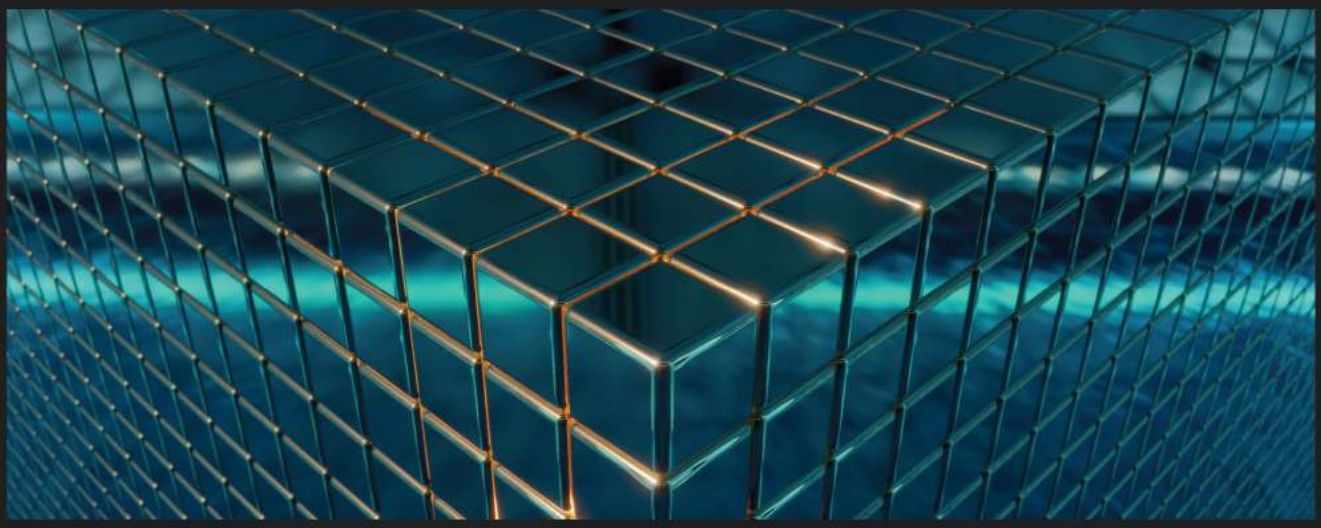
packt



3RD EDITION

The Ultimate Docker Container Book

Build, test, ship, and run containers with Docker and Kubernetes

DR. GABRIEL N. SCHENKER

1

What Are Containers and Why Should I Use Them?

This first chapter will introduce you to the world of containers and their orchestration. This book starts from the very beginning, in that it assumes that you have limited prior knowledge of containers, and will give you a very practical introduction to the topic.

In this chapter, we will focus on the software supply chain and the friction within it. Then, we'll present containers, which are used to reduce this friction and add enterprise-grade security on top of it. We'll also look into how containers and the ecosystem around them are assembled. We'll specifically point out the distinctions between the upstream **Open Source Software** (**OSS**) components, united under the code name Moby, that form the building blocks of the downstream products of Docker and other vendors.

The chapter covers the following topics:

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

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

- Explain what containers are, using an analogy such as physical containers, in a few simple sentences to an interested layperson
- Justify why containers are so important using an analogy such as physical containers versus traditional shipping, or apartment homes versus single-family homes, and so on, to an interested layperson

- Name at least four upstream open source components that are used by Docker products, such as Docker Desktop
- Draw a high-level sketch of the Docker container architecture

Let's get started!

What are containers?

A software container is a pretty abstract thing, so it might help to 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 transported 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 that they could move their carts along. 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 on 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 who want to have their goods transported from A to B package those goods into these containers. Then, they call a shipper, who uses 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.

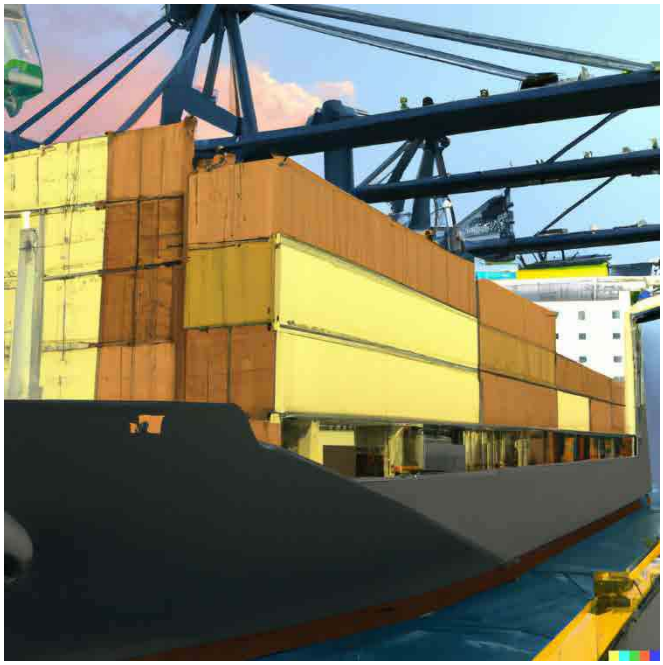


Figure 1.2 – Container ship being loaded in a port

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 when 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 which framework it was built on, what libraries it uses, and so on. Sometimes, two applications use the same framework but of different versions that might or might not be compatible with each other. Our operations engineers' lives became much harder over time. They had to become 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 as if you used a whole ship just to transport a single truckload of bananas in the transportation industry. What a waste! That would never be profitable. The ultimate solution to this problem was to provide something much more lightweight than VMs also able to perfectly encapsulate the goods it needed 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 package their applications, frameworks, and libraries into Docker containers, 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 cyber-attacks 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. But not only is customer data compromised – 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 not running in a container. Containers use Linux security primitives such as Linux kernel **namespaces** to sandbox different applications running on the same computers and **control groups** (**cgroups**) to avoid the noisy-neighbor problem, where one bad application uses all the available resources of a server and starves 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 to hand, we can run a full-blown relational database in a container as easily as saying 1, 2, 3. 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 they 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 for 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 used 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.

For 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 are 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 the optimal tools and support to build containerized applications.

Docker Desktop

Part of the Docker offering are products such as Docker Toolbox and Docker Desktop with its editions for Mac, 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 a Mac, Windows, or Linux machine.

> **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 Enterprise Edition

Docker EE – now owned by Mirantis – consists of the **Universal Control Plane** (**UCP**) and the **Docker Trusted Registry** (**DTR**), both of which run on top of Docker Swarm. Both are Swarm applications. Docker EE builds on top of the upstream components of the Moby project and adds enterprise-grade features such as **role-based access control** (**RBAC**), multi-tenancy, mixed clusters of Docker Swarm and Kubernetes, a web-based UI, and content trust, as well as image scanning on top.

> **Docker Swarm**
> Docker Swarm provides a powerful and flexible platform for deploying and managing containers in a production environment. It provides the tools and features you need to build, deploy, and manage your applications with ease and confidence.

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 that Docker has been installed on 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:

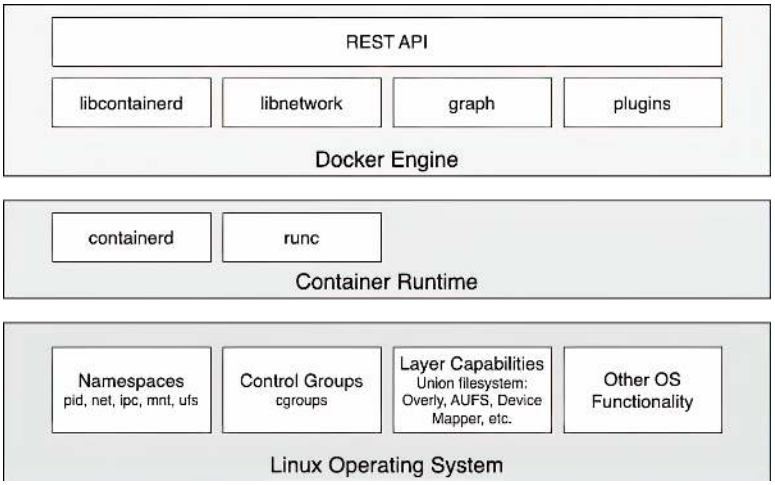


Figure 1.3 – High-level architecture diagram of Docker Engine

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. runc 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 that we will use often in this book is one of the consumers of this REST interface.

Summary

In this chapter, we looked at how containers can massively reduce friction in the software supply chain and, on top of that, make the supply chain much more secure. In the next chapter, we will familiarize ourselves with containers. We will learn how to run, stop, and remove containers and otherwise manipulate them. We will also get a pretty good overview of the anatomy of containers. For the first time, we are really going to get our hands dirty and play with these containers. So, 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*: `https://containerd.io/`
- *Getting Started with Docker Enterprise 3.1*: `https://www.mirantis.com/blog/getting-started-with-docker-enterprise-3-1/`

Questions

Please answer the following questions to assess your learning progress:

1. Which statements are correct (multiple answers are possible)?

   A. A container is kind of a lightweight VM

   B. A container only runs on a Linux host

   C. A container can only run one process

   D. The main process in a container always has PID 1

   E. A container is one or more processes encapsulated by Linux namespaces and restricted by cgroups

2. In your own words, using analogies, explain what a container is.
3. Why are containers considered to be a game-changer in IT? Name three or four reasons.
4. What does it mean when we claim, if a container runs on a given platform, then it runs anywhere? Name two to three reasons why this is true.
5. Is the following claim true or false: *Docker containers are only useful for modern greenfield applications based on microservices*? Please justify your answer.
6. How much does a typical enterprise save when containerizing its legacy applications?

   A. 20%

   B. 33%

   C. 50%

   D. 75%

7. Which two core concepts of Linux are containers based on?
8. On which operating systems is Docker Desktop available?

Answers

1. The correct answers are *D* and *E*.
2. A Docker container is to IT what a shipping container is to the transportation industry. It defines a standard on how to package goods. In this case, goods are the application(s) developers write. The suppliers (in this case, the developers) are responsible for packaging the goods into the container and making sure everything fits as expected. Once the goods are packaged into a container, it can be shipped. Since it is a standard container, the shippers can standardize their means of transportation, such as lorries, trains, or ships. The shipper does not really care what is in the container. Also, the loading and unloading process from one means of transportation

to another (for example, train to ship) can be highly standardized. This massively increases the efficiency of transportation. Analogous to this is an operations engineer in IT, who can take a software container built by a developer and ship it to a production system and run it there in a highly standardized way, without worrying about what is in the container. It will just work.

3. Some of the reasons why containers are game-changers are as follows:

   - Containers are self-contained and thus if they run on one system, they run anywhere that a Docker container can run.
   - Containers run on-premises and in the cloud, as well as in hybrid environments. This is important for today's typical enterprises since it allows a smooth transition from on-premises to the cloud.
   - Container images are built or packaged by the people who know best – the developers.
   - Container images are immutable, which is important for good release management.
   - Containers are enablers of a secure software supply chain based on encapsulation (using Linux namespaces and cgroups), secrets, content trust, and image vulnerability scanning.

4. A container runs on any system that can host containers. This is possible for the following reasons:

   - Containers are self-contained black boxes. They encapsulate not only an application but also all its dependencies, such as libraries and frameworks, configuration data, certificates, and so on.
   - Containers are based on widely accepted standards such as OCI.

5. The answer is false. Containers are useful for modern applications and to containerize traditional applications. The benefits for an enterprise when doing the latter are huge. Cost savings in the maintenance of legacy apps of 50% or more have been reported. The time between new releases of such legacy applications could be reduced by up to 90%. These numbers have been publicly reported by real enterprise customers.
6. 50% or more.
7. Containers are based on Linux **namespaces** (network, process, user, and so on) and **cgroups**. The former help isolate processes running on the same machine, while the latter are used to limit the resources a given process can access, such as memory or network bandwidth.
8. Docker Desktop is available for macOS, Windows, and Linux.

2

Setting Up a Working Environment

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:

- 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
- Installing Docker Toolbox
- Enabling Kubernetes on Docker Desktop
- Installing minikube
- Installing Kind

Technical requirements

For this chapter, you will need a laptop or a workstation with either macOS or Windows, preferably Windows 11, 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 18.04 or newer. I will try to indicate where commands and samples differ significantly from the ones on macOS or Windows.

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 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, Chocolatey is a good choice. 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:

```
$ /usr/bin/ruby -e "$(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 3.6.16
Homebrew/homebrew-core (git revision 025fe79713b; last
commit 2022-12-26)
Homebrew/homebrew-cask (git revision 15acb0b64a; last
commit 2022-12-26)
```

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 the instructions at `https://chocolatey.org/` or open a new PowerShell window in admin mode and execute the following command:

```
PS> Set-ExecutionPolicy Bypass -Scope Process -Force; iex
((New-Object System.Net.WebClient).DownloadString('https://
chocolatey.org/install.ps1'))
```

> **Note**
>
> It is important to run the preceding command as an administrator; otherwise, the installation will not succeed. It is also important to note that the preceding command is one single line and has only been broken into several lines here due to the limited line width.

Once Chocolatey has been installed, test it with the `choco --version` command. You should see output similar to the following:

```
PS> choco --version
0.10.15
```

To install an application such as the Vi editor, use the following command:

```
PS> 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**
> 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 a 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 Air, the output is as follows:

   ```
   git version 2.39.1
   ```

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`.

5. 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
```

This will clone the content of the main branch into your local folder, `~/The-Ultimate-Docker-Container-Book`. 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 just name 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 Containers*. 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 cask install 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
```

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

> **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.

4. 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
```

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

5. 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:

1. On your Debian or Ubuntu-based Linux machine, you can use Snap to install VS Code. Open a Bash Terminal and execute the following statement to install VS Code:

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

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

```
$ cd ~
```

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

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

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

5. 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 of the installed extensions.
3. To get a list of all installed extensions in your VS Code, use this command:

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

Next, let's install Docker Desktop.

Installing Docker Desktop on macOS or Windows

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. No matter what OS you're using, navigate to the Docker start page at `https://www.docker.com/get-started`:

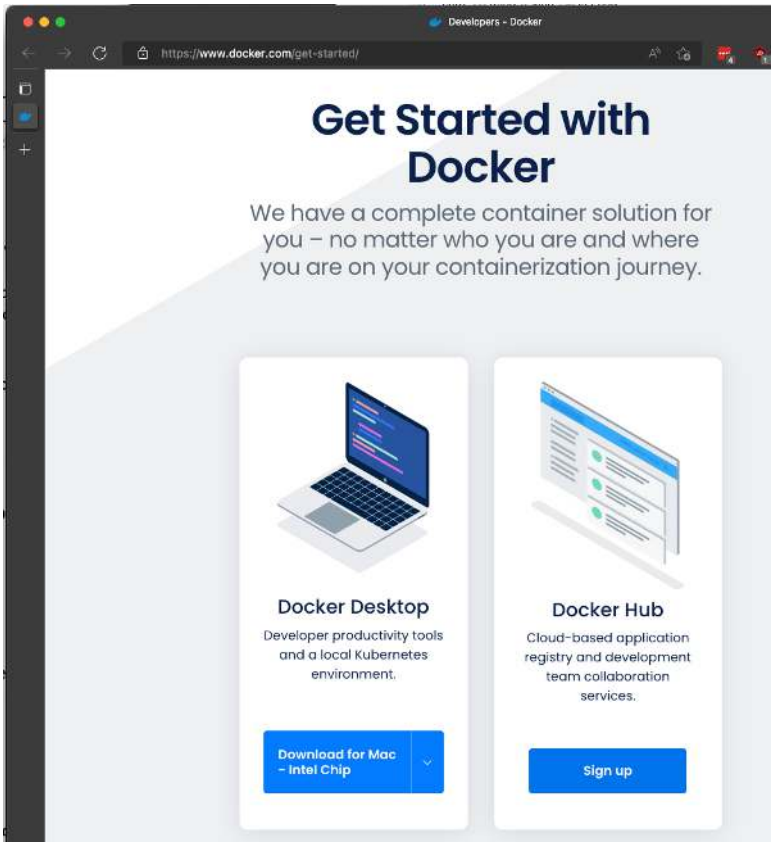


Figure 2.1 – Get Started with Docker

2. On the right-hand side of the view, you will find a blue **Sign up** button for Docker Hub. Click this button if you don't have an account on Docker Hub yet, then create one. It is free, but you need an account to download the software.

3. On the left-hand side of the view, you will find a blue button called **Download for <your OS>**, where **<your OS>** can be Linux, Mac, or Windows, depending on which OS you are working with. In the authors' case, it shows Mac as the target OS, but it got the CPU type wrong since the author is using a Mac with Apple's M1 chip.

   Click the small drop-down triangle on the right-hand side of the button to get the full 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.

4. 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:

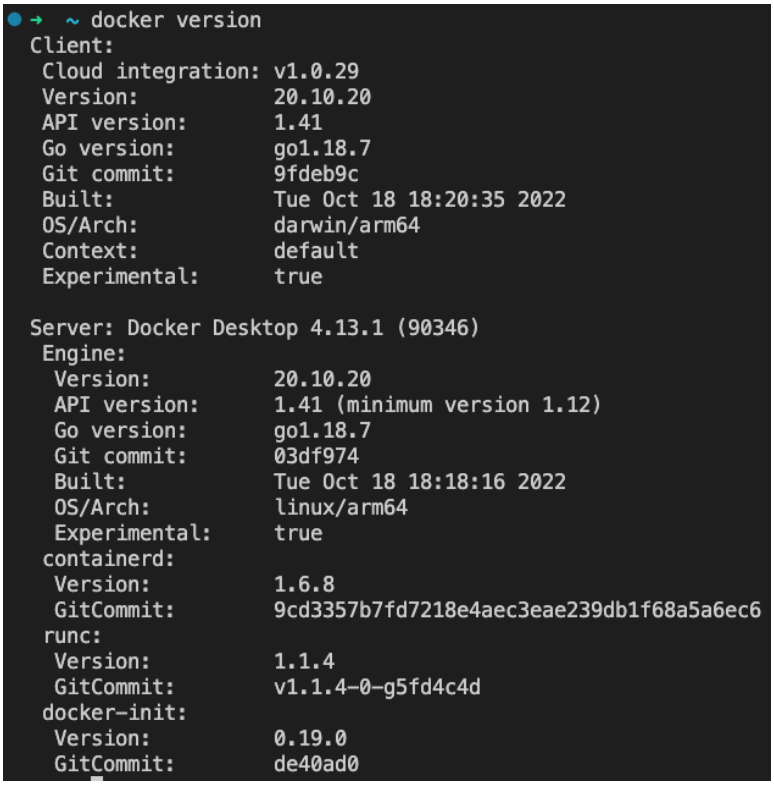


Figure 2.3 – Docker version of Docker Desktop

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 `20.10.21`.

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
7050e35b49f5: Pull complete
Digest: sha256:e18f0a777aefabe047a671ab3ec3eed05414477c951ab1a6f352a06974245fe7
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 authors' Mac, it looks like this:

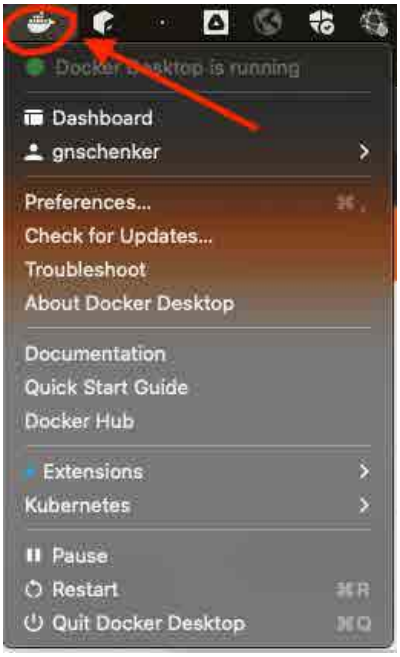


Figure 2.6 – Context menu for Docker Desktop

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

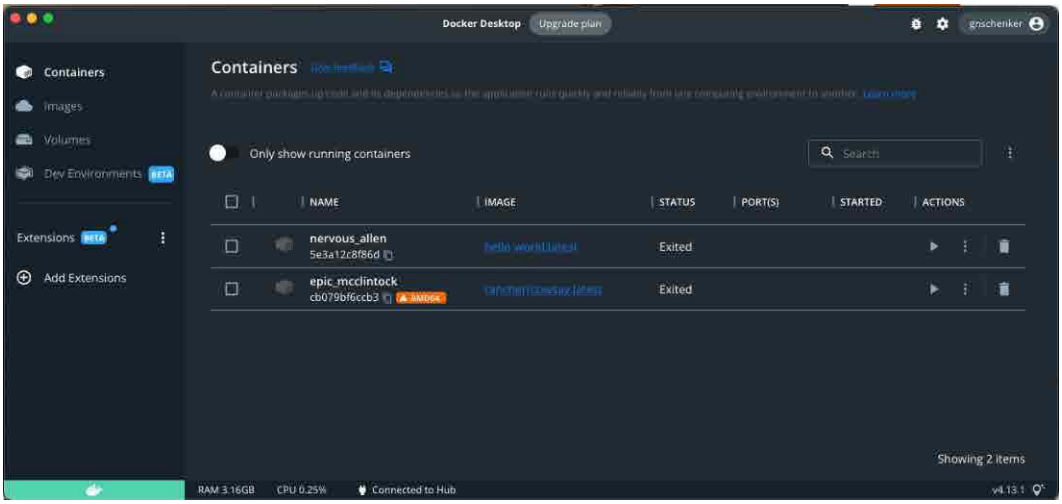


Figure 2.7 – Dashboard of Docker Desktop

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 this carefully, you will see that these are the containers that we previously created from the `hello-world` and `rancher/cowsay` Docker images. They both have a status of **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.

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

> **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 *Step 1*.

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

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**
> 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.

> **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 turn on:

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 check the **Enable Kubernetes** checkbox:

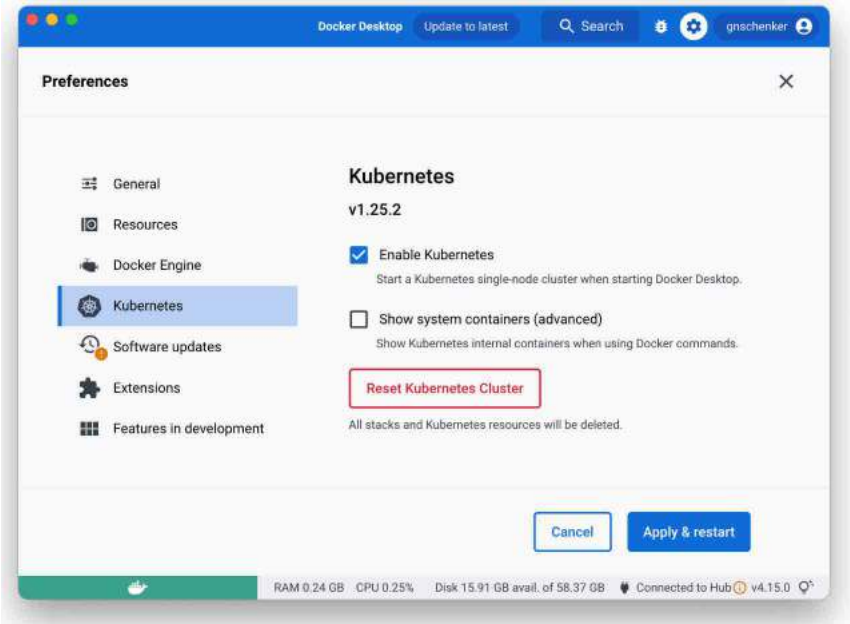


Figure 2.8 – Enabling Kubernetes on Docker Desktop

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

Now, you will have to be patient since Docker is downloading all the supporting infrastructure and then starting Kubernetes.

