 'echo "This is a test" > sample.txt'
```

2. The preceding command creates a container named `demo`, and, inside this container, creates a file called `sample.txt` with the content `This is a test`. The container exits after running the `echo` command but remains in memory, available for us to do our investigations.
3. Let's use the `diff` command to find out what has changed in the container's filesystem concerning the filesystem of the original image, as follows:

```
$ docker container diff demo
```

   The output should look like this:

```
A /sample.txt
```

4. A new file, as indicated by the letter `A`, has been added to the filesystem of the container, as expected. Since all layers that stem from the underlying image (Alpine, in this case) are immutable, the change could only happen in the writeable container layer.

Files that have changed compared to the original image will be marked with a `C`, and those that have been deleted with a `D`.

Now, if we remove the container from memory, its container layer will also be removed, and with it, all the changes will be irreversibly deleted. If we need our changes to persist even beyond the lifetime of the container, this is not a solution. Luckily, we have better options in the form of Docker volumes. Let's get to know them.

Creating volumes

When using Docker Desktop on a macOS or Windows computer, containers are not running natively on macOS or Windows but rather in a (hidden) VM created by Docker Desktop.

To demonstrate how and where the underlying data structures are created in the respective filesystem (MacOS or Windows), we need to be a bit creative. If, on the other hand, we are doing the same on a Linux computer, things are straightforward.

Let's start with a simple exercise to create a volume:

1. Open a new Terminal window and type in this command:

   ```
   $ docker volume create sample
   ```

   You should get this response:

   ```
   sample
   ```

   Here, the name of the created volume will be the output.

   The default volume driver is the so-called **local driver**, which stores the data locally in the host filesystem.

2. The easiest way to find out where the data is stored on the host is by using the `docker volume inspect` command on the volume we just created. The actual location can differ from system to system, so this is the safest way to find the target folder. So, let's use this command:

   ```
   $ docker volume inspect sample
   ```

   We should see something like this:

   ```
   ❯ docker volume inspect sample
   [
       {
           "CreatedAt": "2025-04-27T08:30:36Z",
           "Driver": "local",
           "Labels": null,
           "Mountpoint": "/var/lib/docker/volumes/sample/_data",
           "Name": "sample",
           "Options": null,
           "Scope": "local"
       }
   ]
   ```

   Figure 5.1: Inspecting the Docker volume called sample

   The host folder can be found in the output under `Mountpoint`. In our case, the folder is `/var/lib/docker/volumes/sample/_data`.

3. Alternatively, we can create a volume using the dashboard of Docker Desktop:
    a. Open the dashboard of Docker Desktop.
    b. On the left-hand side, select the **Volumes** tab.
    c. Click the blue button, as shown in the following screenshot:

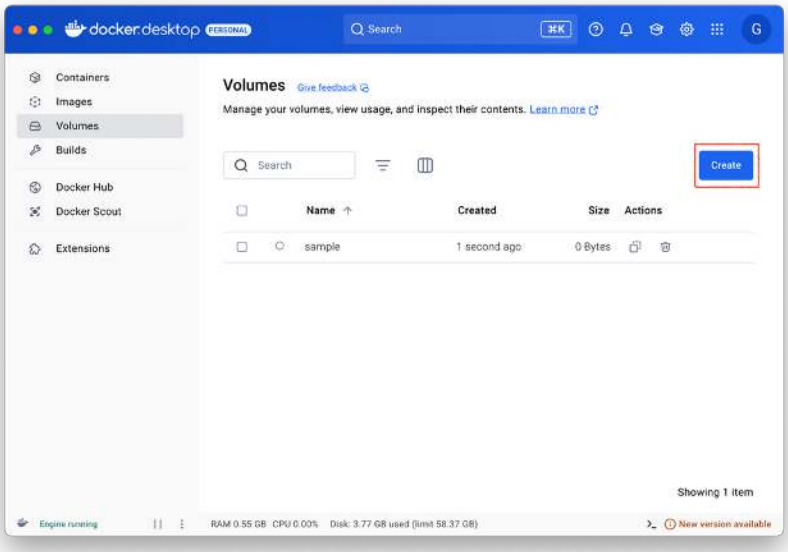


Figure 5.2: Creating a new Docker volume with Docker for Desktop

    d. Then type `sample-2` as the name for the new volume into the textbox of the **New Volume** popup and click **Create**. You should now see this:

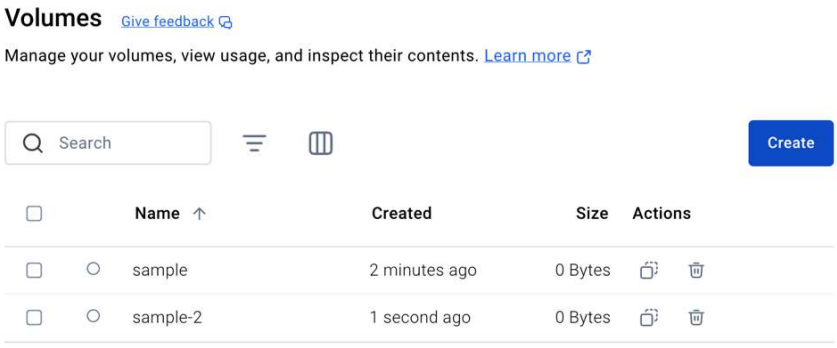


Figure 5.3: List of Docker volumes shown in Docker Desktop

There are other volume drivers available from third parties, in the form of plugins. We can use the `--driver` parameter in the `create` command to select a different volume driver.

Other volume drivers use different types of storage systems to back a volume, such as cloud storage, **Network File System** (**NFS**) drives, software-defined storage, and more. The discussion of the correct usage of other volume drivers is beyond the scope of this book, though.

Nevertheless, here is a list of popular volume drivers, including their manufacturer and main usage:

Driver	**Manufacturer**	**Description and main usage**
`local`	Docker	Default driver storing data on the local host filesystem. Suitable for single-host deployments and development environments.
`nfs`	Docker	Utilizes **Network File System** (**NFS**) for sharing volumes across multiple hosts. Ideal for distributed systems requiring shared storage.
`cifs`	Docker	Employs the **Common Internet File System** (**CIFS**) protocol to mount Samba shares, facilitating integration with Windows-based storage systems.
`rclone`	Rclone	Enables mounting of various cloud storage services (for example, Google Drive, Dropbox) as volumes, allowing containers to interact with cloud-based filesystems.

Driver	Manufacturer	Description and main usage
`rexray/ebs`	REX-Ray	Integrates with Amazon **Elastic Block Store** (**EBS**) to provide persistent block storage for containers in AWS environments.
`flocker`	ClusterHQ	Manages data volumes for Docker containers, facilitating data migration between hosts in a cluster.
`portworx`	Portworx	Offers high-performance, container-granular storage solutions with features such as replication, snapshots, and encryption, suitable for enterprise environments.
`glusterfx`	Gluster	Provides scalable network filesystem capabilities, allowing containers to access shared storage across multiple nodes.
`azurefile`	Microsoft Azure	Connects containers to Azure File Storage, enabling persistent storage for applications running in Azure.

Driver	Manufacturer	Description and main usage
`gce-pd`	Google Cloud	Integrates Google Compute Engine Persistent Disks with Docker, offering durable block storage for containers in GCP.

Table 5.1: Popular Docker volume drivers

These drivers extend Docker's storage capabilities, allowing for flexible and scalable data management across various environments and infrastructures.

Interestingly, AWS – the biggest cloud provider – does not offer a specific Docker volume driver itself, but it supports the use of Docker volume drivers through Amazon **Elastic Container Service** (**ECS**) when using the EC2 launch type. We can use the built-in local driver or third-party volume drivers such as REX-Ray, Portworx, or others to manage persistent storage with Docker volumes.

Mounting a volume

Once we have created a named volume, we can mount it into a container by following these steps:

1. For this, we can use the `--volume` or `-v` parameter in the `docker container run` command, like this:

```
$ docker container run --name test -it \
     -v sample:/data \
     alpine /bin/sh
```

   If you are working in a clean Docker environment, then the output produced by this command should look similar to this:

```
Unable to find image 'alpine:latest' locally latest:
Pulling from library/alpine
050382585609: Pull complete
Digest: sha256: 8914eb54f968791faf6a86...
Status: Downloaded newer image for alpine:latest
/ #
```

Otherwise, you should just see the prompt of the Bourne shell running inside the Alpine container:

```
/ #
```

The preceding command mounts the sample volume to the `/data` folder inside the container.

2. Inside the container, we can now create files in the `/data` folder, as follows:

```
/ # cd /data
/data # echo "Some data" > data.txt
/data # echo "Some more data" > data2.txt
```

3. If we were to navigate to the host folder that contains the data of the volume and list its content, we should see the two files we just created inside the container. But this is a bit more involved, so long as we are working on a Mac or Windows computer, and will be explained in detail in the *Accessing Docker volumes* section. Stay tuned.
4. Exit the tool container by pressing *Ctrl* + *D*.
5. Now, let's delete the dangling `test` container:

```
$ docker container rm test
```

6. Next, we must run another one based on CentOS. This time, we are even mounting our volume to a different container folder, `/app/data`, like this:

```
$ docker container run --name test2 -it --rm \
    -v sample:/app/data \
    centos:7 /bin/bash
```

You should see an output similar to this:

```
❯ docker container run --name test2 -it --rm \
    -v sample:/app/data \
    centos:7 /bin/bash

Unable to find image 'centos:7' locally
7: Pulling from library/centos
6717b8ec66cd: Pull complete
Digest: sha256:be65f488b7764ad3638f236b7b515b3678369a5124c47b8d32916d6487418ea4
Status: Downloaded newer image for centos:7
[root@74c99e6664ca /]#
```

Figure 5.4: Mounting the sample volume into a CentOS 7 container

The last line of the preceding output indicates that we are at the prompt of the Bash shell running inside the CentOS container.

7. Once inside the CentOS container, we can navigate to the `/app/data` folder to which we have mounted the volume and list its content. We should see the following:

```
[root@74c99e6664ca /]# cd app/data/
[root@74c99e6664ca data]# ls -al
total 16
drwxr-xr-x 2 root root 4096 Apr 27 09:08 .
drwxr-xr-x 3 root root 4096 Apr 27 09:10 ..
-rw-r--r-- 1 root root   10 Apr 27 09:08 data.txt
-rw-r--r-- 1 root root   15 Apr 27 09:08 data2.txt
[root@74c99e6664ca data]#
```

Figure 5.5: Listing the files of the sample volume inside the CentOS container

   This is the definitive proof that data in a Docker volume persists beyond the lifetime of a container, as well as that volumes can be reused by other, even different, containers from the one that used it first.

   It is important to note that the folder inside the container to which we mount a Docker volume is excluded from the Union filesystem. That is, each change inside this folder and any of its subfolders will not be part of the container layer but will be persisted in the backing storage provided by the volume driver. This fact is really important since the container layer is deleted when the corresponding container is stopped and removed from the system.

8. Exit the CentOS container with *Ctrl* + *D*.

Great – we have learned how to mount Docker volumes into a container! Next, we will learn how to delete existing volumes from our system.

Removing volumes

Volumes can be removed using the `docker volume rm` command. It is important to remember that removing a volume destroys the containing data irreversibly, and thus is to be considered *a dangerous command*. Docker helps us a bit in this regard, as it does not allow us to delete a volume that is still in use by a container. Always make sure, before you remove or delete a volume,

that you either have a backup of its data or you don't need this data anymore. Let's learn how to remove volumes by following these steps:

1. The following command deletes the sample volume that we created earlier:

```
$ docker volume rm sample
```

   In case you get an error saying that the volume is still in use, please make sure that both `test` and `test2` containers have been removed.

2. After executing the preceding command, double-check that the folder on the host has been deleted. You can use this command to list all volumes defined on your system:

```
$ docker volume ls
```

   Make sure the `sample` volume has been deleted.

3. Now, as an exercise, also remove the `sample-2` volume from your system.
4. To remove all running containers to clean up the system, run the following command:

```
$ docker container rm -v -f $(docker container ls -aq)
```

5. Note that by using the `-v` or `--volume` flag in the command you use to remove a container, you can ask the system to also remove any anonymous volume associated with that particular container. Of course, that will only work if the particular volume is only used by this container.

In the next section, we will show you how to access the backing folder of a volume when working with Docker Desktop.

Accessing Docker volumes

Now, let's, for a moment, assume that we are on a Mac with macOS. This operating system is not based on Linux but on a different Unix flavor. Let's see if we can find the data structure for the `sample` and `sample-2` volumes, where the `docker volume inspect` command told us so:

1. First, let's create two named Docker volumes, either using the command line or doing the same via the dashboard of Docker for Desktop:

```
$ docker volume create sample
$ docker volume create sample-2
```

2. In your Terminal, try to navigate to that folder:

```
$ cd /var/lib/docker/volumes/sample/_data
```

On the author's MacBook Air, this is the response to the preceding command:

Figure 5.6: Accessing the volumes directory on a Mac

This was expected since Docker is not running natively on Mac but inside a slim VM, as mentioned earlier in this chapter.

Similarly, if you are using a Windows machine, you won't find the data where the `inspect` command indicated.

It turns out that on a Mac, the data for the VM that Docker creates can be found in the `~/Library/Containers/com.docker.docker/Data/vms/0` folder.

To access this data, we need to somehow get into this VM. On a Mac, we have two options to do so. The first is to use the `terminal screen` command. However, this is very specific to macOS, and thus, we will not discuss it here. The second option is to get access to the filesystem of Docker Mac via the special `nsenter` command, which should be executed inside a Linux container such as Debian. This also works on Windows, and thus, we will show the steps needed using this second option.

3. To run a container that can inspect the underlying host filesystem on your system, use this command:

```
$ docker container run -it --privileged --pid=host \
    debian nsenter -t 1 -m -u -n -i sh
```

When running the container, we execute the following command inside the container:

```
nsenter -t 1 -m -u -n -i sh
```

If that sounds complicated to you, don't worry; you will understand more as we proceed through this book. If there is one takeaway, then it is to realize how powerful the right use of containers can be.

4. From within this container, we can now list all the volumes that are defined with `/ # ls -l /var/lib/docker/volumes`. What we get should look similar to this:

```
/ # nsenter -t 1 -m -u -n -i sh
/ # ls -l /var/lib/docker/volumes
total 92
brw-------    1 root     root      254,   1 Apr 27 09:06 backingFsBlockDev
-rw-------    1 root     root        131072 Apr 27 09:20 metadata.db
drwx-----x    3 root     root          4096 Apr 27 09:20 sample
drwx-----x    3 root     root          4096 Apr 27 09:20 sample-2
/ #
```

Figure 5.7: List of Docker volumes via nsenter

5. Next, navigate to the folder representing the mount point of the volume:

```
/ # cd /var/lib/docker/volumes/sample/_data
```

6. Then list its content, as follows:

```
/var/lib/docker/volumes/sample/_data # ls -l
```

This should output the following:

```
/ # cd /var/lib/docker/volumes/sample/_data
/var/lib/docker/volumes/sample/_data # ls -al
total 8
drwxr-xr-x    2 root     root          4096 Apr 27 09:20 .
drwx-----x    3 root     root          4096 Apr 27 09:20 ..
/var/lib/docker/volumes/sample/_data #
```

Figure 5.8: List files in volume sample

The folder is currently empty since we have not yet stored any data in the volume.

7. Similarly, for our `sample-2` volume, we can use the following command:

```
/ # cd /var/lib/docker/volumes/sample-2/_data
/var/lib/docker/volumes/sample-2/ # ls -l
```

Again, this should output a similar result, indicating that the folder is currently empty.

8. Next, let's generate two files with data in the `sample` volume from within an Alpine container. First, open a new Terminal window, since the other one is blocked by our `nsenter` session.
9. To run the container and mount the `sample` volume to the `/data` folder of the container, use the following code:

```
$ docker container run --rm -it \
    -v sample:/data alpine /bin/sh
```

10. Generate two files in the `/data` folder inside the container, like this:

```
/ # echo "Hello world" > /data/sample.txt
/ # echo "Other message" > /data/other.txt
```

11. Exit the Alpine container by pressing *Ctrl* + *D*.
12. Back in the `nsenter` session, try to list the content of the sample volume again using this command:

```
/ # cd /var/lib/docker/volumes/sample/_data
/ # ls -al
```

This time, you should see this:

```
~ # cd /var/lib/docker/volumes/sample/_data
/var/lib/docker/volumes/sample/_data # ls -al
total 16
drwxr-xr-x    2 root     root          4096 Apr 27 09:30 .
drwx-----x    3 root     root          4096 Apr 27 09:20 ..
-rw-r--r--    1 root     root            14 Apr 27 09:30 other.txt
-rw-r--r--    1 root     root            12 Apr 27 09:30 sample.txt
/var/lib/docker/volumes/sample/_data #
```

Figure 5.9: Volume sample containing the two files created in the Alpine container

This indicates that we have data written to the filesystem of the host.

13. Let's try to create a file from within this special container, and then list the contents of the folder, as follows:

```
/ # echo "I love Docker" > docker.txt
```

14. Now, let's see what we got:

```
/ # ls -l
```

This gives us something like this:

Figure 5.10: Volume containing file generated directly on the host

1. Let's see if we can see this new file from within a container mounting the sample volume. From within a new Terminal window, run this command:

```
$ docker container run --rm \
    -v sample:/data \
    centos:7 ls -l /data
```

   This should output this:

```
❯ docker container run --rm \
    -v sample:/data \
    centos:7 ls -l /data

total 12
-rw-r--r-- 1 root root 14 Apr 27 09:34 docker.txt
-rw-r--r-- 1 root root 14 Apr 27 09:30 other.txt
-rw-r--r-- 1 root root 12 Apr 27 09:30 sample.txt
```

Figure 5.11: List of files as observed from within the Alpine container

   The preceding output is showing us that we can add content directly to the host folder backing the volume and then access it from a container that has the volume mounted.

2. To exit our special privileged container with the `nsenter` tool, we can just press *Ctrl* + *D* twice.

We have now created data using two different methods, as follows:

- From within a container that has a sample volume mounted
- Using a special privileged folder to access the hidden VM used by Docker for Desktop, and directly writing into the backing folder of the sample volume

In the next section, we will learn how to share data between containers.

Sharing data between containers

Containers are like sandboxes for the applications running inside them. This is mostly beneficial and wanted, to protect applications running in different containers from each other. It also means that the whole filesystem visible to an application running inside a container is private to this application, and no other application running in a different container can interfere with it.

At times, though, we want to share data between containers. Say an application running in **container A** produces some data that will be consumed by another application running in **container B**. How can we achieve this? Well, I'm sure you've already guessed it – we can use Docker volumes for this purpose. We can create a volume and mount it to container A, as well as to container B. In this way, both applications A and B have access to the same data.

Now, as always, when multiple applications or processes concurrently access data, we have to be very careful to avoid inconsistencies. To avoid concurrency problems such as race conditions, we should ideally have only one application or process that is creating or modifying data, while all other processes concurrently accessing this data only read it.

NOTE

Race conditions

A race condition is a situation that can occur in computer programming when the output of a program or process is affected by the order and timing of events in ways that are unpredictable or unexpected. In a race condition, two or more parts of a program are trying to access or modify the same data or resource simultaneously, and the outcome depends on the timing of these events. This can result in incorrect or inconsistent output, errors, or crashes.

We can enforce a process running in a container to only be able to read the data in a volume by mounting this volume as read-only. Here's how we can do this:

1. Execute the following command:

```
$ docker container run -it --name writer \
    -v shared-data:/data \
    alpine /bin/sh
```

Here, we are creating a container called `writer` that has a volume, `shared-data`, mounted in default read/write mode.

2. Try to create a file inside this container, like this:

```
# / echo "I can create a file" > /data/sample.txt
```

It should succeed.

3. Exit this container by pressing *Ctrl* + *D* or typing `exit` and hitting the *Enter* key at the prompt.
4. Then, execute the following command:

```
$ docker container run -it --name reader \
    -v shared-data:/app/data:ro \
    ubuntu:25.04 /bin/bash
```

Here, we have a container called `reader` that has the same volume mounted as **read-only** (**ro**).

5. First, make sure you can see the file created in the first container, like this:

```
$ ls -l /app/data
```

This should give you something like this:

```
❯ docker container run -it --name reader \
    -v shared-data:/app/data:ro \
    ubuntu:25.04 /bin/bash

Unable to find image 'ubuntu:25.04' locally
25.04: Pulling from library/ubuntu
36dff27e34eb: Already exists
Digest: sha256:79efa276fdefa2ee3911db29b0608f8c0561c347ec3f4d4139980d43b168d991
Status: Downloaded newer image for ubuntu:25.04
root@6e014003e628:/# ls -l /app/data
total 4
-rw-r--r-- 1 root root 20 Apr 27 09:42 sample.txt
root@6e014003e628:/#
```

Figure 5.12: Listing files of a read-only volume

6. Then, try to create a file, like this:

```
#/ echo "Try to break read/only" > /app/data/data.txt
```

It will fail with the following message:

```
bash: /app/data/data.txt: Read-only file system
```

This is expected since the volume was mounted as read-only.

7. Let's exit the container by typing `exit` at the command prompt. Back on the host, let's clean up all containers and volumes, as follows:

```
$ docker container rm -f $(docker container ls -aq)
$ docker volume rm $(docker volume ls -q)
```

Exercise: Analyze the preceding commands carefully and try to understand what exactly they do and how they work.

Next, we will show you how to mount arbitrary folders from the Docker host into a container.

Using host volumes

In certain scenarios, such as when developing new containerized applications or when a containerized application needs to consume data from a certain folder produced, say, by a legacy application, it is very useful to use volumes that mount a specific host folder. Let's look at the following example:

```
$ docker container run --rm -it \
    -v $(pwd)/src:/app/src \
    alpine:latest /bin/sh
```

The preceding expression interactively starts an Alpine container with a shell and mounts the `src` subfolder of the current directory into the container at `/app/src`. We need to use `$(pwd)` (or `pwd`, for that matter), which is the current directory, as when working with volumes, we always need to use absolute paths.

The first time you' execute the preceding command, macOS will ask you for permission:

Figure 5.13: The operating system asking for permission to access files in a protected folder

Hit **Allow** and proceed.

Developers use these techniques all the time when they are working on an application that runs in a container and want to make sure that the container always contains the latest changes they make to the code, without the need to rebuild the image and rerun the container after each change.

Let's make a sample to demonstrate how that works. Let's say we want to create a simple static website while using Nginx as our web server, as follows:

1. First, let's create a new subfolder on the host. The best place to do this is inside the chapter folder we created at the beginning of the chapter. There, we will put our web assets, such as HTML, CSS, and JavaScript files. Use this command to create the subfolder and navigate to it:

```
$ cd ~/The-Ultimate-Docker-Container-Book-v4
$ cd chapter-05
$ mkdir my-web && cd my-web
```

2. Then, create a simple web page, like this:

```
$ echo "<h1>Personal Website</h1>" > index.html
```

3. Now, add a Dockerfile that will contain instructions on how to build the image containing our sample website. Add a file called `Dockerfile` to the folder, with this content:

```
FROM nginx:alpine
COPY . /usr/share/nginx/html
```

   The Dockerfile starts with the latest Alpine version of Nginx and then copies all files from the current host directory into the `/usr/share/nginx/html` containers folder. This is where Nginx expects web assets to be located.

4. Now, let's build the image with the following command:

```
$ docker image build -t my-website:1.0 .
```

   Please do not forget the period (.) at the end of the preceding command. The output of this command will look similar to this:

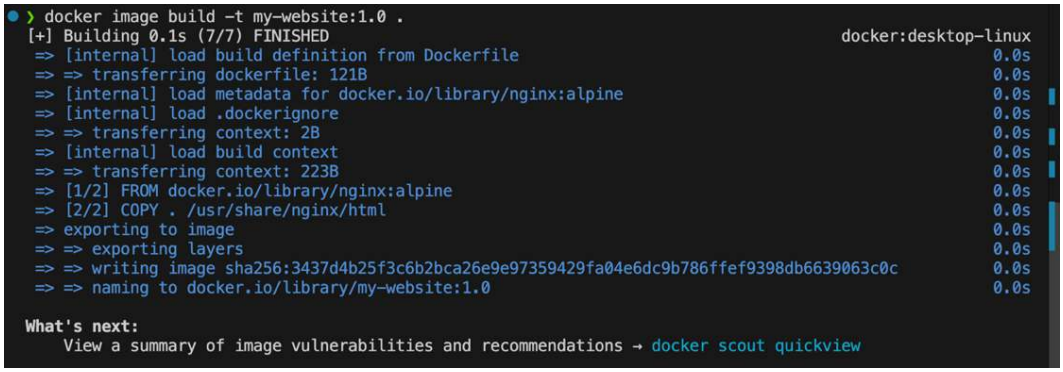


Figure 5.14: Building a Docker image for a sample Nginx web server

5. Finally, we will run a container from this image. We will run the container in detached mode, like this:

```
$ docker container run -d \
    --name my-site \
    -p 8080:80 \
    my-website:1.0
```

Note the `-p 8080:80` parameter. We haven't discussed this yet, but we will do so in detail in *Chapter 10*, *Single-Host Networking*. At the moment, just know that this maps the container port `80` on which Nginx is listening for incoming requests to port `8080` of your laptop, where you can then access the application.

6. Now, open a browser tab and navigate to `http://localhost:8080/index.html`; you should see your website, which currently consists only of a title, **Personal Website**.
7. Now, edit the `index.html` file in your favorite editor so that it looks like this:

```
<h1>Personal Website - Version 2</h1>
<p>This is some text</p>
```

8. Now, save it, and then refresh the browser. Oh! That didn't work. The browser still displays the previous version of the `index.html` file, which consists only of the title. So, let's stop and remove the current container, then rebuild the image and rerun the container, as follows:

```
$ docker container rm -f my-site && \
  docker image build -t my-website:1.0 . && \
  docker container run -d \
    --name my-site \
    -p 8080:80 \
    my-website:1.0
```

9. Refresh the browser again. This time, the new content should be shown. Well, it worked, but there is way too much friction involved. Imagine you have to do this every time you make a simple change to your website. That's not sustainable.
10. Now is the time to use host-mounted volumes. Once again, remove the current container and rerun it with the volume mount, like this:

```
$ docker container rm -f my-site
$ docker container run -d \
    --name my-site \
    -v $(pwd):/usr/share/nginx/html \
    -p 8080:80 \
    my-website:1.0
```

NOTE

Note

If you are working on Windows, a pop-up window will be displayed that says Docker wants to access the hard drive, and that you have to click on the **Share access** button.

1. Now, append some more content to the `index.html` file and save it. Then, refresh your browser. You should see the changes. This is exactly what we wanted to achieve; we also call this an edit-and-continue experience. You can make as many changes in your web files and always immediately see the result in the browser, without having to rebuild the image and restart the container containing your website.
2. When you're done playing with your web server and wish to clean up your system, remove the container with the following command:

```
$ docker container rm -f my-site
```

It is important to note that the updates are now propagated bi-directionally. If you make changes on the host, they will be propagated to the container, and vice versa. It's also important to note that when you mount the current folder into the container target folder, `/usr/share/nginx/html`, the content that is already there is replaced by the content of the host folder.

In the next section, we will learn how to define volumes used in a Docker image.

Defining volumes in images

If we go back to what we learned about containers in *Chapter 4*, *Creating and Managing Container Images*, we will recall this: the filesystem of each container, when started, is made up of the immutable layers of the underlying image, plus a writable container layer specific to this very container. All changes that the processes running inside the container make to the filesystem will be persisted in this container layer. Once the container is stopped and removed from the system, the corresponding container layer is deleted from the system and irreversibly lost.

Some applications, such as databases running in containers, need to persist their data beyond the lifetime of the container. In this case, they can use

volumes. To make things a bit more explicit, let's look at a concrete example. MongoDB is a popular open source document database. Many developers use MongoDB as a storage service for their applications. To support this, the maintainers of MongoDB have created an image and published it on Docker Hub, which can be used to run an instance of the database in a container. This database will produce data that needs to be persisted long-term, but the MongoDB maintainers do not know who uses this image and how it is used. So, they can't influence the `docker container run` command with which the users of the database will start this container. So, how can they define volumes?

Luckily, there is a way of defining volumes in the Dockerfile. The keyword to do so is `VOLUME`, and we can either add the absolute path to a single folder or a comma-separated list of paths. These paths represent the folders of the container's filesystem. Let's look at a few samples of such volume definitions, as follows:

```
VOLUME /app/data
VOLUME /app/data, /app/profiles, /app/config
VOLUME ["/app/data", "/app/profiles", "/app/config"]
```

The first line in the preceding snippet defines a single volume to be mounted at `/app/data`. The second line defines three volumes as a comma-separated list. The last one defines the same as the second line, but this time, the value is formatted as a JSON array.

When a container is started, Docker automatically creates a volume and mounts it to the corresponding target folder of the container for each path defined in the Dockerfile. Since each volume is created automatically by Docker, it will have SHA-256 as its ID.

At container runtime, the folders defined as volumes in the Dockerfile are excluded from the Union filesystem, and thus any changes in those folders do not change the container layer but are persisted to the respective volume. It is now the responsibility of the operations engineers to make sure that the backing storage of the volumes is properly backed up.

We can use the `docker image inspect` command to get information about the volumes defined in the Dockerfile. Let's see what MongoDB gives us by following these steps:

1. First, we will pull the image with the following command:

```
$ docker image pull mongo:8.0.8
```

You should see this:

```
❯ docker image pull mongo:8.0.8
8.0.8: Pulling from library/mongo
49b96e96358d: Pull complete
647de49b3f54: Pull complete
268841dcd611: Pull complete
638631464f16: Pull complete
7e63d955562b: Pull complete
c2096b08e1e2: Pull complete
9d88392cb05a: Pull complete
d04becf8fed0: Pull complete
Digest: sha256:cc62438c8ef61ce02f89b4f7c026e735df4580e8cd8857980d12e0eae73bf044
Status: Downloaded newer image for mongo:8.0.8
docker.io/library/mongo:8.0.8
```

Figure 5.15: Pulling the latest MongoDB image from Docker Hub

2. Then, we will inspect this image and use the `--format` parameter to only extract the essential part from the massive amount of data, as follows:

```
$ docker image inspect \
    --format='{{json .Config.Volumes}}' \
    mongo:8.0.8 | jq .
```

Note `| jq .` at the end of the command. We are piping the output of `docker image inspect` into the `jq` tool, which nicely formats the output.

NOTE

Tip

If you haven't installed `jq` yet on your system, you can do so with `brew install jq` on macOS or `choco install jq` on Windows.

The preceding command will return the following result:

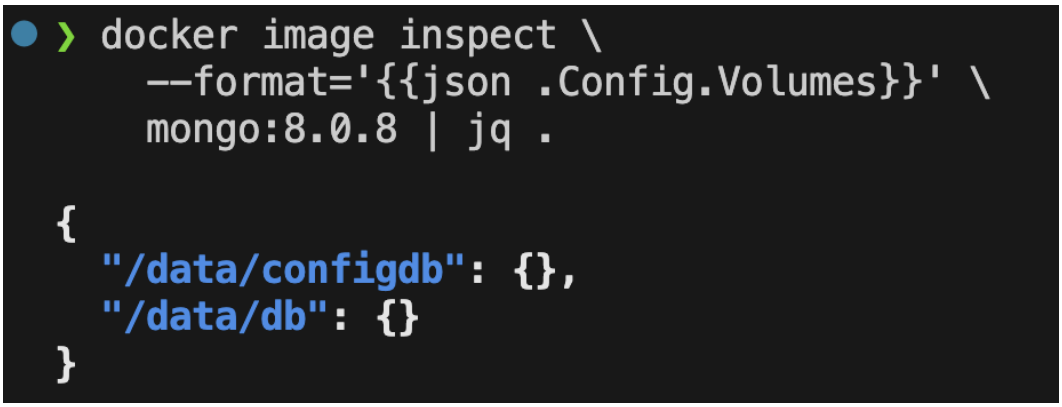


Figure 5.16: Volumes section of the MongoDB configuration

As we can see, the Dockerfile for MongoDB defines two volumes at `/data/configdb` and `/data/db`.

1. Now, let's run an instance of MongoDB in the background as a daemon, as follows:

```
$ docker run --name my-mongo -d mongo:8.0.8
```

2. We can now use the `docker container inspect` command to get information about the volumes that have been created, among other things. Use this command to just get the volume information:

```
$ docker inspect --format '{{json .Mounts}}' \
    my-mongo | jq .
```

The preceding command should output something like this:

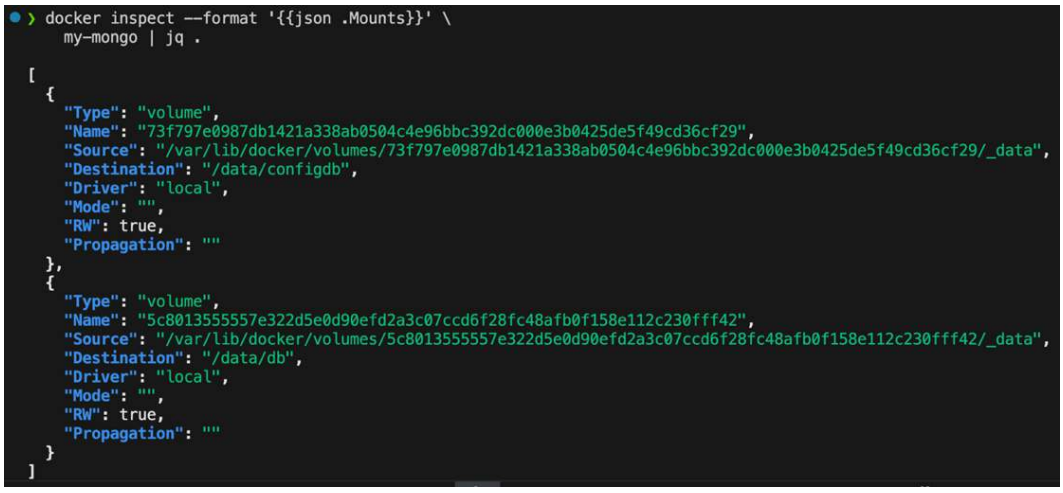


Figure 5.17: Inspecting the MongoDB volumes

The `Source` field gives us the path to the host directory, where the data produced by MongoDB inside the container will be stored. This way, your backup operators will know which folders to back up in production.

Before you leave, clean up the MongoDB container with the following command:

```
$ docker rm -f my-mongo
```

That's it for the moment concerning volumes. In the next section, we will explore how we can configure applications running in containers and the container image build process itself.

Configuring containers

More often than not, we need to provide some configuration to the application running inside a container. The configuration is often used to allow the same container to run in very different environments, such as in development, test, staging, or production environments. In Linux, configuration values are often provided via environment variables.

We have learned that an application running inside a container is completely shielded from its host environment. Thus, the environment variables that we see on the host are different from the ones that we see within a container.

Let's prove this by looking at what is defined on our host:

1. Use this command to display a list of all environment variables defined for your Terminal session:

   ```
   $ export
   ```

   On the author's macOS, they see something like this (shortened):

   ```
   ...
   COLORTERM=truecolor
   COMMAND_MODE=unix2003
   ...
   HOME=/Users/gabriel
   HOMEBREW_CELLAR=/opt/homebrew/Cellar
   HOMEBREW_PREFIX=/opt/homebrew
   HOMEBREW_REPOSITORY=/opt/homebrew
   INFOPATH=/opt/homebrew/share/info:/opt/homebrew/...:
   LANG=en_GB.UTF-8
   ```

```
LESS=-R
LOGNAME=gabriel
...
```

2. Next, let's run a shell inside an Alpine container:

    a. Run the container with this command:

    ```
    $ docker container run --rm -it alpine /bin/sh
    ```

    b. Just as a reminder, we are using the `--rm` command-line parameter so that we do not have to remove the dangling container once we stop it.

    c. Then, list the environment variables we can see there with this command:

    ```
    / # export
    ```

    This should produce the following output:

    ```
    ❯ docker container run --rm -it alpine /bin/sh
    / # export
    export HOME='/root'
    export HOSTNAME='ab3e439e1249'
    export PATH='/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'
    export PWD='/'
    export SHLVL='1'
    export TERM='xterm'
    / #
    ```

    Figure 5.18: Environment variable inside an Alpine container

    The preceding output is different than what we saw directly on the host. It is more proof that a container offers a sandboxed environment, distinct from the host environment to the user.

3. Hit *Ctrl* + *D* to leave and stop the Alpine container.

Next, let's define environment variables for containers.

Defining environment variables for containers

Now, the good thing is that we can pass some configuration values into the container at start time. We can use the `--env` (or the short form, `-e`) parameter in the form of `--env <key>=<value>` to do so, where `<key>` is the name of the environment variable and `<value>` represents the value to be associated with that variable. Let's assume we want the app that is to be run in our container

to have access to an environment variable called `LOG_DIR`, with a value of `/var/log/my-log`. We can do so with this command:

```
$ docker container run --rm -it \
    --env LOG_DIR=/var/log/my-log \
    alpine /bin/sh
```

The preceding code starts a shell in an Alpine container and defines the requested environment inside the running container. To prove that this is true, we can execute this command inside the Alpine container:

```
/ # export | grep LOG_DIR
```

The output should be as follows:

```
export LOG_DIR='/var/log/my-log'
```

The output looks as expected. We now have the requested environment variable with the correct value available inside the container. We can, of course, define more than just one environment variable when we run a container. We just need to repeat the `--env` (or `-e`) parameter. Have a look at this sample:

```
$ docker container run --rm -it \
    --env LOG_DIR=/var/log/my-log \
    --env MAX_LOG_FILES=5 \
    --env MAX_LOG_SIZE=1G \
    alpine /bin/sh
```

After running the preceding command, we are left at the command prompt inside the Alpine container:

```
/ #
```

Let's list the environment variables with the following command:

```
/ # export | grep LOG
```

We will see the following:

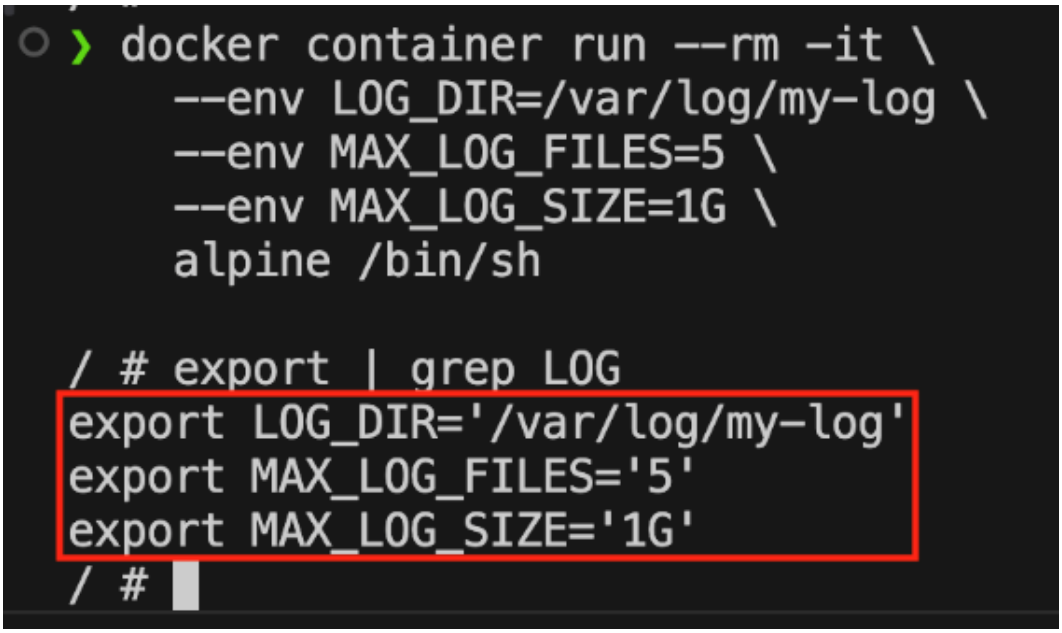


Figure 5.19: Environment variables defined via the --env parameter

Now, let's look at situations where we have many environment variables to configure.

Using configuration files

Complex applications can have many environment variables to configure, and thus, our command to run the corresponding container can quickly become unwieldy. For this purpose, Docker allows us to pass a collection of environment variable definitions as a file. We have the `--env-file` parameter in the `docker container run` command for this purpose.

Let's try this out, as follows:

1. Navigate to the source folder for `chapter 5` that we created at the beginning of this chapter:

```
$ cd ~/The-Ultimate-Docker-Container-Book-v4
$ cd chapter-05
```

2. Create a subfolder, `config-file`, and navigate to it, like this:

```
$ mkdir config-file && cd config-file
```

3. Use your favorite editor to create a file called `development.config` in this folder. Add the following content to the file and save it, as follows:

```
LOG_DIR=/var/log/my-log
MAX_LOG_FILES=5
MAX_LOG_SIZE=1G
```

Notice how we have the definition of a single environment variable per line in `<key>=<value>` format, where, once again, `<key>` is the name of the environment variable, and `<value>` represents the value to be associated with that variable.

4. Now, from within the `config-file` subfolder, let's run an Alpine container, pass the file as an environment file, and run the `export` command inside the container to verify that the variables listed inside the file have indeed been created as environment variables inside the container, like this:

```
$ docker container run --rm -it \
    --env-file ./development.config \
    alpine sh -c "export | grep LOG"
```

Indeed, the variables are defined, as we can see in the output generated:

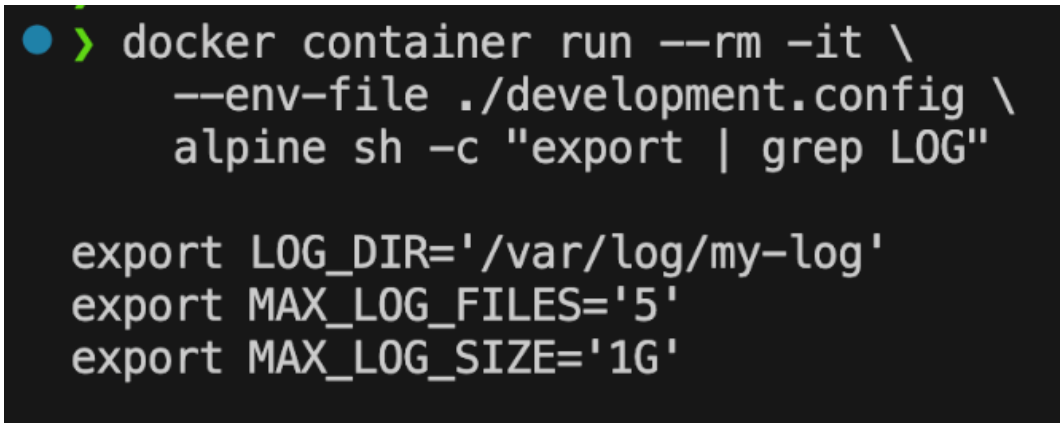


Figure 5.20: Using a file to define environment variables

This is exactly what we expected.

Next, let's look at how to define default values for environment variables that are valid for all container instances of a given Docker image.

Defining environment variables in container images

Sometimes, we want to define some default value for an environment variable that must be present in each container instance of a given container image. We can do so in the Dockerfile that is used to create that image by following these steps:

1. Navigate to the source folder for `chapter 5` that we created at the beginning of this chapter:

```
$ cd ~/The-Ultimate-Docker-Container-Book-v4
$ cd chapter-05
```

2. Create a subfolder called `config-in-image` and navigate to it, like this:

```
$ mkdir config-in-image && cd config-in-image
```

3. Use your favorite editor to create a file called `Dockerfile` in the `config-in-image` subfolder. Add the following content to the file and save it:

```
FROM alpine:latest
ENV LOG_DIR=/var/log/my-log
ENV MAX_LOG_FILES=5
ENV MAX_LOG_SIZE=1G
```

4. Create a container image called `my-alpine` using the preceding Dockerfile, as follows:

```
$ docker image build -t my-alpine .
```

NOTE

Note

Don't forget the period at the end of the preceding line!

1. Run a container instance from this image that outputs the environment variables defined inside the container, like this:

```
$ docker container run --rm -it \
    my-alpine sh -c "export | grep LOG"
```

You should see the following in your output:

```
❯ docker container run --rm -it \
    my-alpine sh -c "export | grep LOG"

export LOG_DIR='/var/log/my-log'
export MAX_LOG_FILES='5'
export MAX_LOG_SIZE='1G'
```

Figure 5.21: Environment variables as defined in Docker image

This is exactly what we expected.

2. The good thing, though, is that we are not stuck with those variable values at all. We can override one or many of them by using the `--env` parameter in the `docker container run` command. Use this command:

```
$ docker container run --rm -it \
    --env MAX_LOG_SIZE=2G \
    --env MAX_LOG_FILES=10 \
    my-alpine sh -c "export | grep LOG"
```

3. Now, have a look at the following command and its output:

```
❯ docker container run --rm -it \
    --env MAX_LOG_SIZE=2G \
    --env MAX_LOG_FILES=10 \
    my-alpine sh -c "export | grep LOG"

export LOG_DIR='/var/log/my-log'
export MAX_LOG_FILES='10'
export MAX_LOG_SIZE='2G'
```

Figure 5.22: Overridden environment variables

4. We can also override default values by using environment files together with the `--env-file` parameter in the `docker container run` command. Please try it out for yourself.

In the next section, we are going to introduce environment variables that are used at the build time of a Docker image.

Environment variables at build time

Sometimes, we want to be able to define some environment variables that are valid at the time we build a container image. Imagine that you want to define a `BASE_IMAGE_VERSION` environment variable that shall then be used as a parameter in your Dockerfile. Imagine the following Dockerfile:

```
ARG BASE_IMAGE_VERSION=12.7-stretch
FROM node:${BASE_IMAGE_VERSION}
WORKDIR /app
COPY packages.json .
RUN npm install
COPY . .
CMD npm start
```

We are using the `ARG` keyword to define a default value that is used each time we build an image from the preceding Dockerfile. In this case, that means that our image uses the `node:12.7-stretch` base image.

Now, if we want to create a special image for, say, testing purposes, we can override this variable at image build time using the `--build-arg` parameter, as follows:

```
$ docker image build \
    --build-arg BASE_IMAGE_VERSION=12.7-alpine \
    -t my-node-app-test .
```

In this case, the resulting `my-node-app-test:latest` image will be built from the `node:12.7-alpine` base image and not from the `node:12.7-stretch` default image.

To summarize, environment variables defined via `--env` or `--env-file` are valid at container runtime. Variables defined with `ARG` in the Dockerfile or `--build-arg` in the `docker container build` command are valid at container image build time. The former is used to configure an application running inside a container, while the latter is used to parameterize the container image build process.

In the next and last section of this chapter, we will explore the topic of persistent storage and stateful container patterns.

Persistent storage and stateful container patterns

Containers are inherently ephemeral; they are designed to be stateless and easily replaceable. However, many real-world applications require the ability to persist data beyond the lifecycle of a single container instance. This necessitates the implementation of persistent storage solutions and patterns that support stateful behavior within containerized environments.

Understanding persistent storage in Docker

In Docker, persistent storage is achieved through volumes and bind mounts:

- **Volumes**: Managed by Docker, volumes are stored in a part of the host filesystem that is managed by Docker (`/var/lib/docker/volumes/` on Linux). They are the preferred mechanism for persisting data generated by and used by Docker containers.
- **Bind mounts**: These mount a file or directory from the host filesystem into the container. While they offer more control, they are dependent on the directory structure and OS of the host machine.

Volumes are generally recommended over bind mounts due to their portability and management features, though bind mounts are often used by software engineers during the development process to dynamically mount code into their application container.

Patterns for managing stateful containers

Managing stateful applications in containers involves specific patterns to ensure data persistence and consistency:

- **Data volume containers**: An older pattern where a container is dedicated solely to holding volumes to be shared with other containers. This pattern has largely been replaced by named volumes.
- **Named volumes**: Creating and managing volumes independently of containers allows for better data persistence and sharing across multiple containers.
- **StatefulSets (in Kubernetes)**: For orchestrated environments – as we will discuss in section 3 of this book, `StatefulSets` manage the deployment

and scaling of a set of Pods, and provide guarantees about the ordering and uniqueness of these Pods. Each Pod gets its own persistent volume.

- **Volume plugins**: Docker supports volume plugins that allow volumes to be stored on remote hosts or cloud providers, enabling data persistence across different environments.

Best practices for persistent storage

We recommend the following best practices when dealing with persistent storage:

- **Use volumes for persistence**: Prefer Docker-managed volumes over bind mounts for better portability and management
- **Backup and restore**: Implement regular backup strategies for volumes to prevent data loss
- **Monitor storage usage**: Keep an eye on storage consumption to avoid running out of space, which can cause containers to fail
- **Security considerations**: Ensure that sensitive data stored in volumes is properly secured, using appropriate permissions and, if necessary, encryption
- **Use volume drivers**: Leverage volume drivers for integrating with external storage systems, providing flexibility and scalability

By adhering to these practices and understanding the patterns for managing stateful containers, we can effectively handle persistent data in Docker environments, ensuring data durability and application reliability.

With this, we have come to the end of this chapter.

Summary

In this chapter, we have explored the essential concepts related to Docker data volumes and configuration, highlighting their critical role in containerized applications. You learned how to effectively create, mount, and manage Docker volumes to ensure data persistence across container lifecycles. We discussed the practical approaches to sharing data between containers and the host system, providing you with the skills necessary to implement robust container solutions.

You also mastered container configuration techniques, such as the use of environment variables and configuration files. These powerful mechanisms allow applications to be flexible and adaptable across different environments, enhancing maintainability and consistency.

Lastly, we introduced the topic of persistent storage and stateful container patterns. In this section, you learned about the nuances of maintaining application state and data integrity beyond the ephemeral lifespan of containers. We examined the critical differences between volumes and bind mounts, emphasizing why Docker-managed volumes are generally preferable. You discovered key patterns for managing stateful containers, such as named volumes, data volume containers, and Kubernetes StatefulSets. Additionally, we discussed best practices to ensure secure, scalable, and reliable persistent storage in your Docker and orchestration environments.

Equipped with this knowledge, you're now ready to handle complex containerized applications that require persistent data storage and stateful management patterns, enabling you to build and deploy enterprise-grade applications confidently.

In the next chapter, we are going to introduce techniques commonly used to allow a developer to evolve, modify, debug, and test their code while running in a container.

Further reading

The following articles provide more in-depth information:

- *Use volumes*: `http://dockr.ly/2EUjTml`
- *Manage data in Docker*: `http://dockr.ly/2EhBpzD`
- *Docker volumes* on **Play with Docker** (**PWD**): `http://bit.ly/2sjIfDj`
- `nsenter`—Linux man page, at `https://bit.ly/2MEPG0n`
- *Set environment variables*: `https://docs.docker.com/reference/cli/docker/`
- *Understanding how ARG and FROM interact*: `https://dockr.ly/2OrhZgx`

Questions

Try to answer the following questions to assess your learning progress:

1. What is the primary difference between Docker volumes and bind mounts?
2. Why are volumes generally recommended over bind mounts for data persistence in Docker?
3. How does Docker ensure data persistence when a container is removed?
4. What is a common use case for bind mounts in Docker?
5. Can you share a volume between multiple containers? If so, how?
6. How would you create a named data volume with a name such as `my-products` using the default driver?
7. How would you run a container using the Alpine image and mount the `my-products` volume in read-only mode into the `/data` container folder?
8. How would you locate the folder that is associated with the `my-products` volume and navigate to it? Also, how would you create a file, `sample.txt`, with some content?
9. How would you run another Alpine container where you mount the `my-products` volume to the `/app-data` folder, in read/write mode? Inside this container, navigate to the `/app-data` folder and create a `hello.txt` file with some content.
10. How would you mount a host volume – for example, `~/my-project` – into a container?
11. How would you remove all unused volumes from your system?
12. How can you inspect the details of a Docker volume?
13. What are the implications of using bind mounts regarding security?
14. The list of environment variables that an application running in a container sees is the same as if the application were to run directly on the host.
    a. True
    b. False
15. Your application that shall run in a container needs a huge list of environment variables for configuration. What is the simplest method to run a container with your application and provide all this information to it?

Answers

Here are the answers to this chapter's questions:

1. Docker volumes are managed by Docker and stored in a part of the host filesystem that Docker manages (`/var/lib/docker/volumes/` on Linux). They are the preferred mechanism for persisting data. Bind mounts, on the other hand, mount a file or directory from the host filesystem into the container and rely on the host's directory structure, making them less portable.
2. Volumes are managed by Docker, offering better portability, easier backup and restore processes, and safer sharing among containers. They are less dependent on the host's directory structure and provide a more consistent environment across different systems.
3. By using volumes, Docker decouples the data from the container's lifecycle. Even if a container is removed, the data stored in a volume persists and can be attached to a new container.
4. Bind mounts are commonly used in development environments to mount source code or configuration files from the host into the container, allowing real-time code changes without rebuilding the image.
5. Yes, Docker volumes can be shared between multiple containers by specifying the same volume name in the -v or --mount flag when running each container. This allows containers to read from and write to the same data store.
6. To create a named volume, run the following command:

```
$ docker volume create my-products
```

7. Execute the following command:

```
$ docker container run -it --rm \
    -v my-products:/data:ro \
    alpine /bin/sh
```

8. To achieve this result, do this:
    a. To get the path on the host for the volume, use this command:

```
$ docker volume inspect my-products | grep Mountpoint
```

This should result in the following output:

```
    "Mountpoint": "/var/lib/docker/volumes/my-products/_data"
```

b. Now, execute the following command to run a container and execute `nsenter` within it:

```
$ docker container run -it --privileged --pid=host \
    debian nsenter -t 1 -m -u -n -i sh
```

c. Navigate to the folder containing the data for the `my-products` volume:

```
/ # cd /var/lib/docker/volumes/my-products/_data
```

d. Create a file containing the text `""I love Docker""` within this folder:

```
/ # echo "I love Docker" > sample.txt
```

e. Exit `nsenter` and its container by pressing *Ctrl* + *D*.

f. Execute the following command to verify that the file generated in the host filesystem is indeed part of the volume and accessible to the container to which we'll mount this volume:

```
$ docker container run --rm \
    --volume my-products:/data \
    alpine ls -l /data
```

The output of the preceding command should look similar to this:

```
total 4
-rw-r--r--    1 root     root    14 Dec  4 17:35 sample.txt
```

And indeed, we can see the file.

g. *Optional*: Run a modified version of the command to output the content of the `sample.txt` file.

9. Execute the following command:

```
$ docker run -it --rm -v my-products:/data:ro \
   alpine /bin/sh
```

```
/ # cd /data
/data # cat sample.txt
```

In another Terminal, execute this command:

```
$ docker run -it --rm -v my-products:/app-data \
   alpine /bin/sh
/ # cd /app-data
/app-data # echo "Hello other container" > hello.txt
/app-data # exit
```

10. Execute a command such as this:

```
$ docker container run -it --rm \
    -v $HOME/my-project:/app/data \
    alpine /bin/sh
```

11. Exit both containers and then, back on the host, execute this command:

```
$ docker volume prune
```

12. Use this command:

```
$ docker volume inspect my_volume
```

This provides detailed information about the volume, including its mount point and usage.

13. Bind mounts can pose security risks because they provide the container with access to the host's filesystem. If not properly managed, this can lead to unauthorized access or modification of host files. It's essential to set appropriate permissions and use read-only mounts when necessary.
14. The answer is **false** (B). Each container is a sandbox and thus has its very own environment.
15. Collect all environment variables and their respective values in a configuration file, which you then provide to the container with the `--env-file` command-line parameter in the `docker container run` command, like so:

```
$ docker container run --rm -it \
    --env-file ./development.config \
    alpine sh -c "export"
```

The Ultimate Docker Container Book