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

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 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.

Installing minikube on Linux, macOS, and Windows

To install minikube on 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 here:

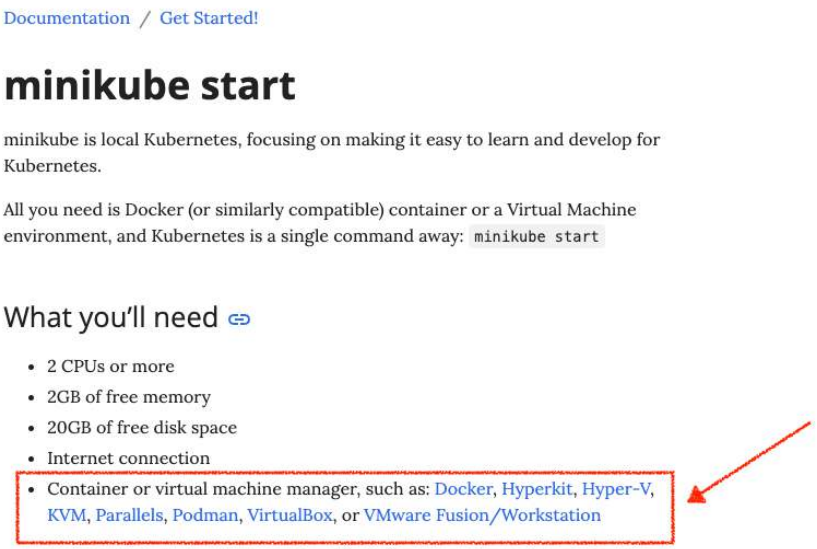


Figure 2.9 – Prerequisites for minikube

2. Under **1 Installation**, select the combination that is valid for you. As an example, you can see the authors' selection for a *MacBook Air M1 laptop* as the target machine:

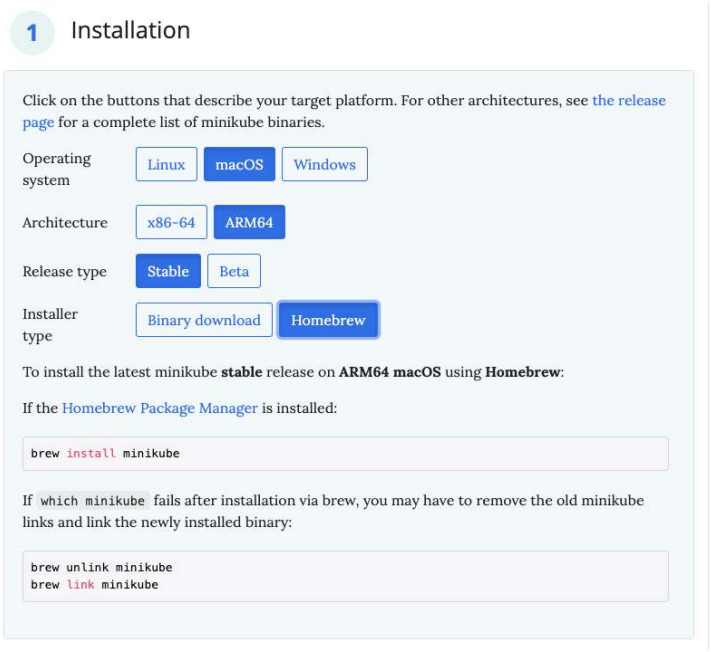


Figure 2.10 – Selecting the configuration

Installing minikube for a MacBook Air M1 using Homebrew

Follow these steps:

1. In a Terminal window, execute the steps shown previously. In the authors' case, this is as follows:

```
$ brew install minikube
```

2. Test the installation with the following command:

```
$ brew version
minikube version: v1.28.0
commit: 986b1ebd987211ed16f8cc10aed7d2c42fc8392f
```

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

```
$ minikube start
```

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

4. 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 this:

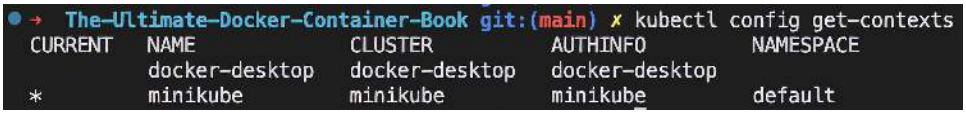


Figure 2.11 – List of contexts for kubectl after installing minikube

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:

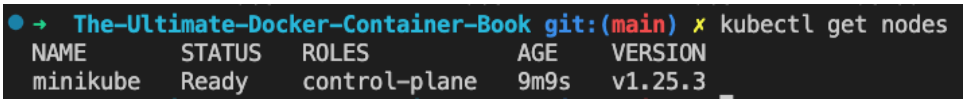


Figure 2.12 – Showing the list of cluster nodes for the minikube cluster

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 `v1.25.3`.

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` folder in your home directory (`~`), then you should find a folder setup inside this folder that contains a `.yaml` file, which we're going to use for this test:

   I. Open a new Terminal window.

   II. Navigate to the `The-Ultimate-Docker-Container-Book` folder:

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

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

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

   You should see this output:

```
pod/nginx created
```

   IV. We can double-check whether the pod is running with kubectl:

```
$ kubectl get pods
```

   We should see this:

```
NAME      READY    STATUS      RESTARTS    AGE
nginx      1/1     Running        0        11m
```

   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 this:

```
→ The-Ultimate-Docker-Container-Book git:(main) ✗ kubectl get services
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP        59m
nginx        NodePort    10.104.77.208   <none>        80:30373/TCP   11m
```

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 `30373` of the cluster node of our Kubernetes cluster in minikube.

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:

```
→ The-Ultimate-Docker-Container-Book git:(main) ✗ minikube service nginx
|-----------|-------|-------------|------------------------|
| NAMESPACE | NAME  | TARGET PORT |          URL           |
|-----------|-------|-------------|------------------------|
| default   | nginx |          80 | http://192.168.49.2:30373 |
|-----------|-------|-------------|------------------------|
🏃  Starting tunnel for service nginx.
|-----------|-------|-------------|------------------------|
| NAMESPACE | NAME  | TARGET PORT |          URL           |
|-----------|-------|-------------|------------------------|
| default   | nginx |             | http://127.0.0.1:64171 |
|-----------|-------|-------------|------------------------|
🎉  Opening service default/nginx in default browser...
❗  Because you are using a Docker driver on darwin, the terminal needs to be open to run it.
```

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 `30373` to port `64171` 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://127.0.0.1:64171`. You should see the welcome screen for Nginx:

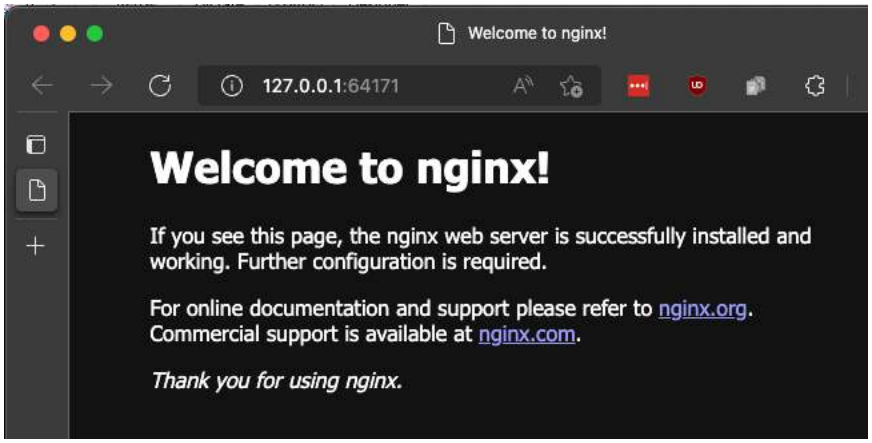


Figure 2.15 – Welcome screen of Nginx running on a Kubernetes cluster on minikube

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
```

4. You should see this:

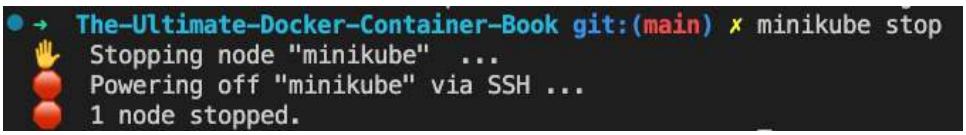


Figure 2.16 – Stopping minikube

Working with a multi-node minikube cluster

At times, testing with a single-node cluster is not enough. Worry not – minikube has got 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      v1.25.3
demo-m02      Ready     <none>           45s      v1.25.3
demo-m03      Ready     <none>           22s      v1.25.3
```

We have a 3-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 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 for 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/`:

   I. On MacOS, use Homebrew to install Kind with the following command:

```
$ brew install kind
```

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

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

III. 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 v0.17.0 go1.19.2 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:

```
➜  The-Ultimate-Docker-Container-Book git:(main) ✗ kind create cluster
Creating cluster "kind" ...
 ✓ Ensuring node image (kindest/node:v1.25.3) 🖼
 ✓ Preparing nodes 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹️
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
Set kubectl context to "kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind

Not sure what to do next? 😅  Check out https://kind.sigs.k8s.io/docs/user/quick-start/
```

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 show clusters
Kind
demo
```

And this works as expected.

Testing Kind

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. 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:

```
→ The-Ultimate-Docker-Container-Book git:(main) ✗ k config get-contexts
CURRENT   NAME             CLUSTER          AUTHINFO         NAMESPACE
          docker-desktop   docker-desktop   docker-desktop
*         kind-demo        kind-demo        kind-demo
          kind-kind        kind-kind        kind-kind
```

Figure 2.18 – List of contexts defined for kubectl

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

2. Use the following command to make the demo cluster your current cluster if the asterisk is indicating that another cluster is current:

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

3. Let's list all the nodes of the `sample-cluster` cluster:

```
$ kubectl get nodes
```

The output should be like this:

```
→ The-Ultimate-Docker-Container-Book git:(main) ✗ kubectl get nodes
NAME                 STATUS   ROLES           AGE     VERSION
demo-control-plane   Ready    control-plane   2m25s   v1.25.3
```

Figure 2.19 – Showing the list of nodes on the kind cluster

4. 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
```

5. 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
```

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

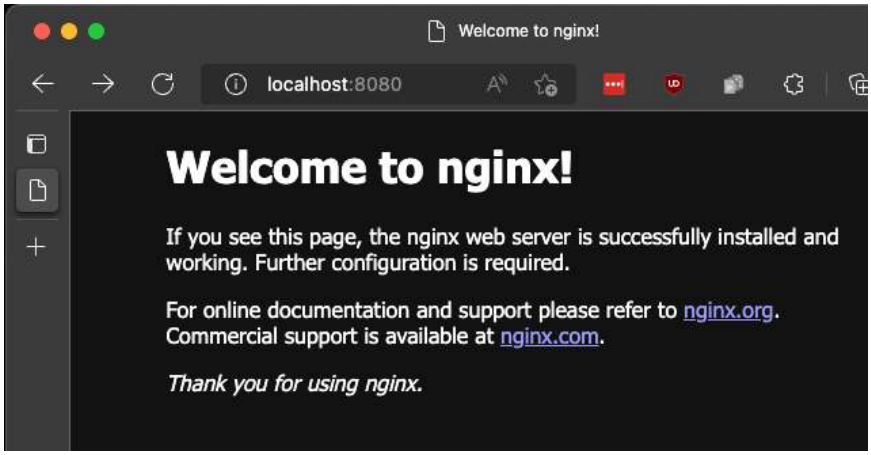


Figure 2.20 – Welcome screen of Nginx running on a Kind cluster

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

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

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

```
$ kind delete cluster --name kind
$ kind delete cluster --name 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 set up and configured our personal or working environment so that we can productively work with Docker containers. This equally applies to developers, DevOps, and operations engineers.

We started with a package manager that should be at the fingertip of every serious engineer. It makes installing and managing applications and tools so much easier. Next, we made sure that we used a good shell for scripting – a powerful editor. We then made sure to have Docker Desktop installed, which we can use to run and test containers natively. Finally, we installed and quickly tested minikube and Kind on our machine. The latter are tools that can be used to run and test our containers on a local Kubernetes cluster.

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/`
- *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. Name three to four Linux distributions on which Docker is certified to run.
5. You installed minikube on your system. What kind of tasks will you use this tool for?

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 Desktop, 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. The following Linux distros are certified to run Docker: **Red Hat Linux** (**RHEL**), CentOS, Oracle Linux, Ubuntu, and more.
5. minikube makes it possible to define and run a single or multi-node cluster on a local computer such as a developer's laptop. This way, using minikube, you can run and test containerized applications locally on your machine and do not have to rely on a remote Kubernetes cluster such as one running in the cloud on, say, AWS, Microsoft Azure, or Google cloud.

Part 2: Containerization Fundamentals

This part teaches you how to start, stop, and remove containers, and how to inspect containers to retrieve additional metadata from them. Furthermore, it explains how to run additional processes and how to attach to the main process in an already running container. It also covers how to retrieve logging information from a container, which is produced by the processes running inside it. Finally, this part introduces the inner workings of a container, including such things as Linux namespaces and groups.

- *Chapter 3*, *Mastering Containers*
- *Chapter 4*, *Creating and Managing Container Images*
- *Chapter 5*, *Data Volumes and Configuration*
- *Chapter 6*, *Debugging Code Running in Containers*
- *Chapter 7*, *Testing Applications Running in Containers*
- *Chapter 8*, *Increasing Productivity with Docker Tips and Tricks*

3
Mastering Containers

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 about 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 layman, the underpinnings of a container

Technical requirements

For this chapter, you should have Docker Desktop installed on your Linux workstation, macOS, or Windows PC. If you are on an older version of Windows or are using Windows 10 Home Edition, then you should have Docker Toolbox installed and ready to use. 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:

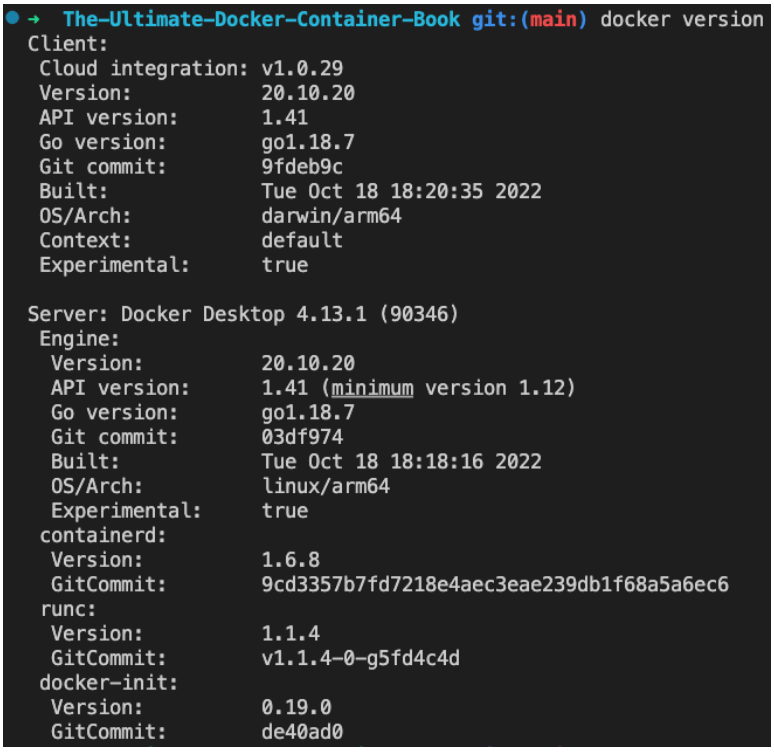


Figure 3.1 – Output of the docker version command

As you can see, I have version `20.10.20` installed on my MacBook Air M1 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:

```
➜  The-Ultimate-Docker-Container-Book git:(main) docker container run alpine echo "Hello World"
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
6875df1f5354: Pull complete
Digest: sha256:b95359c2505145f16c6aa384f9cc74eeff78eb36d308ca4fd902eeeb0a0b161b
Status: Downloaded newer image for alpine:latest
Hello World
```

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 Docker Engine, which 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 `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.

> **Alpine Linux**
>
> `alpine` is a minimal Docker image based on Alpine Linux with a complete package index and is only about 5 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:

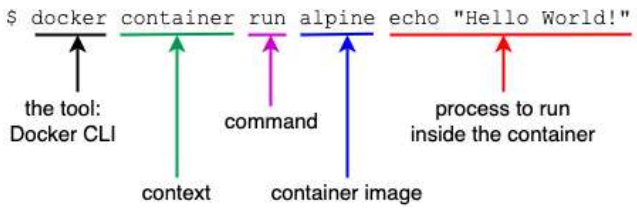


Figure 3.3 – docker run command explained

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 centos ping -c 5 127.0.0.1
```

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

```
→  The-Ultimate-Docker-Container-Book git:(main) docker container run centos ping -c 5 127.0.0.1
Unable to find image 'centos:latest' locally
latest: Pulling from library/centos
52f9ef134af7: Pull complete
Digest: sha256:a27fd8080b517143cbbbab9dfb7c8571c40d67d534bbdee55bd6c473f432b177
Status: Downloaded newer image for centos:latest
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.837 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.142 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.207 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.123 ms
64 bytes from 127.0.0.1: icmp_seq=5 ttl=64 time=0.144 ms

--- 127.0.0.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4091ms
rtt min/avg/max/mdev = 0.123/0.290/0.837/0.275 ms
```

Figure 3.4 – Running the ping command inside a CentOS container

What changed is that this time, the container image we're using is `centos` and the process we're executing inside the `centos` container is `ping -c 5 127.0.0.1`, which pings the loopback IP address (`127.0.0.1`) five times until it stops.

> **CentOS**
>
> `centos` is the official Docker image for CentOS Linux, which is a community-supported distribution derived from sources freely provided to the public by Red Hat for **Red Hat Enterprise Linux** (**RHEL**).

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

```
Unable to find image 'centos:latest' locally
```

This tells us that Docker didn't find an image named `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`. This is expressed by the second line, as follows:

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

The next three lines of output are as follows:

```
52f9ef134af7: Pull complete
Digest: sha256:a27fd8080b517143cbbbab9dfb7c8571c4...
Status: Downloaded newer image for centos:latest
```

This tells us that Docker has successfully pulled the `centos:latest` image from Docker Hub. All of the subsequent lines of the output are generated by the process we ran inside the container, which is the `ping` 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.

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 `http://jservice.io/`.

Now, the goal is to have a process running inside a container that produces a new random trivia question every 5 seconds and outputs the question to `STDOUT`. The following script will do exactly that:

```
while :
do
    curl -s http://jservice.io/api/random | jq '.[0].question'
    sleep 5
done
```

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

```
while ($true) {
  Invoke-WebRequest -Uri "http://jservice.io/api/random" -Method GET
-UseBasicParsing |
  Select-Object -ExpandProperty Content |
  ConvertFrom-Json |
  Select-Object -ExpandProperty 0 |
  Select-Object -ExpandProperty question
  Start-Sleep -Seconds 5
}
```

> **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:

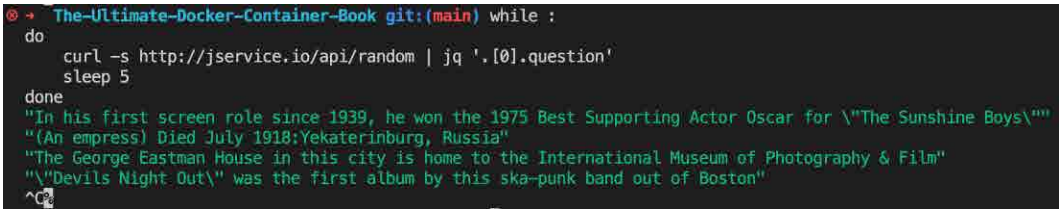


Figure 3.5 – Output random trivia

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 on screen. Use your package manager to install `jq` if needed. On Windows, using Chocolatey, the command would be as follows:

```
$ choco install jq
```

And 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:ed2
```

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:ed2` image. Note how we are also using a tag, `ed2`, for the container. This tag just tells us that this image was originally created for the second 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:

```
● → The-Ultimate-Docker-Container-Book git:(main) docker container ls -l
CONTAINER ID   IMAGE                              COMMAND                  CREATED         STATUS         PORTS     NAMES
4e3153c395d0   fundamentalsofdocker/trivia:ed2   "/bin/sh -c 'source …"   6 minutes ago   Up 6 minutes             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 6 minutes`. That is, the container has been up and running for 6 minutes 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 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:

Figure 3.7 – List of all running containers on the system

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 Secure Hash Algorithm 256-bit (SHA-256) 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.
`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.

In the previous section, we used the `-l` parameter in 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

Sometimes, we want to (temporarily) stop a running container. 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:ed2
   ```

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}')
```

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
```

> **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.

> **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.

Inspecting containers

Containers are runtime instances of an image and have a lot of associated data that characterizes their behavior. To get more information about a specific container, we can use the `inspect` command. As usual, 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, if it is not already running, we have to run it:

```
$ docker container run --name trivia fundamentalsofdocker/ trivia:ed2
```

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:

```
● → The-Ultimate-Docker-Container-Book git:(main) docker container inspect trivia
[
    {
        "Id": "5a805de0ea1ca27d5a945ed826c92bab89aee10ece6e71c79de0b70c14933de3",
        "Created": "2022-11-12T18:33:11.56114155Z",
        "Path": "/bin/sh",
        "Args": [
            "-c",
            "source script.sh"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 6953,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2022-11-12T18:52:44.269934968Z",
            "FinishedAt": "2022-11-12T18:52:20.690762429Z"
        },
```

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:

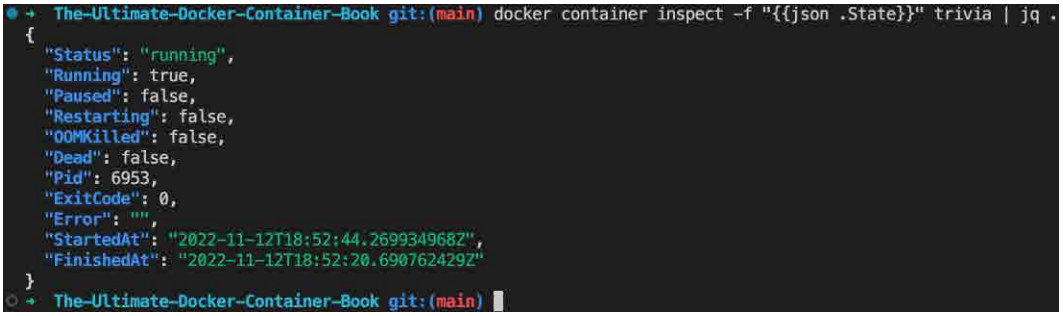


Figure 3.9 – The state node of the inspect output

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

Sometimes, we want to run another process inside an already-running container. A typical reason could be to try to debug a misbehaving container. 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 `-i` (or `--interactive`) flag 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:

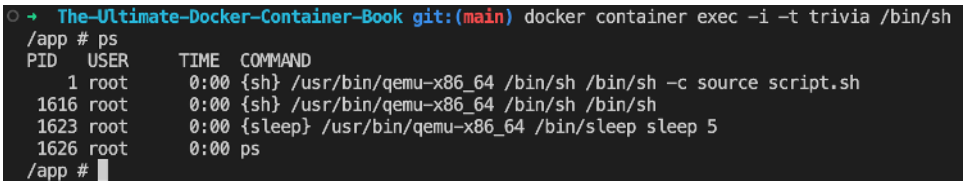


Figure 3.10 – Executing into the running trivia container

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 cannot only execute additional processes interactively in a container. Please consider the following command:

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

The output evidently looks very similar to the preceding output:

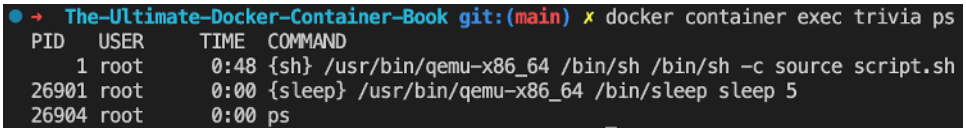


Figure 3.11 – List of processes running inside the trivia container

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 this output in the terminal:

   ```
   Hello World
   ```

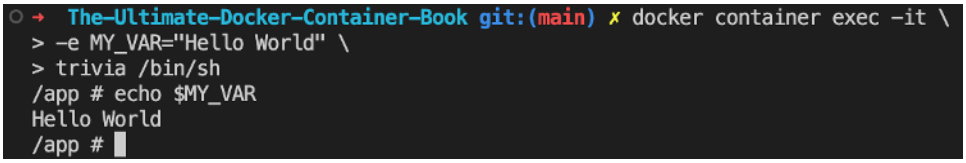


Figure 3.12 – Running a trivia container and defining an environment variable

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

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

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

We can use the `attach` command to attach our terminal's standard input, output, or 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.

> **Tip**
> You may want to use another terminal 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.

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

```
$ docker container run -it \
    --name trivia2 fundamentalsofdocker/trivia:ed2
```

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

```
$ docker container attach trivia2
```

In this case, we will see, every 5 seconds or so, a new quote appearing in the output.

4. To quit the container without stopping or killing it, we can use the *Ctrl* + *P* + *Ctrl* + *Q* key combination. This detaches us from the container while leaving it running in the background.
5. Stop and remove the container forcefully:

```
$ docker container rm --force trivia2
```

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
```

> **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*, *Using 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.

2. 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:

```
● → The-Ultimate-Docker-Container-Book git:(main) ✗ 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

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

```
$ docker container attach nginx
```

4. 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> 1..10 | ForEach-Object {C:\ProgramData\chocolatey\bin\curl.
exe -4 localhost:8080}
```

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

```
➜  The-Ultimate-Docker-Container-Book git:(main) ✗ docker container attach nginx
172.17.0.1 - - [13/Nov/2022:17:11:39 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.79.1" "-"
172.17.0.1 - - [13/Nov/2022:17:11:39 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.79.1" "-"
172.17.0.1 - - [13/Nov/2022:17:11:39 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.79.1" "-"
172.17.0.1 - - [13/Nov/2022:17:11:39 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.79.1" "-"
172.17.0.1 - - [13/Nov/2022:17:11:39 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.79.1" "-"
172.17.0.1 - - [13/Nov/2022:17:11:39 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.79.1" "-"
172.17.0.1 - - [13/Nov/2022:17:11:39 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.79.1" "-"
172.17.0.1 - - [13/Nov/2022:17:11:39 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.79.1" "-"
172.17.0.1 - - [13/Nov/2022:17:11:39 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.79.1" "-"
172.17.0.1 - - [13/Nov/2022:17:11:39 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.79.1" "-"
```

Figure 3.14 – Output of Nginx

5. Quit the container by pressing *Ctrl* + *C*. This will detach your terminal and, at the same time, stop the Nginx container.
6. 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:ed2
```

Let it run for a minute or so so that it has 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**
>
> 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.

3. 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 items that the process running inside the container produced.

4. 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
```

5. Press *Ctrl* + *C* to stop following the logs.
6. 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.
`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

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 run 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 and 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**
>
> On a Mac with an M1/M2 CPU, at the time of writing, 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 my MacBook Air M1, 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, Vagrant replies with the following:

```
Vagrant 2.3.2
```

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

```
$ vagrant init bento/ubuntu-22.04
```

Here is the generated output:

```
➜  The-Ultimate-Docker-Container-Book git:(main) vagrant init bento/ubuntu-22.04
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.

> **Note**
> On a Mac with an M1/M2 CPU, at the time of writing, the `bento/ubuntu-22.4` image does not work. An alternative that seems to work is `illker/ubuntu-2004`.

6. Now, start this VM using Vagrant:

```
$ vagrant up
```

7. Connect from your laptop to the VM using Secure Shell (`ssh`):

```
$ vagrant ssh
```

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

8. Once inside the Ubuntu VM, 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:

```
$ 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.

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

```
$ 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 on 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 from 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 how this all fits together:

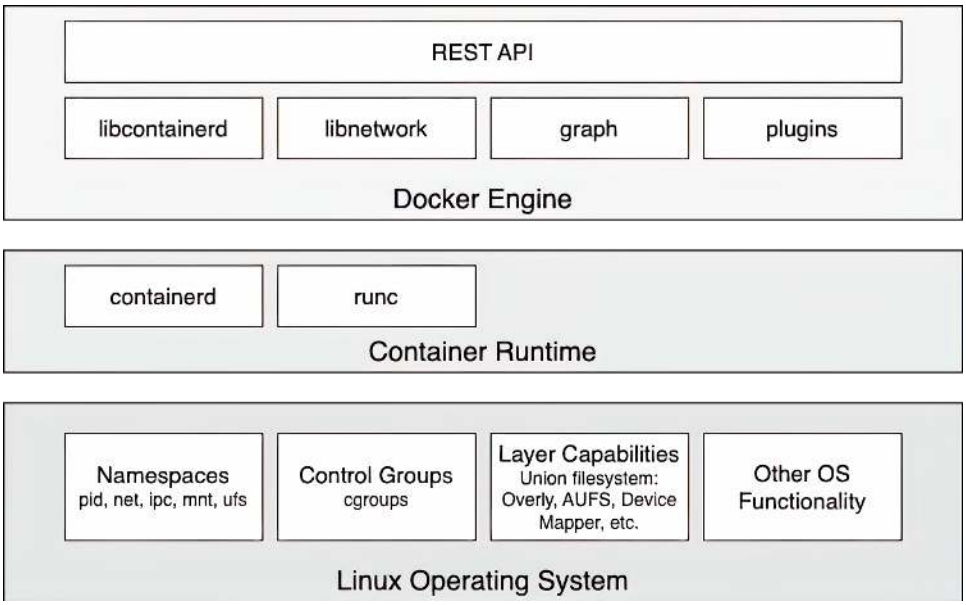


Figure 3.16 – High-level architecture of Docker

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 now sits the Docker engine. The 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 their 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:

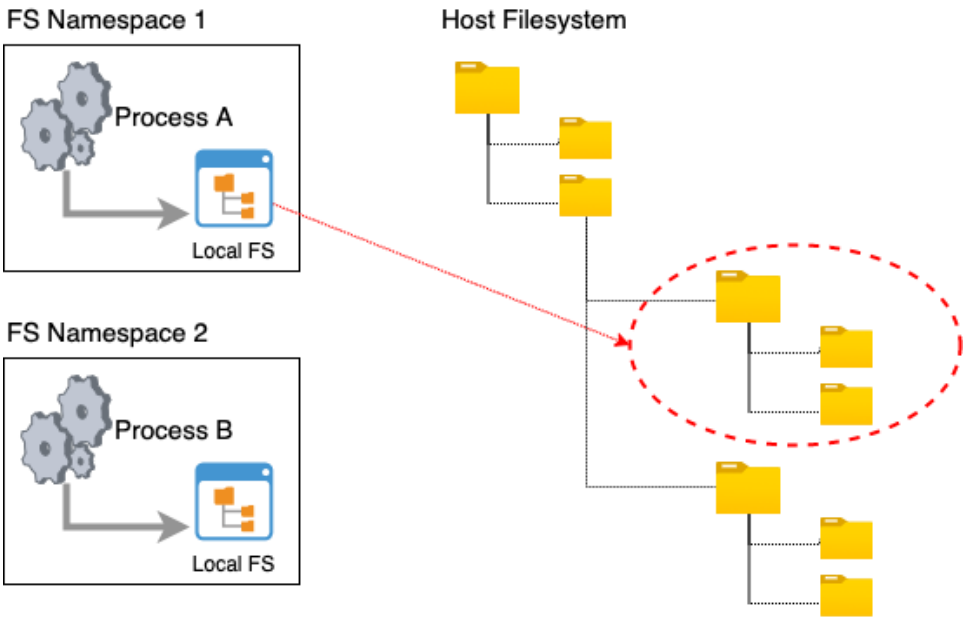


Figure 3.17 – Namespaces explained

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`:

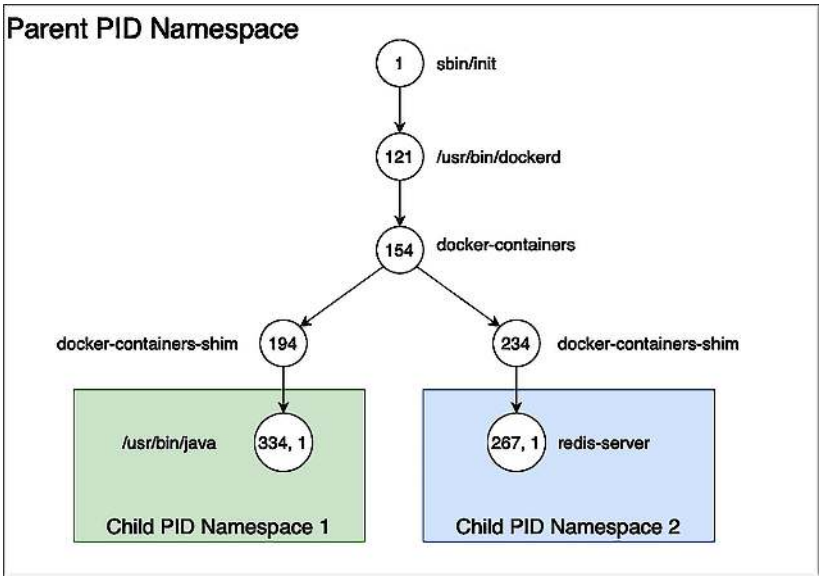


Figure 3.18 – Process tree on a Docker host

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 the 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 the 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 top of which Docker Engine is built comprises 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. Two important components are `runc` and `containerd`.

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.

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 important concepts of Linux are enabling factors for containers?
2. What are the possible states a container can be in?
3. Which command helps us to find out which containers are currently running on our Docker host?
4. Which command is used to list only the IDs of all containers?

Answers

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

1. Linux had to first introduce **namespaces** and **cgroups** to make containers possible. Containers use those two concepts extensively. Namespaces are used to encapsulate and thus protect resources defined and/or running inside a container. cgroups are used to limit the resources processes running inside a container can use, such as memory, bandwidth, or CPU.
2. The possible states of a Docker container are as follows:
    - `created`: A container that has been created but not started

- `restarting`: A container that is in the process of being restarted
- `running`: A currently running container
- `paused`: A container whose processes have been paused
- `exited`: A container that ran and completed
- `dead`: A container that Docker Engine tried and failed to stop

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 container IDs, 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.

4

Creating and Managing Container Images

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 dived 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 images?
- Creating images
- Lift and shift – containerizing a legacy app
- Sharing or shipping images

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

- Name three of the most important characteristics of a container image
- Create a custom image by interactively changing the container layer and committing it
- Author a simple Dockerfile to generate a custom image
- Export an existing image using `docker image save` and import it into another Docker host using `docker image load`
- Write a two-step Dockerfile that minimizes the size of the resulting image by only including the resulting artifacts in the final image

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.

> **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:

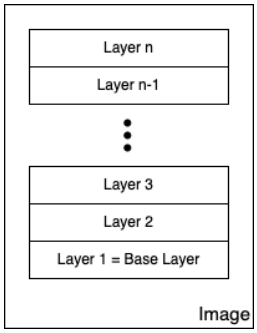


Figure 4.1 – The image as a stack of layers

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. Different storage drivers are available that 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.

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

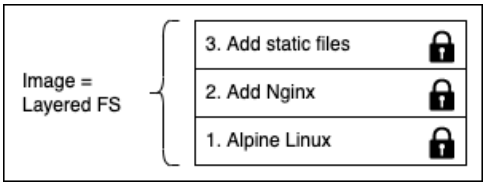


Figure 4.2 – A sample custom image based on Alpine and Nginx

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, Alpine, Ubuntu, or CentOS. However, it is also possible to create an image from scratch.

> **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.

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:

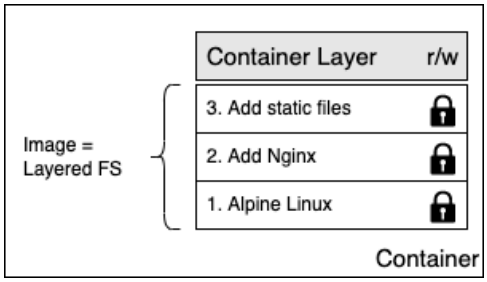


Figure 4.3 – The writable container layer

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:

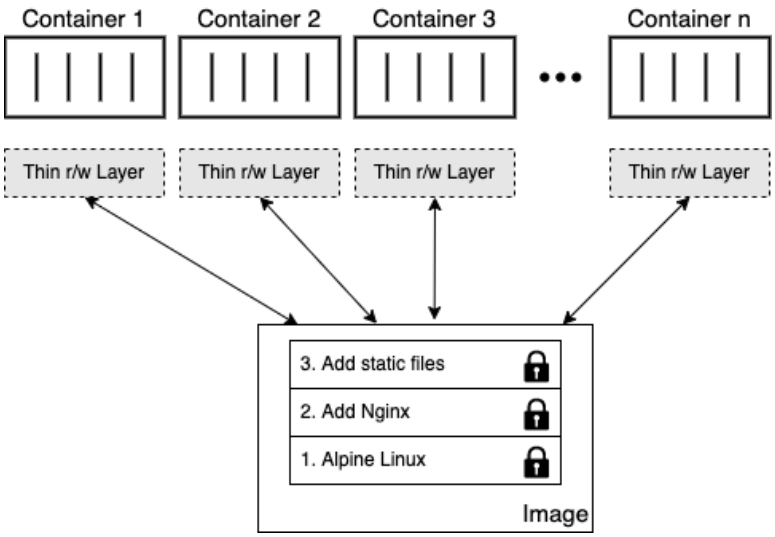


Figure 4.4 – Multiple containers sharing the same image layers

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 low-lying layers, then it just uses it. If, on the other hand, a layer wants to modify, say, a file from a low-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:

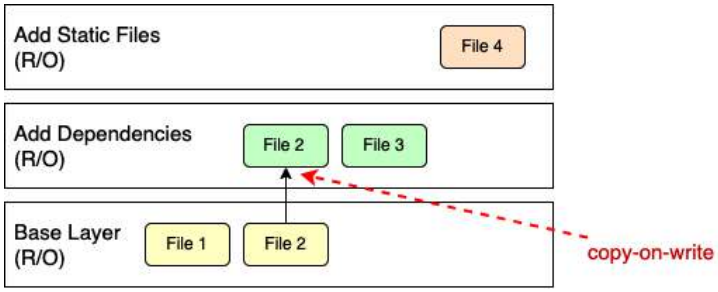


Figure 4.5 – Docker image using copy-on-write

The second layer wants to modify **File 2**, which is present in the base layer. Thus, it copies it up and then modifies it. 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 are what enable the Union filesystem. Graph drivers are also called storage drivers and are used when dealing with layered container images. A graph driver consolidates multiple image layers into a root filesystem for the mount namespace of the container. Or, put differently, the driver controls how images and containers are stored and managed on the Docker host.

Docker supports several different graph drivers using a pluggable architecture. The preferred driver is **overlay2,** followed by **overlay.**

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:3.17 /bin/sh
```

   The preceding command runs a container based on the `alpine:3.17` 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:

```
➜  ch04 git:(main) ✗ docker container run -it \
> --name sample \
> alpine:3.17 /bin/sh
Unable to find image 'alpine:3.17' locally
3.17: Pulling from library/alpine
261da4162673: Pull complete
Digest: sha256:8914eb54f968791faf6a8638949e480fef81e697984fba772b3976835194c6d4
Status: Downloaded newer image for alpine:3.17
/ #
```

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/v3.17/main/aarch64/APKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.17/community/aarch64/APKINDEX.tar.gz
v3.17.0-21-g62c4fc0981 [https://dl-cdn.alpinelinux.org/alpine/v3.17/main]
v3.17.0-24-geb334dcd4f [https://dl-cdn.alpinelinux.org/alpine/v3.17/community]
OK: 17672 distinct packages available
(1/5) Installing ca-certificates (20220614-r2)
(2/5) Installing brotli-libs (1.0.9-r9)
(3/5) Installing nghttp2-libs (1.51.0-r0)
(4/5) Installing libcurl (7.86.0-r1)
(5/5) Installing curl (7.86.0-r1)
Executing busybox-1.35.0-r29.trigger
Executing ca-certificates-20220614-r2.trigger
OK: 10 MiB in 20 packages
/ #
```

Figure 4.7 – Installing curl on Alpine

4. Now, we can indeed use `curl`, 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
date: Sun, 27 Nov 2022 13:33:27 GMT
expires: Sun, 27 Nov 2022 13:33:27 GMT
cache-control: private, max-age=2592000
server: gws
content-length: 220
x-xss-protection: 0
x-frame-options: SAMEORIGIN
set-cookie: CONSENT=PENDING+438; expires=Tue, 26-Nov-2024 13:33:27 GMT; path=/; domain=.google.com; Sec
ure
p3p: CP="This is not a P3P policy! See g.co/p3phelp for more info."
alt-svc: h3=":443"; ma=2592000,h3-29=":443"; ma=2592000,h3-Q050=":443"; ma=2592000,h3-Q046=":443"; ma=2
592000,h3-Q043=":443"; ma=2592000,quic=":443"; ma=2592000; v="46,43"
```

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:

```
5266d7da377c   alpine:3.17    "/bin/sh"                 2 hours
ago      Exited (0) 48 seconds ago
```

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:

```
→  ch04 git:(main) ✗ docker container diff sample
C /var
C /var/cache
C /var/cache/apk
A /var/cache/apk/APKINDEX.ac15ed62.tar.gz
A /var/cache/apk/APKINDEX.c3d4ed66.tar.gz
A /-sSL
C /etc
C /etc/apk
C /etc/apk/world
C /etc/apk/protected_paths.d
A /etc/apk/protected_paths.d/ca-certificates.list
C /etc/ssl
C /etc/ssl/certs
A /etc/ssl/certs/fc5a8f99.0
A /etc/ssl/certs/ca-cert-HiPKI_Root_CA_-_G1.pem
A /etc/ssl/certs/ca-cert-NetLock_Arany_=Class_Gold=_Főtanúsítvány.pem
A /etc/ssl/certs/ca-cert-Secure_Global_CA.pem
A /etc/ssl/certs/ca-cert-GlobalSign_Root_CA_-_R6.pem
A /etc/ssl/certs/ca-cert-SSL.com_EV_Root_Certification_Authority_ECC.pem
A /etc/ssl/certs/0b1b94ef.0
A /etc/ssl/certs/aee5f10d.0
A /etc/ssl/certs/ca-cert-DigiCert_Assured_ID_Root_G2.pem
```

Figure 4.9 – Output of the docker diff command (truncated)

We have shortened the preceding output for better readability. In the list, `A` stands for added, and `C` stands for changed. If we had any deleted files, then those would be prefixed with `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
```

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

```
sha256:5287bccbb3012ded35e7e992a5ba2ded9b8b5d0...
```

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.

11. We can verify this by listing all the images on our system, as follows:

```
$ docker image ls
```

We can see this image ID as follows:

```
→  ch04 git:(main) ✗ docker image ls
REPOSITORY                    TAG       IMAGE ID       CREATED              SIZE
my-alpine                     latest    5287bccbb301   About a minute ago   12.4MB
alpine                        3.17      d3156fec8bcb   4 days ago           7.46MB
nginx                         alpine    d0ddde8e3f4f   2 weeks ago          22.1MB
alpine                        latest    2b4661558fb8   2 weeks ago          5.29MB
busybox                       latest    2d0d8216f525   4 weeks ago          1.46MB
centos                        latest    e6a0117ec169   14 months ago        272MB
fundamentalsofdocker/trivia   ed2       bbc92c8f014d   3 years ago          7.94MB
```

Figure 4.10 – Listing all Docker images

We can see that the image named `my-alpine` has the expected ID of `5287bccbb301` (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.

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

```
$ docker image history my-alipine
```

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

```
→  ch04 git:(main) ✗ docker image history my-alpine
IMAGE          CREATED         CREATED BY                                      SIZE      COMMENT
5287bccbb301   6 minutes ago   /bin/sh                                         4.98MB
d3156fec8bcb   4 days ago      /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>      4 days ago      /bin/sh -c #(nop) ADD file:685b5edadf1d5bf0a…   7.46MB
```

Figure 4.11 – 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 3.17 Docker image. It was created and uploaded 4 days 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.

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

> **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:

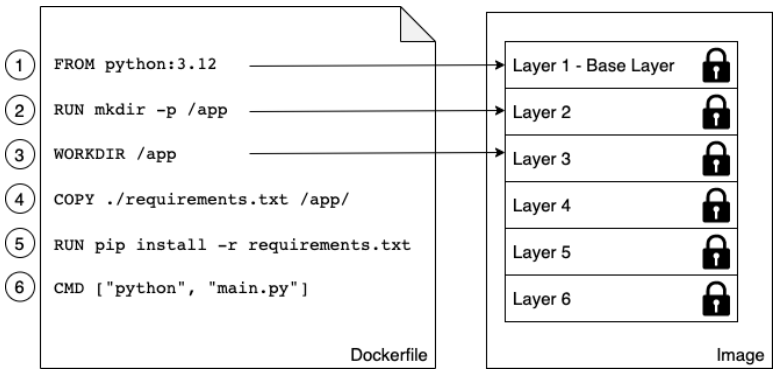


Figure 4.12 – The relationship between a Dockerfile and the layers in an image

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 CentOS 7, for example, we would have the following line in the Dockerfile:

```
FROM centos:7
```

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 into 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.

> **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, as well as provide an URI 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**
> 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 handling it that is not 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:3.17
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
```

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 preceeding command:

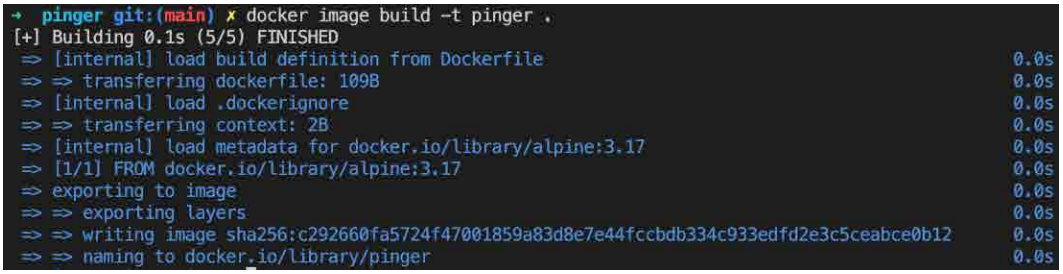


Figure 4.13 – Building the pinger Docker image

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

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

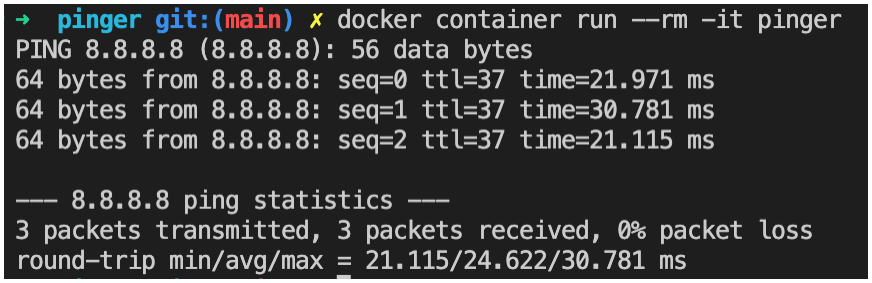


Figure 4.14 – Output of the pinger container

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 I can now override the CMD part that I have defined in the Dockerfile (remember, it was `["-c", "3","8.8.8.8"]`) when I 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 in the container instead of the `ping` command. We could do so by using the following command:

```
$ docker container run --rm -it --entrypoint /bin/sh 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:3.17
CMD wget -O - http://www.google.com
```

Here, I have even used the shell form to define 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:19-buster-slim
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:19-buster-slim` 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.

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
   ```

2. Create a new subfolder for *Chapter 4* and navigate to it:

   ```
   $ mkdir ch04 && cd ch04
   ```

3. In the preceding folder, create a `sample1` subfolder and navigate to it, like this:

   ```
   $ mkdir sample1 && cd sample1
   ```

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

   ```
   FROM centos:7
   RUN yum 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-centos .
   ```

Please note that there is a period (.) at the end of the preceding command:

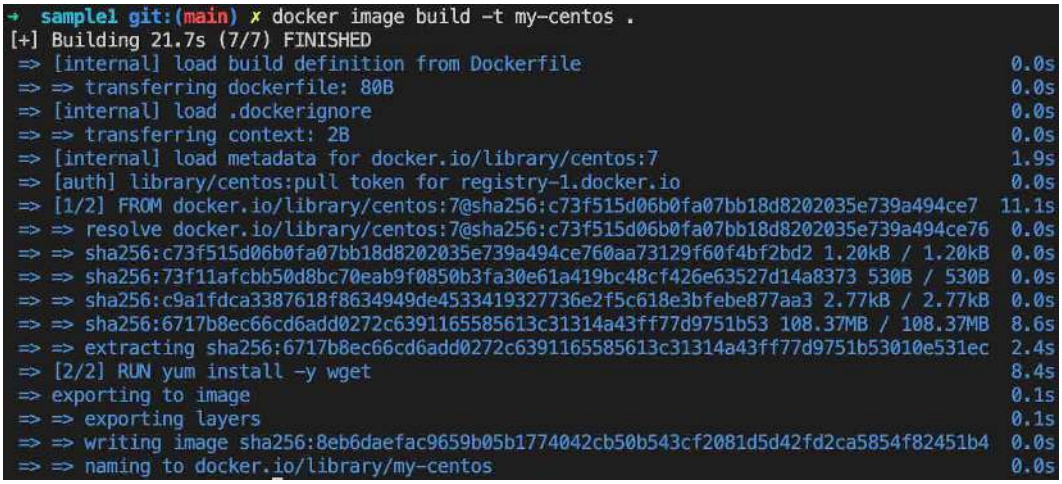


Figure 4.15 – Building our first custom image from CentOS

The previous command means that the Docker builder creates a new image called `my-centos` 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-centos -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 4.15*. This output is created by the Docker build kit:

1. First, we have the following line:

   ```
   [+] Building 21.7s (7/7) 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 22 seconds and was executed in 7 steps.

2. Now, let's skip the next few lines until we reach this one:

   ```
   => [1/2] FROM docker.io/library/centos:7@sha256:c73f51...
   ```

   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 centos:7` 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 yum install -y wget
```

   This is our second line in the Dockerfile, where we want to use the `yum` 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 sha256:8eb6daefac9659b05b17740...
=> => naming to docker.io/library/my-centos
```

   Here, the builder finalizes building the image and provides the image with the `sha256` code of `8eb6daefac9....`

   This tells us that the resulting custom image has been given an ID of `8eb6daefac9...` and has been tagged with `my-centos: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.

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/ch04
```

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 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
CMD /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 has to install the Alpine **Software Development Kit** (**SDK**), which, among other tools, contains the C++ compiler we need to build the application.

7. Once the build is done, we can list the image and see the size that's been shown, as follows:

```
$ docker image ls | grep hello-world
```

In the author's case, the output is as follows:

```
hello-world   latest   42c0c7086fbf   2 minutes ago   215MB
```

With a size of 215 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 to your folder called `Dockerfile.multi-step` with this content:

```
FROM alpine:3.12 AS build
RUN apk update && \
    apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
COPY . /app
RUN mkdir bin
RUN gcc hello.c -o bin/hello

FROM alpine:3.12
```

```
COPY --from=build /app/bin/hello /app/hello
CMD /app/hello
```

Here, we have the first stage with an alias called `build`, which is used to compile the application; then, the second stage uses the same `alpine:3.12` base image but does not install the SDK, and 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:

```
hello-world-small latest 72c... 20 seconds ago 5.34MB
hello-world latest 42c... 10 minutes ago 215
```

We have been able to reduce the size from 215 MB down to 5.34 MB. This is a reduction in size by a factor of approximately 40. A smaller image has many advantages, such as a smaller attack surface area for hackers, reduced memory and disk consumption, faster startup times for the corresponding containers, and a reduction of the bandwidth needed to download the image from a registry, such as Docker Hub.

Dockerfile best practices

There are a few recommended best practices to consider when authoring a Dockerfile, which are as follows:

- 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 the absolute minimum 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 at a minimum, as well as the time needed to terminate or clean up the application.
- 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 at 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:19
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, and those are all downloaded and installed in this step. It can take minutes until it is done. Therefore, we want to avoid running `npm install` each time we rebuild the image, but a developer changes their source code all the time during the development of an application. That 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:19
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.

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.

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 into 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`:

```
➜  ch04 git:(main) ✗ mkdir backup
➜  ch04 git:(main) ✗ docker image save -o ./backup/my-alpine.tar my-alpine
➜  ch04 git:(main) ✗ ls -al backup
total 25400
drwxr-xr-x  3 gabriel  staff        96 27 Nov 16:18 .
drwxr-xr-x  5 gabriel  staff       160 27 Nov 16:17 ..
-rw-------  1 gabriel  staff  13003264 27 Nov 16:18 my-alpine.tar
```

Figure 4.16 – 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.

Lift and shift – containerizing a legacy app

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, the 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 it can also be used 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 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.

Source code and build instructions

The next step is to locate all the source code and other assets, such as images and 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 contribute 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` versus `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 the application uses or depends on, and not secrets provided by the users of the application. The actor here is our application, which needs to be authenticated and authorized by external authorities and service providers.

There are various ways traditional applications got 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 the 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 Vault, AWS Secrets Manager, or Azure Key Vault?

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, so 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**
>
> 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:3.5
```

This will pull the Alpine image that has been tagged with `3.5`.

Demystifying image namespaces

So far, we have pulled various images and haven't worried 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 `https://jfrog.com/integration/artifactorydocker-registry/`
`<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.

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.
`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 in Google's container registry.

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.

In 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
   ```

   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.

Summary

In this chapter, we discussed what container images are and how we can build and ship them. As we have seen, there are three different ways that an image can be created – either manually, automatically, or by importing a tarball into the system. We also learned some of the best practices commonly used when building custom images. Finally, we got a quick introduction to how to share or ship custom images by uploading them to a container image registry such as Docker Hub.

In the next chapter, we're going to introduce Docker volumes, which can be used to persist the state of a container. We'll also show you how to define individual environment variables for the application running inside the container, as well as how to use files containing whole sets of configuration settings.

Questions

Please try to answer the following questions to assess your learning progress:

1. How would you create a Dockerfile that inherits from Ubuntu version 22.04, and that installs `ping` and runs `ping` when a container starts? The default address used to ping should be `127.0.0.1`.
2. 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`.
3. Create a Dockerfile that uses multiple steps to create an image of a Hello World app of minimal size, written in C or Go.
4. Name three essential characteristics of a Docker container image.
5. 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. The Dockerfile could look like this:

```
FROM ubuntu:22.04
RUN apt-get update && \
apt-get install -y iputils-ping
CMD ping 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 .
```

2. 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 .
```

3. The Dockerfile for a Go application could look like this:

```
FROM golang:alpine
WORKDIR /app
ADD . /app
RUN go env -w GO111MODULE=off
RUN cd /app && go build -o goapp
ENTRYPOINT ./goapp
```

   You can find the full solution in the `~/The-Ultimate-Docker-Container-Book/sample-solutions/ch04/answer03` folder.

4. 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

5. 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

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

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 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
```

> **Note**
> If you did not check out the GitHub repository at the default location, the preceding command may vary for you.

2. Create a sub-folder for *Chapter 5* and navigate to it:

```
$ mkdir ch05 && cd ch05
```

Let's get started!

Creating and mounting data volumes

All meaningful applications consume or produce data. Yet containers are, ideally, 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 is creating 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:

```
→  ch05 git:(main) ✗ docker volume inspect sample
[
    {
        "CreatedAt": "2022-12-04T10:46:58Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/sample/_data",
        "Name": "sample",
        "Options": {},
        "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. In the top-right corner, click the **Create** button, as shown in the following screenshot:

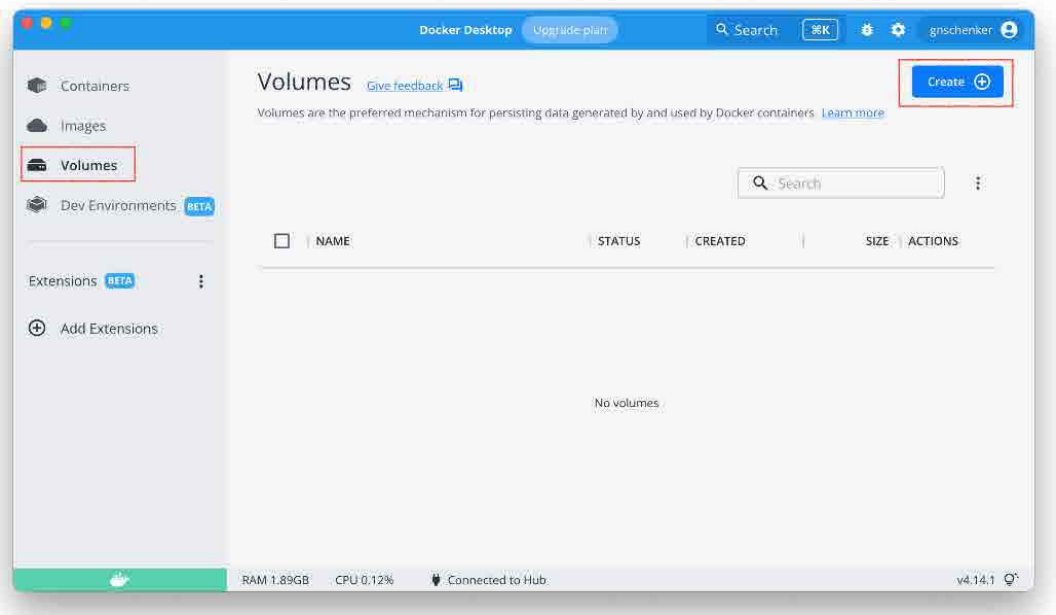


Figure 5.2 – Creating a new Docker volume with Docker Desktop

D. Type in `sample-2` as the name for the new volume and click **Create**. You should now see this:

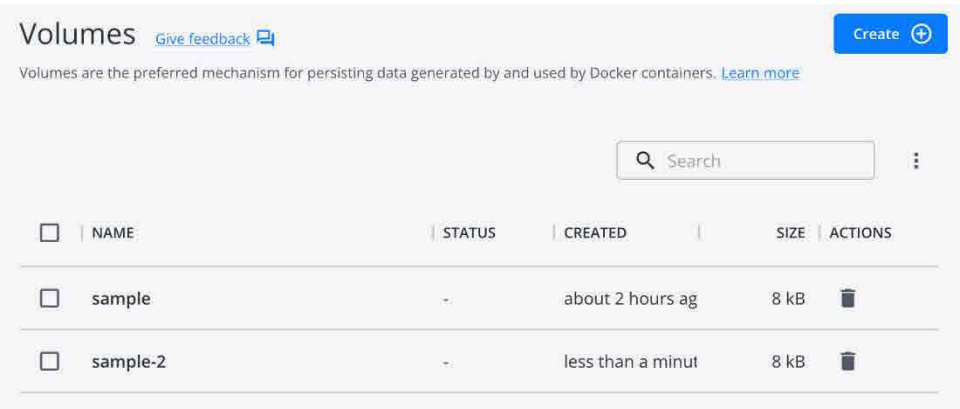


Figure 5.3 – List of Docker volumes shown in Docker Desktop

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.

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 on 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
/ # echo "Some data" > data.txt
/ # 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:

```
Unable to find image 'centos:7' locally
7: Pulling from library/centos
8ba884070f61: Pull complete
Digest: sha256:a799dd8a2ded4a83484bbae769d9765...
Status: Downloaded newer image for centos:7
[root@275c1fe31ec0 /]#
```

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, as follows:

```
[root@275c1fe31ec0 /]# cd /app/data
[root@275c1fe31ec0 /]# ls -l
```

As expected, we should see these two files:

```
-rw-r--r-- 1 root root 10 Dec  4 14:03 data.txt
-rw-r--r-- 1 root root 15 Dec  4 14:03 data2.txt
```

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 *Ctr*l + *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
```

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, 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 whether 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 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:

```
cd: no such file or directory: /var/lib/docker/volumes/
sample/_data
```

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 on 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:

```
/ # ls -l /var/lib/docker/volumes/
total 60
brw-------    1 root     root      254,   1 Dec  4 10:43 backingFsBlockDev
-rw-------    1 root     root         65536 Dec  4 12:46 metadata.db
drwx-----x    3 root     root          4096 Dec  4 10:46 sample
drwx-----x    3 root     root          4096 Dec  4 12:46 sample-2
/ #
```

Figure 5.4 – 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. And then list its content, as follows:

```
/var/lib/docker/volumes/sample/_data # ls -l
```

This should output the following:

```
total 0
```

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
```

This should output the following:

```
total 0
```

Again, this indicates 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 -l
```

This time, you should see this:

```
total 8
-rw-r--r--    1 root     root    10 Dec  4 14:03 data.txt
-rw-r--r--    1 root     root    15 Dec  4 14:03 data2.txt
```

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 content 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:

```
total 12
-rw-r--r--    1 root    root    10 Dec  4 14:03 data.txt
-rw-r--r--    1 root    root    15 Dec  4 14:03 data2.txt
-rw-r--r--    1 root    root    14 Dec  4 14:25 docker.txt
```

15. Let's see whether 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
```

That should output this:

```
total 12
-rw-r--r-- 1 root root 10 Dec  4 14:03 data.txt
-rw-r--r-- 1 root root 15 Dec  4 14:03 data2.txt
-rw-r--r-- 1 root root 14 Dec  4 14:25 docker.txt
```

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.

16. 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:

- From within a container that has a sample volume mounted
- Using a special privileged folder to access the hidden VM used by Docker 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.

> **Race condition**
>
> 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:22.04 /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:

```
total 4
-rw-r--r-- 1 root root 20 Jan 28 22:55 sample.txt
```

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 helpful 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.

Developers use these techniques all the time when they are working on their application that runs in a container and wants to make sure that the container always contains the latest changes 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/ch05
   $ 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:

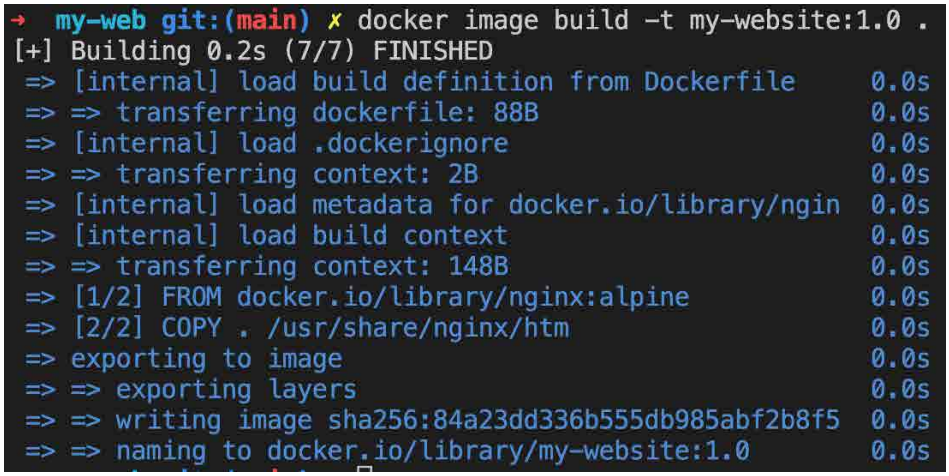


Figure 5.5 – Building a Docker image for a sample Nginx web server

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*, *Using 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</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 having to do this every time that 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**
>
> 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.

11. 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.
12. 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 have learned about containers in *Chapter 4*, *Creating and Managing Container Images*, for more moment, then we have 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. 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 be producing 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 an 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:5.0
```

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 .ContainerConfig.Volumes}}' \
    mongo:5.0 | 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.

> **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:

```
{
    "/data/configdb": {},
    "/data/db": {}
}
```

As we can see, the Dockerfile for MongoDB defines two volumes at `/data/configdb` and `/data/db`.

3. Now, let's run an instance of MongoDB in the background as a daemon, as follows:

```
$ docker run --name my-mongo -d mongo:5.0
```

4. 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 (shortened):

```
➜  ch05 git:(main) ✗ docker inspect --format '{{json .Mounts}}' my-mongo | jq .
[
  {
    "Type": "volume",
    "Name": "8006ec38c77553376c585833c1b8cf900e632e18f7ef722033e77266b632d77b",
    "Source": "/var/lib/docker/volumes/8006ec38c77553376c585833c1b8cf900e632e18f7ef722
033e77266b632d77b/_data",
    "Destination": "/data/configdb",
    "Driver": "local",
    "Mode": "",
    "RW": true,
    "Propagation": ""
  },
  {
    "Type": "volume",
    "Name": "f4f8c0b3a29c74280eb24cc954ddfaeb8abbfa65922712f5cc1dfe022d507089",
```

Figure 5.6 – 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.

Before you leave, clean up the Mongo DB 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, the output is 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
   ```

   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.

   B. Then, list the environment variables we can see there with this command:

   ```
   / # export
   ```

   This should produce the following output:

   ```
   export HOME='/root'
   export HOSTNAME='91250b722bc3'
   export PATH='/usr/local/sbin:/usr/local/bin:...'
   export PWD='/'
   export SHLVL='1'
   export TERM='xterm'
   ```

   The preceding output is different than what we saw directly on the host.

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:

```
export LOG_DIR='/var/log/my-log'
export MAX_LOG_FILES='5'
export MAX_LOG_SIZE='1G'
```

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/ch05
```

2. Create a `config-file` subfolder 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"
```

   And indeed, the variables are defined, as we can see in the output generated:

```
export LOG_DIR='/var/log/my-log'
export MAX_LOG_FILES='5'
export MAX_LOG_SIZE='1G'
```

   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/ch05
```

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**
> Don't forget the period at the end of the preceding line!

5. 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:

```
export LOG_DIR='/var/log/my-log'
export MAX_LOG_FILES='5'
export MAX_LOG_SIZE='1G'
```

This is exactly what we expected.

6. 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"
```

7. Now, have a look at the following command and its output:

```
export LOG_DIR='/var/log/my-log'
export MAX_LOG_FILES='10'
export MAX_LOG_SIZE='2G'
```

8. 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 when 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-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.

And with that, we have come to the end of this chapter.

Summary

In this chapter, we introduced Docker volumes, which can be used to persist the state produced by containers and make them durable. We can also use volumes to provide containers with data originating from various sources. We learned how to create, mount, and use volumes. We also learned various techniques for defining volumes such as by name, by mounting a host directory, or by defining volumes in a container image.

In this chapter, we also discussed how we can configure environment variables that can be used by applications running inside a container. We have shown how to define those variables in the `docker container run` command, either explicitly, one by one, or as a collection in a configuration file. Finally, we learned how to parameterize the build process of container images by using build arguments.

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://dockr.ly/2HxMCjS`
- *Understanding how ARG and FROM interact*: `https://dockr.ly/2OrhZgx`

Questions

Please try to answer the following questions to assess your learning progress:

1. How would you create a named data volume with a name such as `my-products` using the default driver?
2. 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?
3. 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?
4. 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.
5. How would you mount a host volume – for example, `~/my-project` – into a container?
6. How would you remove all unused volumes from your system?
7. 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

8. Your application, which 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. To create a named volume, run the following command:

```
$ docker volume create my-products
```

2. Execute the following command:

```
$ docker container run -it --rm \
    -v my-products:/data:ro \
    alpine /bin/sh
```

3. 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
   ```

   B. This should result in the following output

   ```
   "Mountpoint": "/var/lib/docker/volumes/my-products/_data"
   ```

      i. 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.

4. 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
```

5. Execute a command such as this:

```
$ docker container run -it --rm \
    -v $HOME/my-project:/app/data \
    alpine /bin/sh
```

6. Exit both containers and then, back on the host, execute this command:

```
$ docker volume prune
```

7. The answer is *False* (B). Each container is a sandbox and thus has its very own environment.
8. 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"
```

6
Debugging Code Running in Containers

In the previous chapter, we learned how to work with stateful containers – that is, containers that consume and produce data. We also learned how to configure our containers at runtime and at image build time using environment variables and config files.

In this chapter, we're going to introduce techniques commonly used to allow a developer to evolve, modify, debug, and test their code while it's running in a container. With these techniques at hand, you will enjoy a frictionless development process for applications running in a container, similar to what you experience when developing applications that run natively.

Here is a list of the topics we're going to discuss:

- Evolving and testing code running in a container
- Auto-restarting code upon changes
- Line-by-line code debugging inside a container
- Instrumenting your code to produce meaningful logging information
- Using Jaeger to monitor and troubleshoot

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

- Mount source code residing on the host in a running container
- Configure an application running in a container to auto-restart after a code change
- Configure **Visual Studio Code** (**VS Code**) to debug applications written in Java, Node.js, Python, or .NET running inside a container line by line
- Log important events from your application code
- Configure your multi-component application for distributed tracing using the OpenTracing standard and a tool such as Jaeger

Technical requirements

In this chapter, if you want to follow along with the code, you will need Docker Desktop on macOS or Windows and a code editor – preferably VS Code. The samples will also work on a Linux machine with Docker and VS Code installed.

To prepare your environment for the coming hands-on labs, follow these steps:

1. Please navigate to the folder where you have cloned the sample repository to. Normally, this should be `~/The-Ultimate-Docker-Container-Book`, so do the following:

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

2. Create a new subfolder called `ch06` and navigate to it:

```
$ mkdir ch06 && cd ch06
```

A complete set of sample solutions for all the examples discussed in this chapter can be found in the `sample-solutions/ch06` folder or directly on GitHub: `https://github.com/PacktPublishing/The-Ultimate-Docker-Container-Book/tree/main/sample-solutions/ch06`.

Evolving and testing code running in a container

Make sure you have Node.js and `npm` installed on your computer before you continue. On Mac, use this command:

```
$ brew install node
```

On Windows, use the following command:

```
$ choco install -y nodejs
```

When developing code that will eventually be running in a container, the best approach is often to run the code in the container from the very beginning, to make sure there will be no surprises. But we have to do this in the right way so that we don't introduce any unnecessary friction to our development process. First, let's look at a naïve way we could run and test code in a container. We can do this using a basic Node.js sample application:

1. Create a new project folder and navigate to it:

```
$ mkdir node-sample && cd node-sample
```

2. Let's use `npm` to create a new Node.js project:

```
$ npm init
```

3. Accept all the defaults. Notice that a `package.json` file is created with the following content:

Figure 6.1 – Content of the package.json file of the sample Node.js application

4. We want to use the `Express.js` library in our Node application; thus, use `npm` to install it:

```
$ npm install express -save
```

This will install the newest version of `Express.js` on our machine and, because of the `-save` parameter, add a reference to our `package.json` file that looks similar to this:

```
"dependencies": {
    "express": "^4.18.2"
}
```

Note that in your case, the version number of `express` may be different.

5. Start VS Code from within this folder:

```
$ code .
```

6. In VS Code, create a new file called `index.js` and add this code snippet to it. Do not forget to save:

```
const express = require('express');
const app = express();

app.listen(3000, '0.0.0.0', ()=>{
  console.log('Application listening at 0.0.0.0:3000');
})

app.get('/', (req,res)=>{
  res.send('Sample Application: Hello World!');
})
```

Figure 6.2 – Content of the index.js file of the sample Node.js application

7. From within your terminal window, start the application:

```
$ node index.js
```

> **Note**
> On Windows and Mac, when you execute the preceding command for the first time, a window will pop up, asking you to approve it on the firewall.

You should see this as the output:

```
Application listening at 0.0.0.0:3000
```

This means that the application is running and ready to listen at the `0.0.0.0:3000` endpoint.

> **Tip**
> You might be wondering what the meaning of the host address, `0.0.0.0`, is and why we have chosen it. We will come back to this later when we run the application inside a container. For the moment, just know that `0.0.0.0` is a reserved IP address with a special meaning, similar to the loopback address, `127.0.0.1`. The `0.0.0.0` address simply means all IPv4 addresses on the local machine. If a host has two IP addresses, say `52.11.32.13` and `10.11.0.1`, and a server running on the host listens on `0.0.0.0`, it will be reachable at both of those IPs.

8. Now, open a new tab in your favorite browser and navigate to `http://localhost:3000`. You should see this:

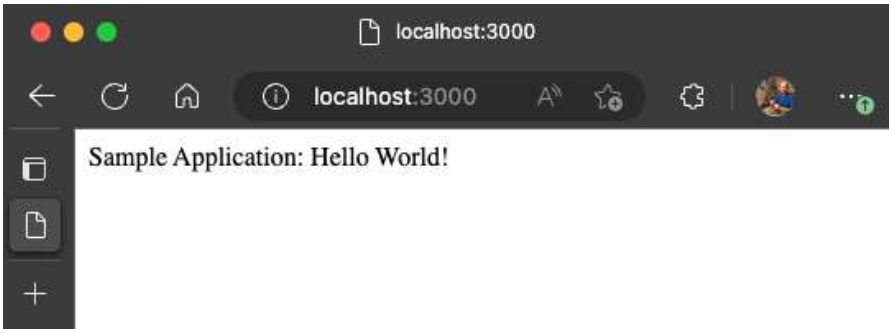


Figure 6.3 – Sample Node.js application running in a browser

Great – our Node.js application is running on our developer machine. Stop the application by pressing *Ctrl* + *C* in the terminal.

9. Now, we want to test the application we have developed so far by running it inside a container. To do this, we must create a Dockerfile so that we can build a container image, from which we can then run a container. Let's use VS Code again to add a file called `Dockerfile` to our project folder and give it the following content:

```
1  FROM node:latest
2  WORKDIR /app
3  COPY package.json ./
4  RUN npm install
5  COPY . .
6  CMD node index.js
```

Figure 6.4 – Dockerfile for the sample Node.js application

10. We can then use this Dockerfile to build an image called `sample-app`, as follows:

```
$ docker image build -t sample-app .
```

It will take a few seconds for the base image to be downloaded and your custom image to be built on top of it.

11. After building, run the application in the container with this command:

```
$ docker container run --rm -it \
    --name my-sample-app \
    -p 3000:3000 \
    sample-app
```

The output will be as follows:

```
Application listening at 0.0.0.0:3000
```

> **Note**
>
> The preceding command runs a container called `my-sample-app` from the `sample-app` container image and maps the container's port, `3000`, to the equivalent host port. This port mapping is necessary; otherwise, we won't be able to access the application running inside the container from outside the container. We will learn more about port mapping in *Chapter 10*, *Using Single-Host Networking*. It is similar to when we ran the application directly on our host.

12. Refresh your previous browser tab (or open a new browser tab and navigate to `localhost:3000`, if you closed it). You should see that the application still runs and produces the same output as when running natively. This is good. We have just shown that our application not only runs on our host but also inside a container.
13. Stop and remove the container by pressing *Ctrl* + *C* in the terminal.
14. Now, let's modify our code and add some additional functionality. We will define another `HTTP GET` endpoint at `/hobbies`. Please add the following code snippet at the end of your `index.js` file:

```
const hobbies = [
    'Swimming', 'Diving', 'Jogging', 'Cooking', 'Singing'
];
app.get('/hobbies', (req,res)=>{
    res.send(hobbies);
})
```

15. We can test the new functionality on our host by running the app with the following command:

```
$ node index.js
```

    Then, we can navigate to `http://localhost:3000/hobbies` in our browser. We should see the expected output – a JSON array with the list of hobbies – in the browser window. Don't forget to stop the application with *Ctrl* + *C* when you've finished testing.

16. Next, we need to test the code when it runs inside the container. So, first, we must create a new version of the container image:

```
$ docker image build -t sample-app .
```

    This time, the build should be quicker than the first time we did this since the base image is already in our local cache.

17. Next, we must run a container from this new image:

```
$ docker container run --rm -it \
    --name my-sample-app \
    -p 3000:3000 \
    sample-app
```

18. Now, we can navigate to `http://localhost:3000/hobbies` in our browser and confirm that the application works as expected inside the container too.
19. Once again, don't forget to stop the container when you're done by pressing *Ctrl* + *C*.

We can repeat this sequence of tasks over and over again for each feature we add or any existing features we improve. It turns out that this is a lot of added friction compared to times when all the applications we developed always ran directly on the host.

However, we can do better. In the next section, we will look at a technique that allows us to remove most of this friction.

Mounting evolving code into the running container

What if, after a code change, we do not have to rebuild the container image and rerun a container? Wouldn't it be great if the changes would immediately, as we save them in an editor such as VS Code, be available inside the container too? Well, that is possible with volume mapping. In the previous chapter, we learned how to map an arbitrary host folder to an arbitrary location inside a container. We want to leverage that in this section. In *Chapter 5*, *Data Volumes and Configuration*, we learned how to map host folders as volumes in a container. For example, if we want to mount a host folder, `/projects/sample-app`, into a container at `/app`, the syntax for this will look as follows:

```
$ docker container run --rm -it \
    --volume /projects/sample-app:/app \
    alpine /bin/sh
```

Notice the `--volume <host-folder>:<container-folder>` line. The path to the host folder needs to be an absolute path, which in this example is `/projects/sample-app`.

Now, if we want to run a container from our `sample-app` container image and we do that from the project folder, we can map the current folder to the `/app` folder of the container, as follows:

```
$ docker container run --rm -it \
    --volume $(pwd):/app \
    -p 3000:3000 \
    sample-app
```

> **Note**
>
> Please note `$(pwd)` in place of the host folder path. `$(pwd)` equals the absolute path of the current folder, which comes in very handy.

Now, if we use the above volume mapping parameter, then whatever was in the `/app` folder of the `sample-app` container image will be overridden by the content of the mapped host folder, which in our case is the current folder. That's exactly what we want – we want the current source to be mapped from the host into the container. Let's test whether it works:

1. Stop the container if you have started it by pressing *Ctrl + C*.
2. Then, add the following snippet to the end of the `index.js` file:

```
app.get('/status', (req,res)=>{
    res.send('OK');
})
```

   Do not forget to save.

3. Then, run the container again – this time, without rebuilding the image first – to see what happens:

```
$ docker container run --rm -it \
    --name my-sample-app \
    --volume $(pwd):/app \
    -p 3000:3000 \
    sample-app
```

4. In your browser, navigate to `http://localhost:3000/status`. You will see the `OK` output in your browser window. Alternatively, instead of using your browser, you could use `curl` in another terminal window to probe the `/status` endpoint, as follows:

```
$ curl localhost:3000/status
OK
```

> **Note**
> For all those working on Windows and/or Docker Desktop for Windows, you can use the PowerShell `Invoke-WebRequest` command or `iwr` for short instead of `curl`. In this case, the equivalent to the preceding command would be `PS> iwr -Url http://localhost:3000/status`.

5. Leave the application in the container running for the moment and make yet another change. Instead of just returning `OK` when navigating to `/status`, we want a message stating `OK, all good` to be returned. Make your modification and save your changes.
6. Then, execute the `curl` command again or, if you did use your browser, refresh the page. What do you see? Right – nothing happened. The change we made is not reflected in the running application.
7. Well, let's double-check whether the change has been propagated in the running container. To do this, let's execute the following command:

```
$ docker container exec my-sample-app cat index.js
```

This executes the `cat index.js` command inside our already running container. We should see something like this – I have shortened the output for readability:

```
...
app.get('/hobbies', (req,res)=>{
    res.send(hobbies);
})
app.get('/status', (req,res)=>{
    res.send('OK, all good');
})
...
```

As we can see, our changes have been propagated into the container as expected. Why, then, are the changes not reflected in the running application? Well, the answer is simple: for changes to be applied to the application, the Node.js sample application has to be restarted.

8. Let's try that. Stop the container with the application running by pressing *Ctrl* + *C*. Then, re-execute the preceding `docker container run` command and use `curl` to probe the `http://localhost:3000/status` endpoint. This time, the following new message should be displayed:

```
$ curl http://localhost:3000/status
OK, all good
```

With that, we have significantly reduced the friction in the development process by mapping the source code in the running container. We can now add new codeor modify existing code and test it without having to build the container image first. However, a bit of friction has been left in play. We have to manually restart the container every time we want to test some new or modified code. Can we automate this? The answer is yes! We will demonstrate exactly this in the next section.

Auto-restarting code upon changes

In the previous section, we showed you how we can massively reduce friction by volume mapping the source code folder in the container, thus avoiding having to rebuild the container image and rerun the container over and over again. Yet we still feel some remaining friction. The application running inside the container does not automatically restart when a code change is made. Thus, we have to manually stop and restart the container to pick up these new changes.

In this section, we will learn how we can containerize our applications written in various languages, such as Node.js, Java, Python, and .NET, and have them restart automatically whenever a code change is detected. Let's start with Node.js.

Auto-restarting for Node.js

If you have been coding for a while, you will certainly have heard about helpful tools that can run your applications and restart them automatically whenever they discover a change in the code base. For Node.js applications, the most popular tool is `nodemon`. Let's take a look:

1. We can install `nodemon` globally on our system with the following command:

```
$ npm install -g nodemon
```

2. Now that `nodemon` is available, instead of starting our application (for example, on the host) with `node index.js`, we can just execute `nodemon` and we should see the following:

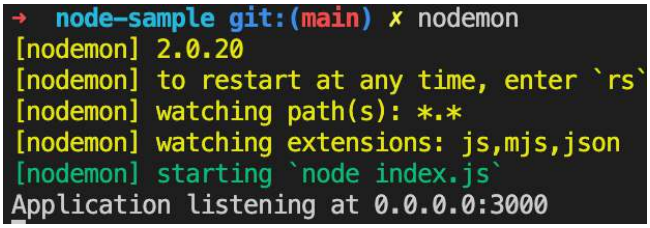


Figure 6.5 – Running our Node.js sample application with nodemon

> **Note**
>
> As we can see, from parsing our `package.json` file, `nodemon` has recognized that it should use `node index.js` as the starting command.

3. Now, try to change some code. For this example, add the following code snippet to the end of `index.js` and then save the file:

```
app.get('/colors', (req,res)=>{
    res.send(['red','green','blue']);
})
```

4. Look at the terminal window. Did you see something happen? You should see this additional output:

```
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Application listening at 0.0.0.0:3000
```

This indicates that `nodemon` has recognized some changes and automatically restarted the application.

5. Try this out on your browser by navigating to `localhost:3000/colors`. You should see the following expected output in your browser:

```
["red", "green", "blue"]
```

This is cool – you got this result without having to manually restart the application. This makes us yet another bit more productive. Now, can we do the same within the container?

Yes, we can. However, we won't use the start command, `node index.js`, as defined in the last line of our Dockerfile:

```
CMD node index.js
```

We will use `nodemon` instead.

Do we have to modify our Dockerfile? Or do we need two different Dockerfiles, one for development and one for production?

Our original Dockerfile creates an image that unfortunately does not contain `nodemon`. Thus, we need to create a new Dockerfile:

1. Create a new file. Let's call it `Dockerfile.dev`. Its content should look like this:

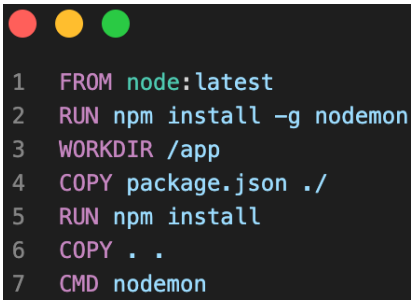


Figure 6.6 – Dockerfile used for developing our Node.js application

   Comparing this with our original `Dockerfile`, we have added line 2, where we install `nodemon`. We have also changed the last line and are now using `nodemon` as our start command.

2. Let's build our development image, as follows:

```
$ docker image build \
    -f Dockerfile.dev \
    -t node-demo-dev .
```

   Please note the `-f Dockerfile.dev` command-line parameter. We must use this since we are using a Dockerfile with a non-standard name.

3. Run a container, like this:

```
$ docker container run --rm -it \
    -v $(pwd):/app \
    -p 3000:3000 \
    node-demo-dev
```

4. Now, while the application is running in the container, change some code, save it, and notice that the application inside the container is automatically restarted. With this, we have achieved the same reduction in friction while running in a container as we did when running directly on the host.
5. Hit *Ctrl* + *C* when you're done to exit your container.
6. Use the following command to clean up your system and remove all running or dangling containers:

```
$ docker container rm -f $(docker container ls -aq)
```

You might be wondering, does this only apply to Node.js? No – fortunately, many popular languages support similar concepts.

Auto-restarting for Java and Spring Boot

Java and Spring Boot are still by far the most popular programming languages and libraries when developing **line of business** (**LOB**) type applications. Let's learn how to work as friction-free as possible when developing such an application and containerizing it.

For this example to work, you have to have Java installed on your computer. At the time of writing, the recommended version is Java 17. Use your favorite package manager to do so, such as Homebrew on Mac or Chocolatey on Windows.

You may also want to make sure you have *Extension Pack for Java* by Microsoft installed for VS Code. You can find more details here: `https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-pack`.

Once you have the Java 17 SDK installed and ready on your computer, proceed as follows:

1. The easiest way to bootstrap a Spring Boot application is by using the **Spring Initializr** page:
    I. Navigate to `https://start.spring.io`.
    II. Under **Project**, select **Maven**.
    III. Under **Language**, select **Java**.
    IV. Under **Spring Boot**, select **3.0.2** (or newer if available at the time of writing).
    V. For **Packaging**, select **Jar**.
    VI. Finally, for **Java**, select **17**.
    VII. Click **ADD DEPENDENCIES**, search for `Spring Web`, and select it (*do not* select **Spring Reactive Web**).

    Your page should look like this:

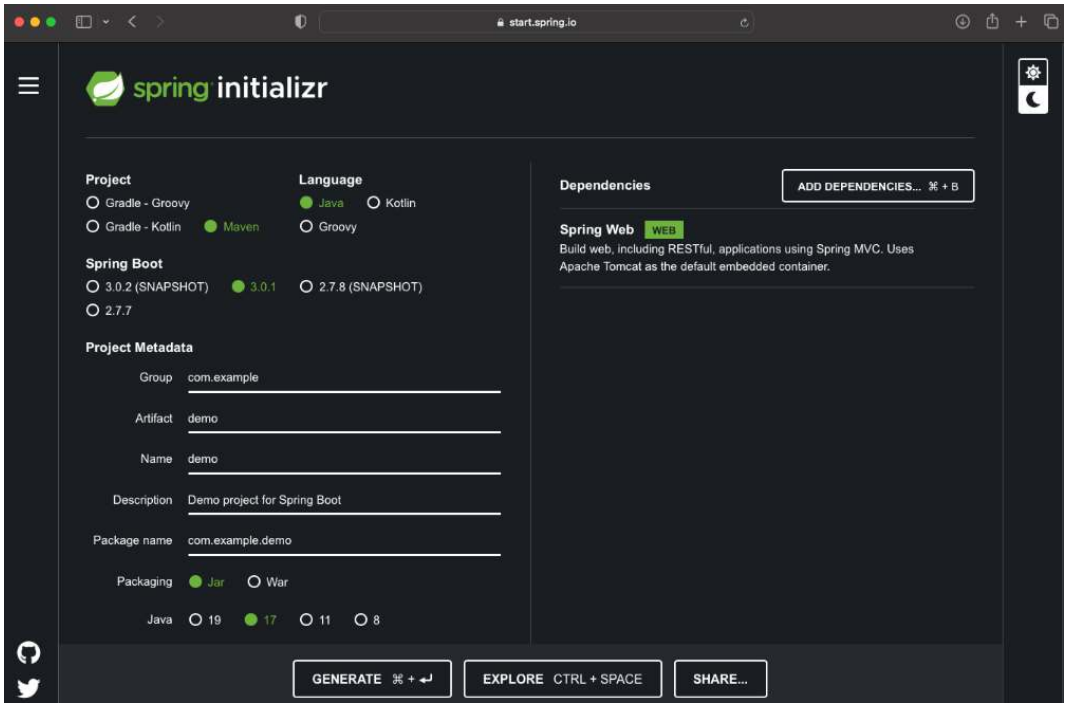


Figure 6.7 – Bootstrapping a new Java project with Spring Initializr

2. Click **GENERATE** and unpack the resulting ZIP file into a folder called `ch06/java-springboot-demo`.
3. Navigate to this folder:

```
$ cd ch06/java-springboot-demo
```

4. Open VS Code from within this folder by using the following command:

```
$ code .
```

5. Locate the main file of the project, which is called `DemoApplication.java`, and click on the **Run** hyperlink directly above the `main` method on line 9, as shown in the following screenshot:

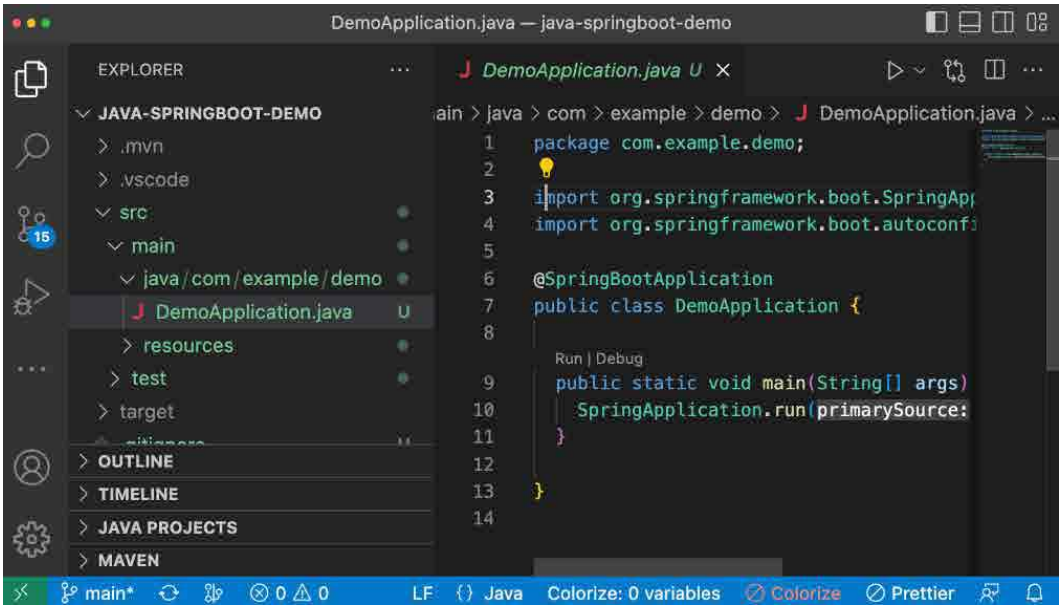


Figure 6.8 – Starting the Java Spring Boot application

6. Observe that the application has been compiled and that a terminal window opens. Content similar to the following will be displayed:

```
→ java-springboot-demo git:(main) ✗ >...
ntents/Home/bin/java @/var/folders/lt/8ss0zp4j4yl4q2z0700j6mq80000gn/T/cp_17r311t0xr7utg0jum48lxtsl.argfile com.example.demo.DemoApplication

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.0.1)

2022-12-27T13:03:12.923+01:00  INFO 14717 --- [           main] com.example.demo.DemoApplication         : Starting DemoApplication using Java 17.0.5 with PID 14717 (/Users/gabriel/The-Ultimate-Docker-Container-Book/sample-solutions/ch06/java-springboot-demo/target/classes started by gabriel in /Users/gabriel/The-Ultimate-Docker-Container-Book/sample-solutions/ch06/java-springboot-demo)
2022-12-27T13:03:12.925+01:00  INFO 14717 --- [           main] com.example.demo.DemoApplication         : No active profile set, falling back to 1 default profile: "default"
2022-12-27T13:03:13.314+01:00  INFO 14717 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2022-12-27T13:03:13.320+01:00  INFO 14717 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2022-12-27T13:03:13.320+01:00  INFO 14717 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.4]
2022-12-27T13:03:13.379+01:00  INFO 14717 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2022-12-27T13:03:13.380+01:00  INFO 14717 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 432 ms
2022-12-27T13:03:13.562+01:00  INFO 14717 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2022-12-27T13:03:13.568+01:00  INFO 14717 --- [           main] com.example.demo.DemoApplication         : Started DemoApplication in 0.823 seconds (process running for 1.11)
```

Figure 6.9 – Output generated by a running Spring Boot application

7. On the second to last line of the preceding output, we can see that the application uses the Tomcat web server and is listening at port `8080`.

8. Now, let's add an endpoint that we can then try to access:

   I. Decorate the `DemoApplication` class with a `@RestController` annotation.

   II. Add a `getSpecies` method that returns a list of strings

   III. Decorate the method with the following annotation:

   ```
   @GetMapping("/species")
   ```

   Don't forget to add the required `import` statements. The complete code will look like this:

```
1  package com.example.demo;
2
3  import java.util.List;
4
5  import org.springframework.boot.SpringApplication;
6  import org.springframework.boot.autoconfigure.SpringBootApplication;
7  import org.springframework.web.bind.annotation.GetMapping;
8  import org.springframework.web.bind.annotation.RestController;
9
10 @SpringBootApplication
11 @RestController
12 public class DemoApplication {
13
14   public static void main(String[] args) {
15     SpringApplication.run(DemoApplication.class, args);
16   }
17
18   @GetMapping("/species")
19   public List<String> getSpecies() {
20     return List.of("Elephant", "Mouse", "Cat");
21   }
22 }
23
```

Figure 6.10 – Complete demo code for the Spring Boot example

9. Use `curl` or the **Thunder Client** plugin for VS Code to try and access the `/species` endpoint:

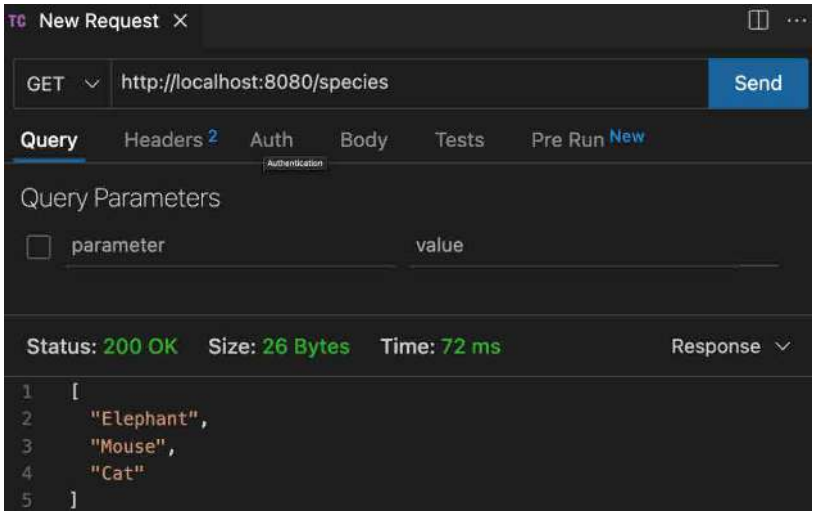


Figure 6.11 – Using the Thunder Client plugin to test the Java demo application

10. To add auto-restart support to our Java Spring Boot application, we need to add the so-called dev tools:

    I. Locate the `pom.xml` file in your Java project and open it in the editor.

    II. Add the following snippet to the dependencies section of the file:

```
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <version>3.0.1</version>
      <optional>true</optional>
    </dependency>

  </dependencies>
```

Figure 6.12 – Adding a reference to the Spring Boot dev tools

Note that the version node in the dependency definition can be omitted as the project uses `spring-boot-starter-parent` as the parent.

11. Stop and rerun the application.
12. Modify line 20 of the `DemoApplication` class and add `Crocodile` as a fourth species to return to the caller.
13. Save your changes and observe that the application automatically rebuilds and restarts.
14. Use `curl` or Thunder Client again to access the `/species` endpoint. This time, a list of four species should be returned, including the just-added `Crocodile`.

Great – we have a Java Spring Boot application that automatically re-compiles and restarts when we change any code in it. Now, we need to dockerize the whole thing, as we did with the Node.js example:

1. Add a `Dockerfile` to the root of the project with the following content:

```
1  FROM eclipse-temurin:17-jdk-focal AS build
2  WORKDIR /app
3  COPY mvnw .
4  COPY .mvn .mvn
5  COPY pom.xml .
6  COPY src src
7
8  CMD ./mvnw spring-boot:run
```

Figure 6.13 – Dockerfile for the Java Spring Boot demo

Note
We have used the `eclipse-temurin` image with the `17-jdk-focal` tag for this example since this image, at the time of writing, works on the M1 or M2 processor used by modern MacBooks.

2. Create an image using the preceding Dockerfile with this command:

```
$ docker image build -t java-demo .
```

3. Create a container from this Docker image with the following command:

```
$ docker container run --name java-demo --rm \
    -p 8080:8080
    -v $(pwd)/.:/app
    java-demo
```

> **Note**
> The first time you run the container, it will take a while to compile since all the Maven dependencies need to be downloaded.

4. Try to access the `/species` endpoint, as you did previously.
5. Now, change some code – for example, add a fifth species to be returned to the `getSpecies` method, such as `Penguin`, and then save your changes.
6. Observe how the application running inside the container is rebuilt. Verify that the change has been incorporated by accessing the `/species` endpoint once again and asserting that five species are returned, including `Penguin`.
7. When you're done playing around, stop the container either via the dashboard of Docker Desktop or the Docker plugin in VS Code.

Well, that was quite straightforward, wasn't it? But let me tell you, setting up your development environment this way can make developing containerized applications much more enjoyable by eliminating much of the unnecessary friction.

> **Challenge**
> Try to find out how you could map your local Maven cache into the container, to accelerate the first startup of the container even further.

Next, we are going to show you how easy it is to do the same exercise in Python. Stay tuned.

Auto-restarting for Python

Let's look at how the same thing works for Python.

Prerequisites

For this example to work, you need to have Python 3.x installed on your computer. You can do this using your preferred package manager, such as Homebrew on Mac or Chocolatey on Windows.

On your Mac, use this command to install the latest Python version:

```
$ brew install python
```

On your Windows computer, use this command to do the same:

```
$ choco install python
```

Use this command to verify that the installation was successful:

```
$ python3 --version
```

In the author's case, the output looks like this:

```
Python 3.10.8
```

Let's begin:

1. First, create a new project folder for our sample Python application and navigate to it:

```
$ mkdir python-demo && cd python-demo
```

2. Open VS Code from within this folder by using the following command:

```
$ code .
```

3. We will create a sample Python application that uses the popular Flask library. Thus, add a file to this folder called `requirements.txt` that contains this content:

```
flask
```

4. Next, add a `main.py` file and give it this content:

```
1  from flask import Flask
2  app = Flask(__name__)
3  @app.route("/")
4  def hello():
5      return "Hello World!"
6
7  if __name__ == "__main__":
8      app.run(host="0.0.0.0", port=5000)
```

Figure 6.14 – Content of the main.py file of our sample Python application

This is a simple Hello World-type app that implements a single RESTful endpoint at `http://localhost:5000/`.

Note

The `host="0.0.0.0"` parameter in the `app.run` command is needed so that we can expose the port on which the Python app is listening (`5000`) to the host. We will need this later in this example.

Please also note that some people have reported that, when running on a Mac and using port `5000`, an error stating "Address already in use. Port `5000` is in use by another program..." is triggered. In this case, just try to use a different port, such as `5001`.

5. Before we can run and test this application, we need to install the necessary dependencies – in our case, Flask. In the terminal, run the following command:

```
$ pip3 install -r requirements.txt
```

This should install Flask on your host. We are now ready to go.

6. When using Python, we can also use `nodemon` to have our application auto-restart when any changes are made to the code. For example, assume that your command to start the Python application is `python main.py`. In this case, you would just use `nodemon` like so:

```
$ nodemon --exec python3 main.py
```

You should see the following output:

```
➜  python-demo git:(main) ✗ nodemon --exec python3 main.py
[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: py,json
[nodemon] starting `python3 main.py`
 * Serving Flask app 'main'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

Figure 6.15 – Using nodemon to auto-restart a Python 3 application

7. When using `nodemon` to start and monitor a Python application, we can test the application by using `curl`. Open another terminal window and enter this:

```
$ curl localhost:5000
```

You should see this in the output:

```
Hello World!
```

8. Now, let's modify the code by adding the following snippet to `main.py`, right after the definition of the `/` endpoint (that is, right after line 5), and save it:

```
from flask import jsonify
@app.route("/colors")
def colors():
    return jsonify(["red", "green", "blue"])
```

`nodemon` will discover the changes and restart the Python app, as we can see in the output produced in the terminal:

```
[nodemon] restarting due to changes...
[nodemon] starting `python3 main.py`
 * Serving Flask app 'main'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

Figure 6.16 – nodemon discovering a change in the Python code

9. Once again, believing is good, but testing is better. Thus, let's use our friend `curl` once again to probe the new endpoint and see what we get:

```
$ curl localhost:5000/colors
```

The output should look like this:

```
["red", "green", "blue"]
```

Nice – it works! With that, we have covered Python.

10. Now, it's time to containerize this application. Add a file called `Dockerfile` to the project with the following content:

```
1  FROM nikolaik/python-nodejs:latest
2  RUN npm install -g nodemon
3  WORKDIR /app
4  COPY requirements.txt .
5  RUN pip install -r requirements.txt
6  COPY . .
7  CMD [ "nodemon", "--exec", "python", "main.py" ]
```

Figure 6.17 – Dockerfile for the sample Python application

Note that on line 1, we are using a special base image that contains both Python and Node.js code. Then, on line 2, we install the `nodemon` tool before we copy the `requirements.txt` file into the container and execute the `pip install` command. Next, we copy all other files into the container and define the start command for whenever an instance of this image – that is, a container – is created.

11. Let's build a Docker image with this command:

```
$ docker image build -t python-sample .
```

12. Now, we can run a container from this image with the following code:

```
$ docker container run --rm \
    -p 5000:5000 \
    -v $(pwd)/.:/app \
    python-sample
```

We should have an output similar to what was produced by the application running inside the container in *step 6*, where we ran the application natively:

```
[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: py,json
[nodemon] starting `python main.py`
 * Serving Flask app 'main'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment.
erver instead.
 * Running on  Follow link (cmd + click)
 * Running on http://127.0.0.1:5000
 * Running on http://172.17.0.2:5000
Press CTRL+C to quit
```

Figure 6.18 – Running the containerized Python sample application

Note how we have mapped the container port, `5000`, to the equivalent host port so that we can access the application from outside. We have also mapped the content of the sample directory on the host to the `/app` folder inside the running container. This way, we can update the code and the containerized application will automatically restart.

13. Try to change the application code, and return a fourth color when the `/colors` endpoint is hit. Save the change and observe how the application running inside the container is restarted.
14. Use the `curl` command to verify that an array of four colors is returned.
15. When you're done playing with this example, hit *Ctrl* + *C* in the terminal window where you have the container running to stop the application and the container.

With this, we have shown a fully working example for Python that helps you massively reduce the friction of working with containers during the development process.

.NET is another popular platform. Let's see if we can do something like this when developing a C# application on .NET.

Auto-restarting for .NET

Our next candidate is a .NET application written in C#. Let's look at how dynamic code updates and auto-restarts work in .NET.

Prerequisites

If you have not done so before, please install .NET on your laptop or workstation. You can use your favorite package manager, such as Homebrew on Mac or Chocolatey on Windows, to do so.

On Mac, use this command to install the .NET 7 SDK:

```
$ brew install --cask dotnet-sdk
```

On a Windows machine, you can use this command:

```
$ choco install -y dotnet-sdk
```

Finally, use this command to verify your installation:

```
$ dotnet –version
```

On the author's machine, the output is as follows:

```
7.0.100
```

Let's begin:

1. In a new terminal window, navigate to this chapter's folder:

   ```
   $ cd ~/The-Ultimate-Docker-Container-Book/ch06
   ```

2. From within this folder, use the `dotnet` tool to create a new Web API and have it placed in the `dotnet` subfolder:

   ```
   $ dotnet new webapi -o csharp-sample
   ```

3. Navigate to this new project folder:

   ```
   $ cd csharp-sample
   ```

4. Open VS Code from within this folder:

   ```
   $ code .
   ```

Note

If this is the first time you have opened a .NET project with VS Code, then the editor may display a popup asking you to add the missing dependencies for our `dotnet` project. Click the **Yes** button in this case:

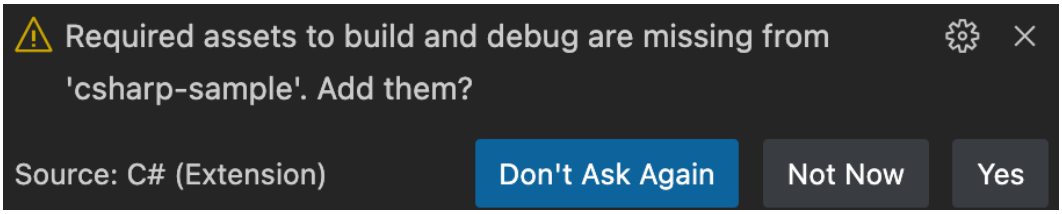


Figure 6.19 – Request to load missing assets for the .NET sample application

5. In the Project Explorer of VS Code, you should see this:

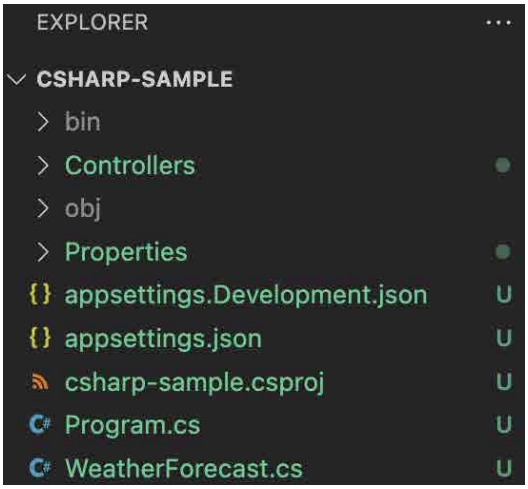


Figure 6.20 – The .NET sample application in the VS Code Project Explorer

6. Please note the `Controllers` folder with the `WeatherForecastController.cs` file in it. Open this file and analyze its content. It contains the definition for the `WeatherForecastController` class, which implements a simple RESTful controller with a GET endpoint at `/WeatherForecast`.
7. From your terminal, run the application with `dotnet run`. You should see something like this:

```
➜  csharp-sample git:(main) ✗ dotnet run
Building...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5080
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: /Users/gabriel/The-Ultimate-Docker-Container-Book/ch06/csharp-sample
```

Figure 6.21 – Running the .NET sample Web API on the host

Please note the fourth line in the above output, where .NET tells us that the application is listening at `http://localhost:5080`. In your case, the port may be a different one. Use the one reported for you for all subsequent steps.

8. We can use `curl` to test the application, like so:

```
$ curl http://localhost:5080/WeatherForecast
```

This will output an array of five JSON objects containing random weather data:

```
➜  csharp-sample git:(main) ✗ curl http://localhost:5080/weatherforecast
[{"date":"2022-12-11","temperatureC":14,"temperatureF":57,"summary":"Balmy"}
,{"date":"2022-12-12","temperatureC":53,"temperatureF":127,"summary":"Hot"},
{"date":"2022-12-13","temperatureC":6,"temperatureF":42,"summary":"Warm"},{"
date":"2022-12-14","temperatureC":-17,"temperatureF":2,"summary":"Chilly"},{
"date":"2022-12-15","temperatureC":32,"temperatureF":89,"summary":"Cool"}]%
```

Figure 6.22 – Weather data produced by the .NET sample application

9. We can now try to modify the code in `WeatherForecastController.cs` and return, say, 10 instead of the default 5 items. Change line 24 so that it looks like this:

```
...
    return Enumerable.Range(1, 10).Select(...
...
```

10. Save your changes and rerun the `curl` command. Notice how the result does not contain the newly added value. This is the same problem that we observed for Node.js and Python. To see the newly updated return value, we need to (manually) restart the application.

11. Thus, in your terminal, stop the application with *Ctrl* + *C* and restart it with `dotnet run`. Try the `curl` command again. The result should now reflect your changes.

12. Luckily for us, the `dotnet` tool has the `watch` command. Stop the application by pressing *Ctrl* + *C* and execute this slightly modified command:

```
$ dotnet watch run
```

You should see output resembling the following (shortened):

```
dotnet watch 🚀 Started
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5080
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: /Users/gabriel/The-Ultimate-Docker-Container-Book/ch06/csharp-sample
```

Figure 6.23 – Running the .NET sample application with the watch task

Notice the first line in the preceding output, which states that the running application is now watched for changes.

13. Make another change in `WeatherForecastController.cs`; for example, make the `GET` endpoint method return 100 weather items and then save your changes. Observe the output in the terminal. It should look something like this:

```
dotnet watch 🕒 File changed: ./Controllers/WeatherForecastController.cs.
dotnet watch 🔥 Hot reload of changes succeeded.
```

Figure 6.24 – Auto-restarting the running sample .NET Core application

14. By automatically restarting the application upon making changes to the code, the result is immediately available to us, and we can easily test it by running the following `curl` command:

```
$ curl http://localhost:5080/WeatherForecast
```

100 instead of 10 weather items should be output this time.

15. Now that we have auto-restart working on the host, we can author a `Dockerfile` that does the same for the application running inside a container. In VS Code, add a new file called `Dockerfile-dev` to the project and add the following content to it:

```
1  FROM mcr.microsoft.com/dotnet/sdk:7.0
2  WORKDIR /app
3  COPY csharp-sample.csproj ./
4  RUN dotnet restore
5  COPY . .
6  CMD dotnet watch run --urls "http://0.0.0.0:5000"
```

Figure 6.25 – Dockerfile for the .NET sample application

Note the `--urls` command-line parameter on line 6. This explicitly tells the application to listen on port `5000` at all endpoints inside the container (denoted by the special `0.0.0.0` IP address). If we were to leave the default of `localhost`, then we wouldn't be able to reach the application from outside the container.

Port is already in use

Please note that some people have reported that when running on a Mac and using port `5000`, an error stating "Address already in use. Port `5000` is in use by another program..." is triggered. In this case, just try to use a different port, such as `5001`.

Now, we're ready to build the container image:

1. Use the following command to build a container image for the .NET sample:

```
$ docker image build -f Dockerfile-dev \
    -t csharp-sample .
```

2. Once the image has been built, we can run a container from it:

```
$ docker container run --rm \
    --name csharp-sample \
    -p 5000:5000 \
    -v $(pwd):/app \
    csharp-sample
```

   We should see a similar output to what we saw when running natively.

3. Let's test the application with our friend, `curl`:

```
$ curl localhost:5000/weatherforecast
```

   We should get the array of weather forecast items. No surprises here – it works as expected.

4. Now, let's make a code change in the controller and save it. Observe what's happening in the terminal window. We should see an output like this:

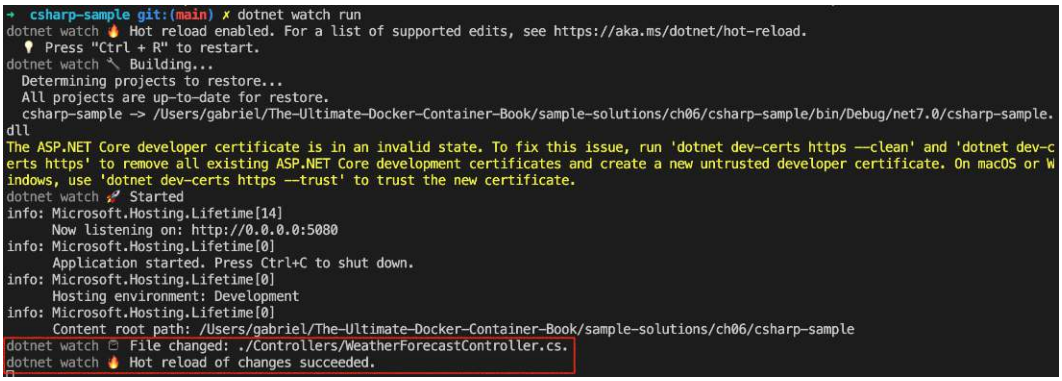


Figure 6.26 – Hot reloading the .NET sample application running inside a container

Well, that's exactly what we expected. With this, we have removed most of the friction that we introduced by using containers when developing a .NET application.

5. When you're done playing with the .NET sample application, open the dashboard of your Docker Desktop application. Locate the `csharp-sample` container and select it. Then, click the red **Delete** button to remove it from your system. This is the easiest way to do this since, unfortunately, just pressing *Ctrl* + *C* in the terminal window where you ran the container does not work. Alternatively, you can open another terminal window and use this command to get rid of the container:

```
$ docker container rm --force csharp-sample
```

That's it for now. In this section, we explored how we can reduce friction during development when working with containerized applications written in Node.js, Python, Spring Boot, Java, or .NET. Next, we are going to learn how we can debug an application running in a container line by line.

Line-by-line code debugging inside a container

Before we dive into this section about debugging code running inside a container line by line, let me make a disclaimer. What you will learn in this section should usually be your last resort if nothing else works. Ideally, when following a test-driven approach when developing your application, the code is mostly guaranteed to work since you have written unit and integration tests for it and run them against your code, which also runs in a container. Alternatively, if unit or integration tests don't provide you with enough insight and you need to debug your code line by line, you can do so by running your code directly on your host, thus leveraging the support of development environments such as VS Code, Eclipse, or IntelliJ, to name just a few IDEs.

With all this preparation, you should rarely need to manually debug your code as it is running inside a container. That said, let's see how you can do it anyways!

In this section, we are going to concentrate exclusively on how to debug when using VS Code. Other editors and IDEs may or may not offer similar capabilities.

Debugging a Node.js application

We'll start with the easiest one – a Node.js application. We will use our sample application in the `~/The-Ultimate-Docker-Container-Book/ch06/node-sample` folder, which we worked with earlier in this chapter:

1. Open a new terminal window and make sure that you navigate to this project folder:

```
$ cd ~/The-Ultimate-Docker-Container-Book/ch06/node-sample
```

2. Open VS Code from within this container:

```
$ code .
```

3. In the terminal window, from within the project folder, run a container with our sample Node.js application:

```
$ docker container run --rm -it \
    --name node-sample \
    -p 3000:3000 \
    -p 9229:9229 \
    -v $(pwd):/app \
    node-demo-dev node --inspect=0.0.0.0 index.js
```

> **Note**
>
> In the preceding command, we mapped port `9229` to the host. This port is used by the Node.js debugger, and VS Studio will communicate with our Node application via this port. Thus, it is important that you open this port – but only during a debugging session! Also, note that we overrode the standard start command that we defined in the Dockerfile (remember, it was just `node index.js`) with `node --inspect=0.0.0.0 index.js`. The `--inspect=0.0.0.0` command-line parameter tells Node to run in debug mode and listen on all IPv4 addresses in the container.

Now, we are ready to define a VS Code launch task for the scenario at hand – that is, our code running inside a container.

4. Add a folder called `.vscode` to your project (please note the leading period in the name of the folder). Within this folder, add a file called `launch.json` with the following content:

```
{
  "configurations": [
    {
      "name": "Attach to Node JS",
      "type": "node",
      "port": 9229,
      "request": "attach",
      "remoteRoot": "/app"
    }
  ]
}
```

Figure 6.27 – The launch configuration to debug a Node.js application

5. To open the `launch.json` file, press *cmd* + *Shift* + *P* (or *Ctrl* + *Shift* + *P* on Windows) to open the command palette; look for **Debug:Open launch.json** and select it. The `launch.json` file should open in the editor.

6. Open the `index.js` file and click on the left sidebar on line 25 to set a breakpoint:

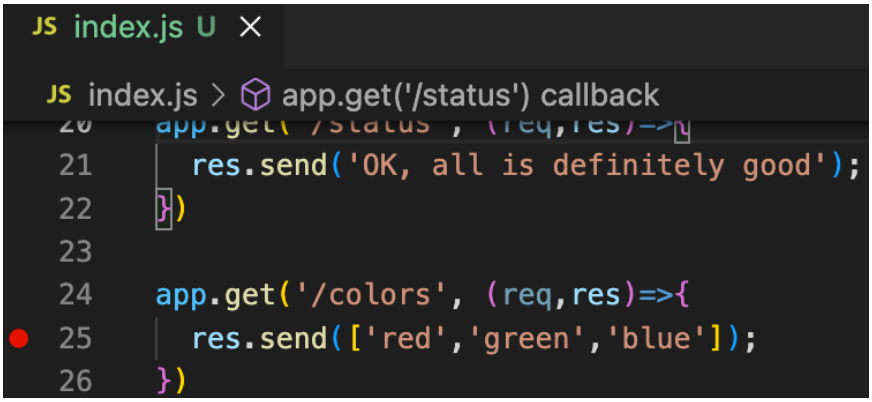


Figure 6.28 – Setting a breakpoint in our Node.js sample application

7. Open the Debug view in VS Code by pressing *cmd* + *Shift* + *D* (or *Ctrl* + *Shift* + *D* on Windows).
8. Make sure you select the correct launch task in the dropdown next to the green start button at the top of the view. Select **Attach to Node JS** and specify the name of the launch configuration in the `launch.json` file. It should look like this:

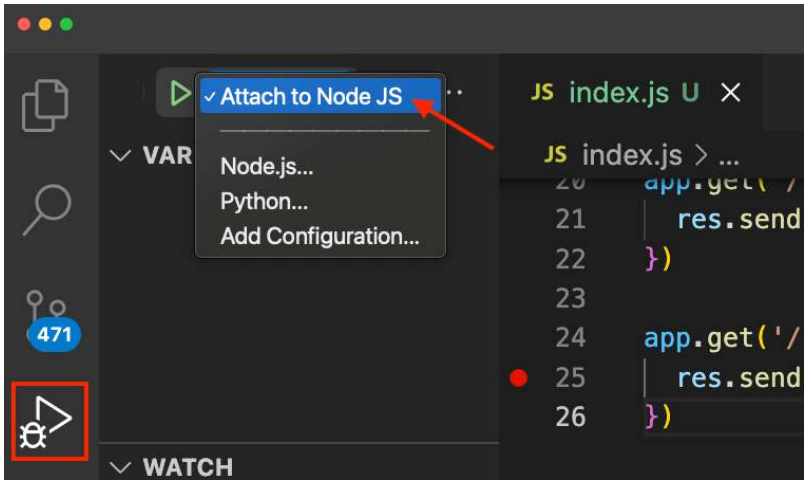


Figure 6.29 – Selecting the correct launch task to debug our Node.js application

9. Next, click on the green start button to attach VS Code to the Node.js application running in the container.
10. In another terminal window, use `curl` to navigate to the `/colors` endpoint:

```
$ curl localhost:3000/colors
```

Observe that the code's execution stops at the breakpoint:

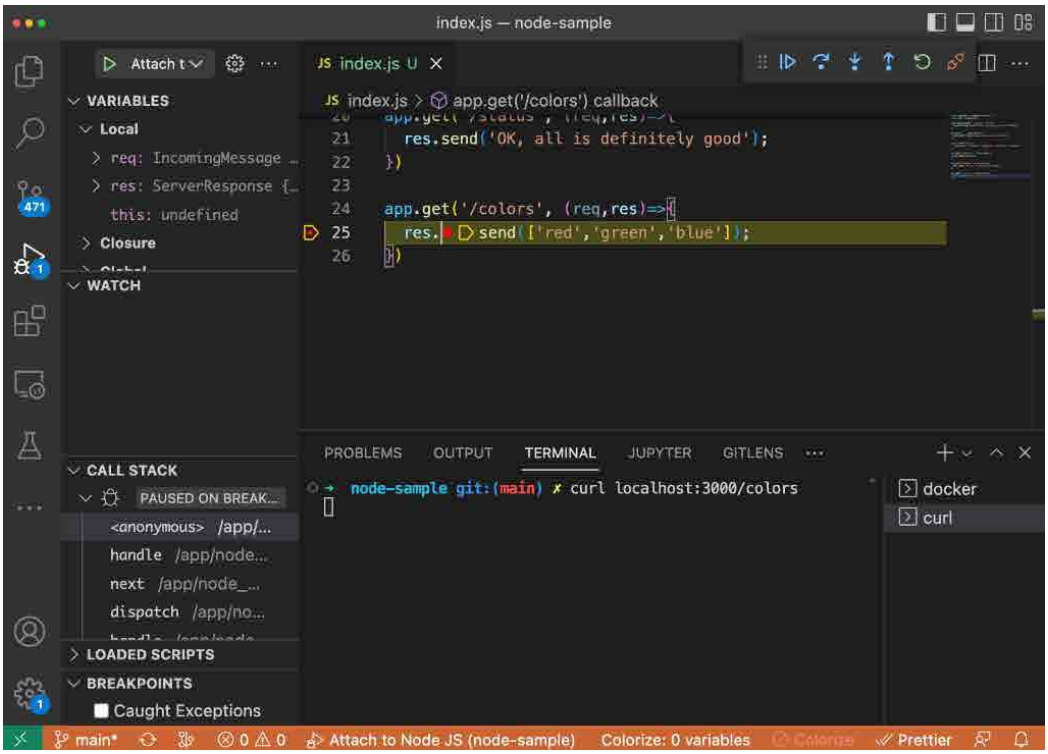


Figure 6.30 – The code's execution stops at the breakpoint

In the preceding screenshot, we can see a yellow bar, indicating that the code's execution has stopped at the breakpoint. In the top-right corner, we have a toolbar that allows us to navigate through the code step by step. On the left-hand side, we can see the **VARIABLES**, **WATCH**, and **CALL STACK** windows, which we can use to observe the details of our running application. The fact that we are debugging the code running inside the container can be verified by the fact that, in the terminal windows where we started the container, we can see that the output debugger is attached, which was generated the moment we started debugging inside VS Code.

11. To stop the container, enter the following command in the terminal window:

```
$ docker container rm --force node-sample
```

12. If we want to use `nodemon` for even more flexibility, then we have to change the `container run` command slightly:

```
$ docker container run --rm \
    --name node-sample \
    -p 3000:3000 \
    -p 9229:9229 \
```

```
    -v $(pwd):/app \
    node-sample-dev nodemon --inspect=0.0.0.0 index.js
```

Note how we use the start command, `nodemon --inspect=0.0.0.0 index.js`. This will have the benefit that, upon any code changes, the application running inside the container will restart automatically, as we learned earlier in this chapter. You should see the following:

```
➜  node-sample git:(main) ✗ docker container run --rm --name node-sample \
-p 3000:3000 -p 9229:9229 -v $(pwd):/app \
node-sample-dev nodemon --inspect=0.0.0.0 index.js
[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node --inspect=0.0.0.0 index.js`
Debugger listening on ws://0.0.0.0:9229/f2e524ea-2a27-4774-87d0-7d1993c89603
For help, see: https://nodejs.org/en/docs/inspector
Application listening at 0.0.0.0:3000
Debugger attached.
[nodemon] restarting due to changes...
[nodemon] starting `node --inspect=0.0.0.0 index.js`
Debugger listening on ws://0.0.0.0:9229/f9c3af87-c433-419b-8054-9b4918f68d6f
For help, see: https://nodejs.org/en/docs/inspector
Application listening at 0.0.0.0:3000
```

Figure 6.31 – Starting the Node.js application with nodemon and debugging turned on

13. Unfortunately, the consequence of an application restart is that the debugger loses its connection with VS Code. But don't worry – we can mitigate this by adding `"restart": true` to our launch task in the `launch.json` file. Modify the task so that it looks like this:

```
{
    "type": "node",
    "request": "attach",
    "name": "Docker: Attach to Node",
    "remoteRoot": "/app",
    "restart": true
},
```

14. After saving your changes, start the debugger in VS Code by clicking the green start button in the debug window. In the terminal, you should see that the debugger is attached, with a message as the output. In addition to that, VS Code will have an orange status bar at the bottom, indicating that the editor is in debug mode.
15. In a different terminal window, use `curl` and try to navigate to `localhost:3000/colors` to test that your line-by-line debugging still works. Make sure the code execution stops at any breakpoint you have set in the code.

16. Once you have verified that debugging still works, try to modify some code; for example, change the array of returned colors and add yet another color. Save your changes. Observe how `nodemon` restarts the application and that the debugger is automatically re-attached to the application running inside the container:

```
➜  node-sample git:(main) ✗ docker container run --rm --name node-sample \
-p 3000:3000 -p 9229:9229 -v $(pwd):/app \
node-sample-dev nodemon --inspect=0.0.0.0 index.js
[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node --inspect=0.0.0.0 index.js`
Debugger listening on ws://0.0.0.0:9229/1873b319-40ca-4860-8e57-c8c6bb85a188
For help, see: https://nodejs.org/en/docs/inspector
Application listening at 0.0.0.0:3000
Debugger attached.
[nodemon] restarting due to changes...
[nodemon] starting `node --inspect=0.0.0.0 index.js`
Debugger listening on ws://0.0.0.0:9229/90543579-93e8-472f-8d39-b605502877ce
For help, see: https://nodejs.org/en/docs/inspector
Application listening at 0.0.0.0:3000
Debugger attached.
```

Figure 6.32 – nodemon restarting the application and the debugger automatically re-attaching to the application

With that, we have everything assembled and can now work with code running inside a container as if the same code were running natively on the host. We have removed pretty much all of the friction that containers brought into the development process. We can now just enjoy the benefits of deploying our code in containers.

17. To clean up, stop the container by pressing *Ctrl + C* within the terminal window from where you started it.

Now that you've learned how to debug a Node.js application running in a container line by line, let's learn how to do the same for a .NET application.

Debugging a .NET application

In this section, we want to give you a quick run-through of how to debug a .NET application line by line. We will use the sample .NET application that we created earlier in this chapter:

1. Navigate to the project folder and open VS Code from within it:

```
$ cd ~/The-Ultimate-Docker-Container-Book/ch06/csharp-sample
```

2. Then, open VS Code with the following command:

```
$ code .
```

3. To work with the debugger, we can fully rely on the help of VS Code commands. Hit *cmd* + *Shift* + *P* (*Shift* + *Ctrl* + *P* on Windows) to open the command palette.
4. Search for `Docker: Add Docker Files to Workspace` and select it:

   I. Select **.NET: ASP.NET Core** for the application platform when prompted.

   II. Select Linux when prompted to select the operating system.

   III. When you're asked if you want to add Docker Compose files, answer with **No** at this time. Later in this book, we will discover what Docker Compose files are.

   IV. Change the port to use for the application to `5000`.

   Once you have entered all the required information, a `Dockerfile` and a `.dockerignore` file will be added to the project. Take a moment to explore both. Notice that this `Dockerfile` is defined as a multistage build.

   The previous command also added the `launch.json` and `tasks.json` files to a new `.vscode` folder in the project. These will be used by VS Code to help it define what to do when we ask it to debug our sample application.

5. Let's put a breakpoint in the first `GET` request of the `WeatherForecastController.cs` file.
6. Locate the `.vscode/launch.json` file in the project and open it.
7. Locate the Docker .NET Core Launch debug configuration and add the snippet marked with the red rectangle to it:

```
{
  "name": "Docker .NET Core Launch",
  "type": "docker",
  "request": "launch",
  "dockerServerReadyAction": {
    "uriFormat": "%s://localhost:%s/WeatherForecast"
  },
  "preLaunchTask": "docker-run: debug",
  "netCore": {
    "appProject": "${workspaceFolder}/csharp-sample.csproj"
  }
```

Figure 6.33 – Modifying the Docker Launch configuration

   The `dockerServerReadyAction` property in the `launch.json` file of a .NET project in VS Code is used to specify an action that should be taken when a Docker container is ready to accept requests.

8. Switch to the debug window of VS Code (use *Command* + *Shift* + *D* or *Ctrl* + *Shift* + *D* on Linux or Windows to open it, respectively). Make sure you have selected the correct debug launch task – its name is Docker .NET Core Launch:

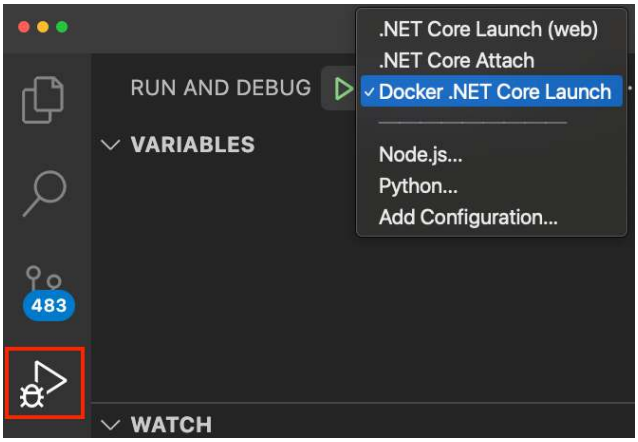


Figure 6.34 – Selecting the correct debug launch task in VS Code

9. Now, click the green start button to start the debugger. VS Code will build the Docker images, run a container of them, and configure the container for debugging. The output will be shown in the terminal window of VS Code. A browser window will open and navigate to `http://localhost:5000/wetherforecast` since this is what we defined in the launch configuration (*step 6*). At the same time, the breakpoint in the application controller is hit, as shown here:

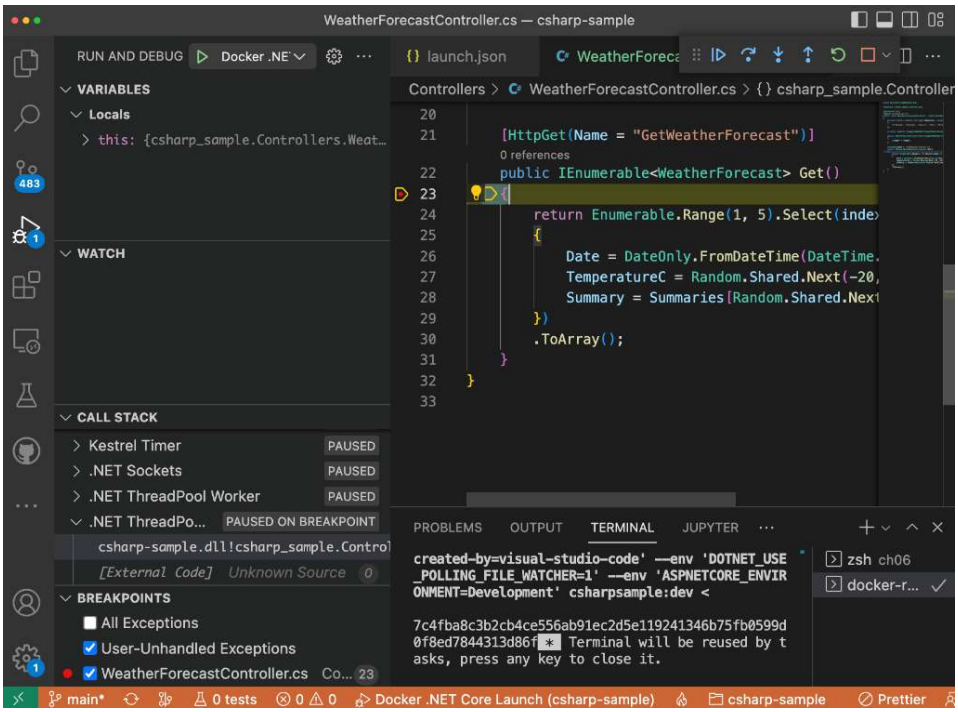


Figure 6.35 – Debugging a .NET Core application running inside a container line by line

10. We can now step through the code, define watches, or analyze the call stack of the application, similar to what we did with the sample Node.js application. Hit the **Continue** button on the debug toolbar or press *F5* to continue executing the code.
11. To stop the application, click the red stop button in the debugging toolbar, which is visible in the top-right corner of the preceding screenshot.

Now that we know how to debug code running in a container line by line, it is time to instrument our code so that it produces meaningful logging information.

Instrumenting your code to produce meaningful logging information

Once an application is running in production, it is impossible or strongly discouraged to interactively debug the application. Thus, we need to come up with other ways to find the root cause when the system is behaving unexpectedly or causing errors. The best way is to have the application generate detailed logging information that can then be used by the developers that need to track down any errors. Since logging is such a common task, all relevant programming languages or frameworks offer libraries that make the task of producing logging information inside an application straightforward.

It is common to categorize the information that's output by an application as logs into so-called severity levels. Here is a list of those severity levels with a short description of each:

Log Level	Description
TRACE	Very fine-grained information. At this level, you are looking at capturing every detail possible about your application's behavior.
DEBUG	Relatively granular and mostly diagnostic information that helps you pin down potential problems if they occur.
INFO	Normal application behavior or milestones, such as startup or shutdown information.
WARNING	The application might have encountered a problem, or you detected an unusual situation.
ERROR	The application encountered a serious issue. This most probably represents the failure of an important application task.
FATAL	The catastrophic failure of your application. The immediate shutdown of the application is advised.

Table 6.1 – A list of the severity levels used when generating logging information

Logging libraries usually allow a developer to define different log sinks – that is, destinations for the logging information. Popular sinks are file sinks or a stream to the console. When working with containerized applications, it is strongly recommended that you always direct logging output to the console or `STDOUT`. Docker will then make this information available to you via the `docker container logs` command. Other log collectors, such as Logstash, Fluentd, Loki, and others, can also be used to scrape this information.

Instrumenting a Python application

Let's try to instrument our existing Python sample application:

1. First, in your terminal, navigate to the project folder and open VS Code:

```
$ cd ~/The-Ultimate-Docker-Container-Book/ch06/python-demo
```

2. Open VS Code with the following command:

```
$ code .
```

3. Open the `main.py` file and add the following code snippet to the top of it:

```
1  import logging
2
3  logger = logging.getLogger("Python demo app")
4  logger.setLevel(logging.WARN)
5  # create a console handler
6  ch = logging.StreamHandler()
7  # create a formatter and add it to the handlers
8  formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
9  ch.setFormatter(formatter)
10 logger.addHandler(ch)
```

Figure 6.36 – Defining a logger for our Python sample application

On line 1, we import the standard logging library. We then define a logger for our sample application on line 3. On line 4, we define the filter for logging to be used. In this case, we set it to `WARN`. This means that all logging messages produced by the application with a severity equal to or higher than `WARN` will be output to the defined logging handlers or sinks, which is what we called them at the beginning of this section. In our case, only log messages with a log level of `WARN`, `ERROR`, or `FATAL` will be output.

On line 6, we create a logging sink or handler. In our case, it is `StreamHandler`, which outputs to `STDOUT`. Then, on line 8, we define how we want the logger to format the messages it outputs. Here, the format that we chose will output the time and date, the application (or logger) name, the log severity level, and finally, the actual message that we developers define in the code. On line 9, we add the formatter to the log handler, while on line 10, we add the handler to the logger.

> **Note**
> We can define more than one handler per logger.

Now, we are ready to use the logger.

4. Let's instrument the `hello` function, which is called when we navigate to the `/` endpoint:

Figure 6.37 – Instrumenting a method with logging

As shown in the preceding screenshot, we added line 3 to the preceding snippet, where we used the `logger` object to produce a logging message with the `INFO` log level. The message is `"Accessing endpoint '/'"`.

5. Let's instrument another function and output a message with the `WARN` log level:

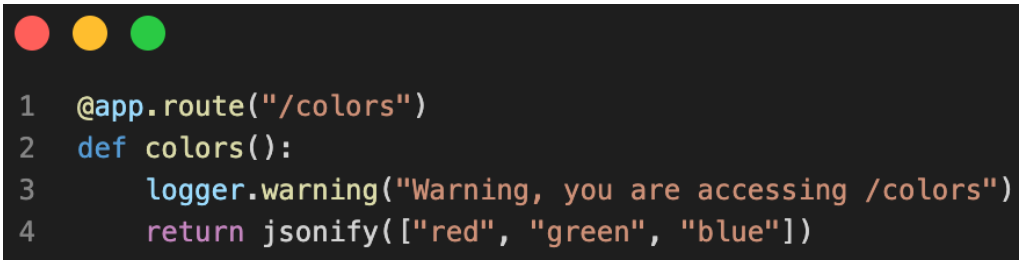


Figure 6.38 – Generating a warning

This time, we produced a message with the `WARN` log level on line 3 in the `colors` function. So far, so good – that wasn't hard!

6. Now, let's run the application and see what output we get:

```
$ python3 main.py
```

7. Then, in your browser, navigate to `localhost:5000/` first and then to `localhost:5000/colors`. You should see an output like this:

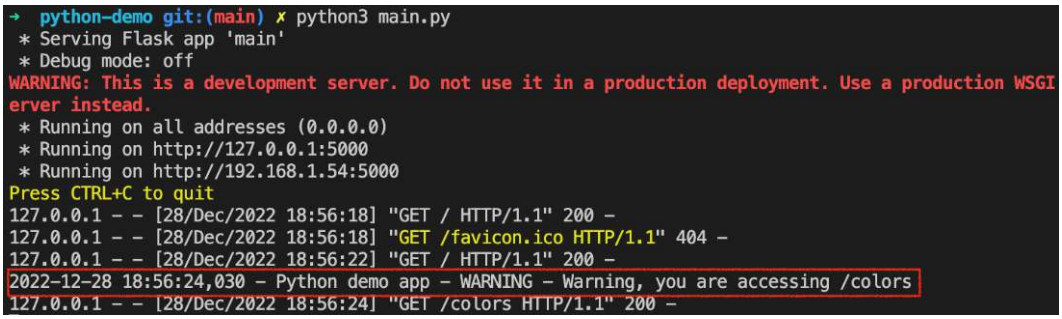


Figure 6.39 – Running the instrumented sample Python application

As you can see, only the warning is output to the console; the `INFO` message is not. This is due to the filter we set when defining the logger. Also, note how our logging message is formatted with the date and time at the beginning, then the name of the logger, the log level, and finally, the message that was defined on line 3 of the snippet shown in *Figure 6.39*.

8. When you're done, stop the application by pressing *Ctrl* + *C*.

Now that we've learned how to instrument a Python application, let's learn how to do the same for .NET.

Instrumenting a .NET C# application

Let's instrument our sample C# application:

1. First, navigate to the project folder, from where you'll open VS Code:

```
$ cd ~/The-Ultimate.Docker-Container-Book/ch06/csharp-sample
```

2. Open VS Code with the following command:

```
$ code .
```

3. Next, we need to add a NuGet package containing the logging library to the project:

```
$ dotnet add package Microsoft.Extensions.Logging
```

This should add the following line to your `dotnet.csproj` project file:

```
<PackageReference Include="Microsoft.Extensions.Logging"
Version="7.0.0" />
```

4. Open the `Program.cs` class and notice that we have the following statement on line 1:

```
var builder = WebApplication.CreateBuilder(args);
```

This method call, by default, adds a few logging providers to the application, among which is the console logging provider. This comes in very handy and frees us from having to do any complicated configuration first. You can, of course, override the default setting at any time with your own settings.

5. Next, open the `WeatherForecastController.cs` file in the `Controllers` folder and add the following:

   I. Add an instance variable, `logger`, of the `ILogger` type.

   II. Add a constructor that has a parameter of the `ILogger< WeatherForecastController >` type. Assign this parameter to the `logger` instance variable:

```
1  [ApiController]
2  [Route("[controller]")]
3  public class WeatherForecastController : ControllerBase
4  {
5      private ILogger logger;
6      public WeatherForecastController(ILogger<WeatherForecastController> logger)
7      {
8          this.logger = logger;
9      }
```

Figure 6.40 – Defining a logger for the Web API controller

6. Now, we're ready to use the logger in the controller methods. Let's instrument the `Get` method with an *info* message (line 4 in the following code):

```
1      [HttpGet(Name = "GetWeatherForecast")]
2      public IEnumerable<WeatherForecast> Get()
3      {
4          logger.LogInformation("Accessing the /weatherforecast endpoint.");
5          return Enumerable.Range(1, 10).Select(index => new WeatherForecast
6          {
7
```

Figure 6.41 – Logging an INFO message from the API controller

7. Now, let's add a method that implements a `/warning` endpoint right after the `Get` method and instrument it (line 4 here):

```
1  [HttpGet("/warning")]
2  public string ShowWarning()
3  {
4      logger.LogWarning("This endpoint shows a warning!");
5      return "Just a warning";
6  }
```

Figure 6.42 – Logging messages with the WARN log level

8. Let's run the application by using the following command:

```
$ dotnet run
```

9. We should see the following output when in a new browser tab. To do so, we must navigate to `localhost:3000/weatherforecast` and then `localhost:3000/warning`:

```
→  csharp-sample git:(main) ✗ dotnet run
Building...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://0.0.0.0:5080
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: /Users/gabriel/The-Ultimate-Docker-Container-Book/sample-solutions/ch06/csharp-sample
warn: Microsoft.AspNetCore.HttpsPolicy.HttpsRedirectionMiddleware[3]
      Failed to determine the https port for redirect.
info: csharp_sample.Controllers.WeatherForecastController[0]
      Accessing the /weatherforecast endpoint.
warn: csharp_sample.Controllers.WeatherForecastController[0]
      This endpoint shows a warning!
```

Figure 6.43 – The log output of our sample .NET application

We can see the output of our log message, which is of the `info` and `warn` types, marked by red arrows. All the other log items have been produced by the ASP.NET library. You can see that there is a lot of helpful information available if you need to debug the application.

10. When you're done, end the application with *Ctrl* + *C*.

Now that we have learned how to instrument code to simplify how we can find the root cause of an issue when running in production, next, we will look at how we can instrument a distributed application using the Open Tracing standard for distributed tracing and then use Jaeger as a tool.

Using Jaeger to monitor and troubleshoot

When we want to monitor and troubleshoot transactions in a complex distributed system, we need something a bit more powerful than what we have just learned. Of course, we can and should continue to instrument our code with meaningful logging messages, yet we need something more on top of

that. This *more* is the capability to trace a single request or transaction end to end, as it flows through a system consisting of many application services. Ideally, we also want to capture other interesting metrics, such as the time spent on each component versus the total time that the request took.

Luckily, we do not have to reinvent the wheel. There is battle-tested open source software out there that helps us achieve the aforementioned goals. One example of such an infrastructure component or software is Jaeger (`https:/ www.jaegertracing.io/`). When using Jaeger, you run a central Jaeger server component and each application component uses a Jaeger client that will forward debug and tracing information transparently to the Jaeger server component. There are Jaeger clients for all major programming languages and frameworks, such as Node.js, Python, Java, and .NET.

We will not go into all the intimate details of how to use Jaeger in this book, but we will provide a high-level overview of how it works conceptually:

1. First, we must define a Jaeger tracer object. This object coordinates the whole process of tracing a request through our distributed application. We can use this tracer object and also create a logger object from it, which our application code can use to generate log items, similar to what we did in the previous Python and .NET examples.
2. Next, we must wrap each method in the code that we want to trace with what Jaeger calls a span. This span has a name and provides us with a scope object.
3. Let's look at some C# pseudocode that illustrates this:

```
1  public void SayHello(string helloTo) {
2      using(var scope = _tracer.BuildSpan("sayhello").
3          StartActive(true)) {
4          // here is the actual logic of the method
5          …
6          var helloString = FormatString(helloTo);
7          …
8      }
9  }
```

Figure 6.44 – Defining a span in Jaeger – pseudocode

As you can see, we're instrumenting the `SayHello` method. With a `using` statement creating a span, we're wrapping the whole application code of this method. We have called the span `sayhello`; this will be the ID with which we can identify the method in the trace log produced by Jaeger.

Note that the method calls another nested method, `FormatString`. This method will look quite similar to the code needed to instrument it.

The span that our tracer object builds in this method will be a child span of the calling method. This child span is called `format-string`. Also, note that we are using the logger object in the preceding method to explicitly generate a log item of the `INFO` log level:

```
public void string Format(string helloTo) {
    using(var scope = _tracer.BuildSpan("format-string").StartActive(true))
    {
        // here is the actual logic of the method
        …
        _logger.LogInformation(helloTo);
        return
    …
    }
}
```

Figure 6.45 – Creating a child span in Jaeger – pseudocode

In the code included with this chapter, you can find a complete sample application written in Java and Spring Boot consisting of a Jaeger server container and two application containers called `api` and `inventory` that use the Jaeger client library to instrument the code. Follow these steps to rebuild this solution:

1. Navigate to the **Spring Initializr** page at `https://start.spring.io` and create bootstrap code for a project called `api`, as follows:

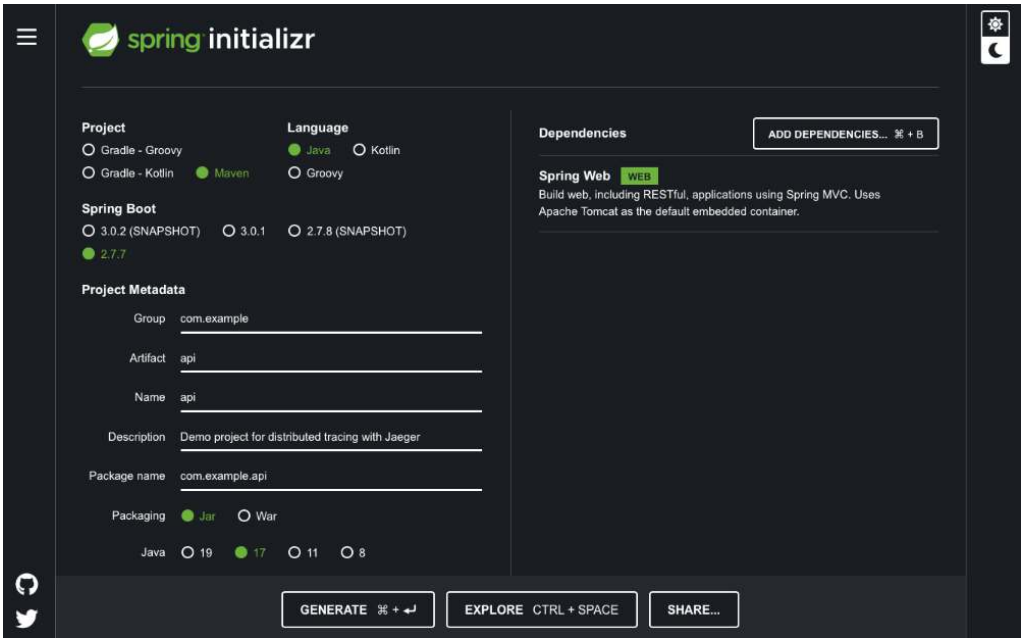


Figure 6.46 – Bootstrapping the API component of the Jaeger sample

Note how we are using Spring Boot 2.7.7 for this example since, at the time of writing, the Jaeger and Open Tracing integration does not yet work with Spring Boot 3. Also, note how we have added the Spring Web reference to the project.

2. Click **GENERATE**. A ZIP file called `api.zip` will be downloaded to your computer.
3. Repeat the same steps but this time change the **Artifact** and **Name** entries to `inventory`. Then, click **GENERATE** again; a file called `inventory.zip` containing the bootstrap code will be downloaded to your computer.
4. Navigate to the source folder for this chapter:

```
$ cd ~/The-Ultimate-Docker-Container-Book/ch06
```

5. Then, create a subfolder called `jaeger-demo` in it:

```
$ mkdir jaeger-demo
```

6. Extract the two ZIP files into the `jaeger-demo` folder. Make sure the subfolders are called `api` and `inventory`, respectively.
7. Open VS Code from within this folder:

```
$ code .
```

8. Next, create a `docker-compose.yml` file in the root with this content:

```
version: "3"
services:
  jaeger:
    image: jaegertracing/all-in-one:latest
    container_name: jaeger
    environment:
      COLLECTOR_ZIPKIN_HTTP_PORT: 9411
    ports:
      - 5775:5775/udp
      - 6831:6831/udp
      - 6832:6832/udp
      - 5778:5778
      - 16686:16686
      - 14268:14268
      - 9411:9411
```

Figure 6.47 – The Docker Compose file for the Jaeger demo

We will explain what a `docker-compose` file is in detail in *Chapter 11*, *Managing Containers with Docker Compose*.

9. Run Jaeger with this command:

```
$ docker compose up -d
```

10. In a new browser tab, navigate to the Jaeger UI at `http://localhost:16686`.
11. Locate the two `pom.xml` files for the `api` and `inventory` projects in your VS Code. Add the Jaeger integration component to each file by adding this snippet to their `dependencies` sections:

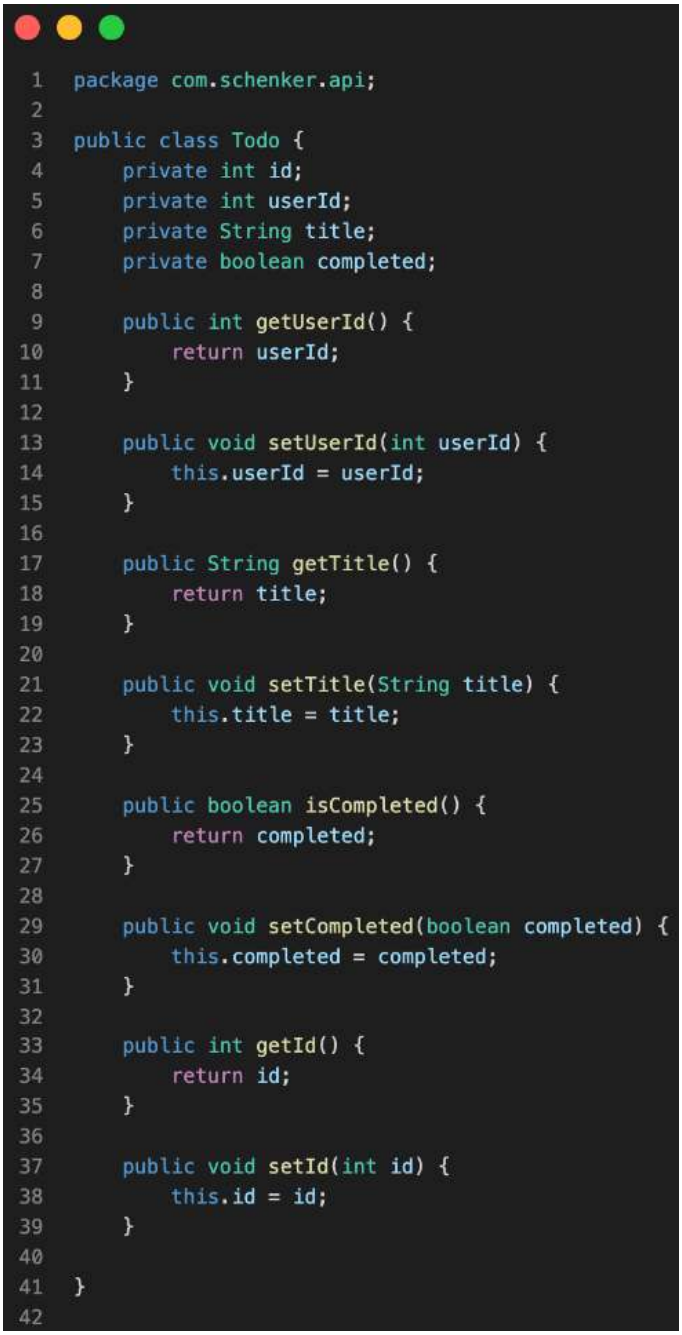
```
18    <dependencies>
19        <dependency>
20            <groupId>org.springframework.boot</groupId>
21            <artifactId>spring-boot-starter-web</artifactId>
22            <version>2.7.5</version>
23        </dependency>
24
25        <dependency>
26            <groupId>io.opentracing.contrib</groupId>
27            <artifactId>opentracing-spring-jaeger-cloud-starter</artifactId>
28            <version>3.3.1</version>
29        </dependency>
30
31        <dependency>
32            <groupId>org.springframework.boot</groupId>
33            <artifactId>spring-boot-starter-test</artifactId>
```

Figure 6.48 – Adding integration with Jaeger to the Java project(s)

12. In the `inventory` project, locate the start class, `InventoryApplication`, and add a bean to it that generates an instance of `RestTemplate`. We will use this to access an external API to download some data. The code snippet should look like this:

```
@Bean
RestTemplate restTemplate() {
    return new RestTemplate();
}
```

13. Do the same for the start class of the `api` project, called `ApiApplication`.
14. Now, let's go back to the `inventory` project. Add a new file called `Todo.java` as a sibling next to the start class. The file will have the following content:



```
package com.schenker.api;

public class Todo {
    private int id;
    private int userId;
    private String title;
    private boolean completed;

    public int getUserId() {
        return userId;
    }

    public void setUserId(int userId) {
        this.userId = userId;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public boolean isCompleted() {
        return completed;
    }

    public void setCompleted(boolean completed) {
        this.completed = completed;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

}

```

Figure 6.49 – The Todo class in the api project for the Jaeger demo

This is a really simple POJO class that we are using as a data container.

15. Do the same in the `api` project.
16. Go to the `inventory` project and add a new file called `TodosController.java` with the following content:

```
1  package com.schenker.inventory;
2
3  import org.springframework.web.bind.annotation.GetMapping;
4  import org.springframework.web.bind.annotation.RequestMapping;
5  import org.springframework.web.bind.annotation.RestController;
6  import org.springframework.web.client.RestTemplate;
7
8  @RestController
9  @RequestMapping("/api/todos")
10 public class TodosController {
11     private final RestTemplate restTemplate;
12
13     public TodosController(RestTemplate restTemplate) {
14         this.restTemplate = restTemplate;
15     }
16
17     @GetMapping
18     public Todo[] getTodos() {
19         var response = restTemplate.getForEntity("https://jsonplaceholder.typicode.com/todos", Todo[].class);
20         return response.getBody();
21     }
22 }
23
```

Figure 6.50 – The TodosController class for the Jaeger demo

Notice how, on line 19, we reach out to the public **JSONPlaceholder API** to download a list of todo items and return those items to the caller on line 20. There's nothing fancy here.

17. For the `api` project, add a new file called `HelloController.java` with the following content:

```java
package com.schenker.api;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@RestController
public class HelloController {
    private static final Logger logger = LoggerFactory.getLogger(HelloController.class);
    private RestTemplate restTemplate;

    public HelloController(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    @GetMapping("/hello")
    public String hello() {
        logger.info("Requesting hello world!");
        return "Hello World!";
    }


    @GetMapping("/todos")
    public Todo[] getTodos() {
        logger.info("Requesting list of todos!");
        var todos = restTemplate.getForEntity("http://localhost:8090/api/todos", Todo[].class);
        return todos.getBody();
    }
}

```

Figure 6.51 – The HelloController class for the Jaeger demo

Notice how the first method, which is listening on the `/hello` endpoint, just returns a string. However, the second endpoint, which is listening on the `/todos` endpoint, reaches out to the `api` service and its endpoint, `/api/todos`. The `api` service will send back the list of todos that it downloaded from the JSON Placeholder API. This way, we have a real distributed application ready to demonstrate the power of Jaeger and Open Tracing.

18. We are not quite done yet. We need to configure both projects via their respective `applications.properties` files:
19. Locate the `application.properties` file in the `api` project and add the following line to it:

```
spring.application.name=jaeger-demo:api
```

The preceding code defines the name of the service and how Jaeger will report it.

20. Locate the same file in the `inventory` project and add the following two lines to it:

```
server.port=8090
spring.application.name=jaeger-demo:inventory
```

21. The first line makes sure the inventory service is listening at port `8090` and not at the default port of `8080` to avoid any conflict with the `api` service, which will run on the default port.

    The second line defines the name of the service and how Jaeger will report it.

22. Now, start the `inventory` and `api` projects from within VS Code by clicking the **Run** hyperlink and decorating the `main` methods of their respective start classes.
23. Use `curl` or Thunder Client to access the exposed endpoint of the inventory service at `http://localhost:8090/api/todos`. You can also do the same in a new browser tab. You should receive a list of 100 random todo items.
24. Next, try to access the `api` service at the `http://localhost:8080/todos` endpoint. The same list of todos should be returned, but this time, they should originate from the `api` service and not directly from the JSON Placeholder API.
25. Now, go back to the browser tab where you opened the Jaeger UI.
26. Make sure you are on the **Search** tab.
27. From the **Services** drop-down list, select **jaeger-demo:api**.
28. Click **Find Traces**. You should see something like this:

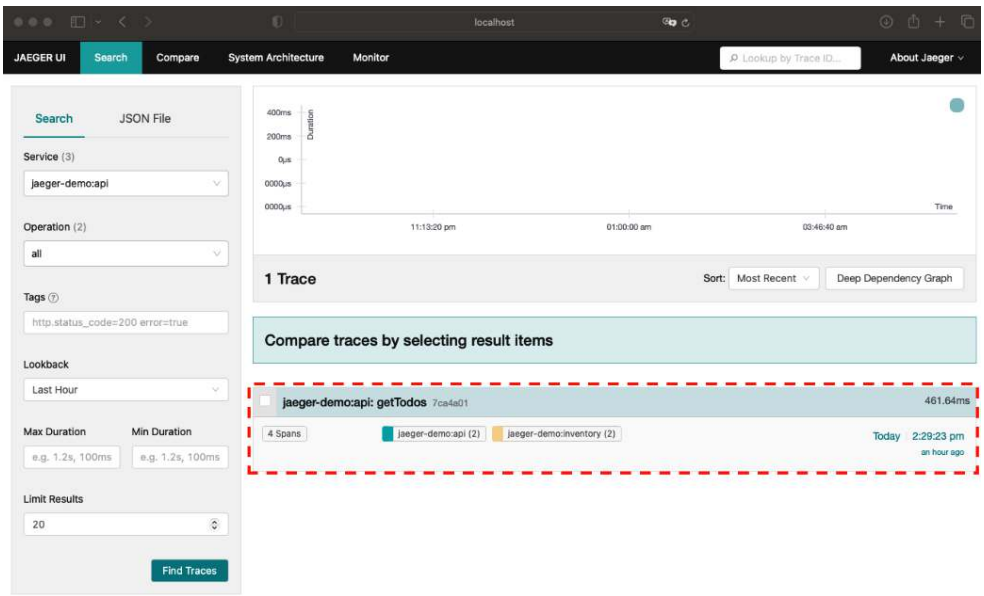


Figure 6.52 – Jaeger trace for the api service

29. Click on the trace to expand it. You should see this:

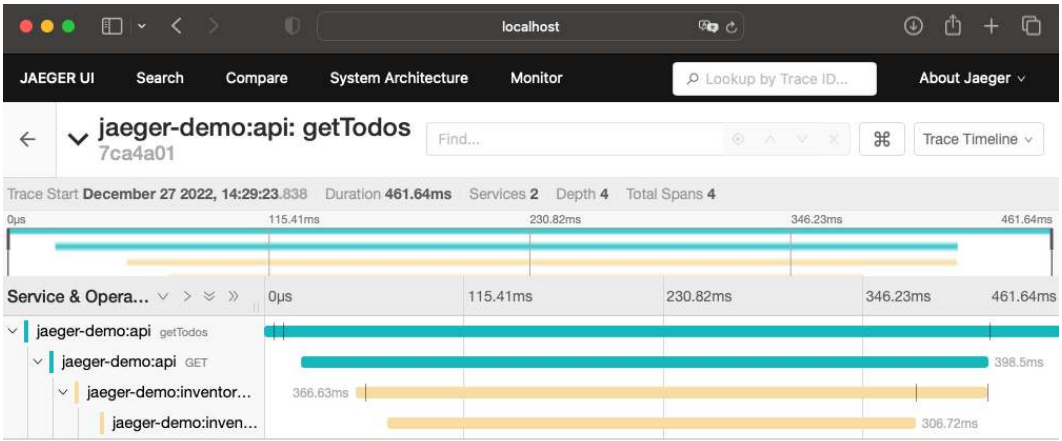


Figure 6.53 – Details of the Jaeger trace for the api service

Here, we can see how the call is reaching out from the `api` service to the `inventory` service. We can also see the time that's been spent on each component.

30. To clean up, stop the Jaeger server container:

```
$ docker compose down
```

31. Also, stop the API with *Ctrl* + *C*.

In this demo, we saw that without any special code, by just adding a component that integrates our Spring Boot applications with Jaeger and Open Tracing, we have gained a lot of insight. However, we're only just scratching the surface of what is possible.

Challenge: Try to containerize the `api` and `inventory` services using a similar `Dockerfile` for each, as we did in the Java demo application earlier in this chapter. The respective `Dockerfile` should be in the root of the `api` and `inventory` projects.

Then, amend the `docker-compose.yml` file. When you're done, run the whole application with this command:

```
$ docker compose up -d
```

Don't worry if you're not familiar with Docker Compose yet. We will discuss this very useful tool in *Chapter 11*, *Managing Containers with Docker Compose*.

Summary

In this chapter, we learned how to run and debug Node.js, Python, Java, and .NET code running inside a container. We started by mounting the source code from the host into the container to avoid the container image being rebuilt each time the code changes. Then, we smoothed out the development process further by enabling automatic application restarts inside the container upon code changes. Next, we learned how to configure VS Code to enable full interactive code debugging when code is running inside a container.

Finally, we learned how we can instrument our applications so that they generate logging information that can help us do root cause analysis on failures of misbehaving applications or application services running in production. We started by instrumenting our code using a logging library. Then, we used the Open Tracing standard for distributed tracing and the Jaeger tool to instrument a Java and Spring Boot application and gain valuable insight into the application's inner workings.

In the next chapter, we are going to show you how using Docker containers can supercharge your automation, from running a simple automation task in a container to using containers to build a CI/CD pipeline.

Questions

Try to answer the following questions to assess your learning progress:

1. Name two methods that help reduce the friction in the development process that's introduced by using containers.
2. How can you achieve live code inside a container?
3. When and why would you debug code line by line when running inside a container?
4. Why is instrumenting code with good debugging information paramount?

Answers

Here are the answers to this chapter's questions:

1. Possible answers:
    - Volume-mount your source code in the container
    - Use a tool that automatically restarts the app running inside the container when code changes are detected
    - Configure your container for remote debugging
2. You can mount the folder containing the source code on your host in the container.

3. If you cannot cover certain scenarios easily with unit or integration tests and if the observed behavior of the application cannot be reproduced when the application runs on the host. Another scenario is a situation where you cannot run the application on the host directly due to a lack of the necessary language or framework.
4. Once an application is running in production, we cannot easily gain access to it as developers. If the application shows unexpected behavior or even crashes, logs are often the only source of information we have to help us reproduce the situation and pinpoint the root cause of the bug.

7

Testing Applications Running in Containers

In the previous chapters, we have learned how we can containerize our applications written in any language, such as Node.js, Python, Java, C#, and .NET. We all know that just writing code and then shipping it to production is not enough. We also need to guarantee that the code is error-free and that it does what it is supposed to do. This is commonly subsumed under the term **quality assurance**, or **QA** for short.

It has been proven in practice over and over again that fixing a bug in an application that has been discovered in production as opposed to during development is very costly. We want to avoid this. The most cost-effective way to do so is to have the developer who writes the code also write automated tests that make sure the new or changed code is of high quality and performs exactly as specified in the acceptance criteria of the business requirement or feature specification.

Here is a list of the topics we are going to discuss in this chapter:

- The benefits of testing applications running in containers
- Different types of testing
- Commonly used tools and technologies
- Best practices for setting up a testing environment
- Tips for debugging and troubleshooting issues
- Challenges and considerations when testing applications running in containers
- Case studies

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

- Explain the benefits of testing applications running in containers to an interested layperson
- Set up a productive environment that allows you to write and execute tests for applications or services running in containers
- Develop unit and integration tests for code running in a container
- Run your unit and integration tests in a container with the application code under test
- Run a dedicated container with functional tests that act on your application as a black box
- Manage application dependencies and create test data

Technical requirements

In this chapter, you need Docker Desktop, a terminal, and VS Code installed on your Mac, Windows, or Linux machine. As we will work with code, you should prepare a chapter folder in the code repository you cloned from GitHub:

1. Navigate to the folder to which you cloned the GitHub repository accompanying this book. Normally, you do this as follows:

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

2. Create a chapter folder in this directory and navigate to it:

```
$ mkdir ch07 && cd ch07
```

As always, you can find complete sample solutions for all the exercises we will do in this chapter in the `sample-solutions/ch07` subfolder.

Benefits of testing applications in containers

In this section, we are discussing the benefits of testing applications in containers, including the ability to replicate production environments, ease of configuration and setup, and faster test execution.

But before we start, let's pause for a second and ask ourselves, why do we care to test at all?

Why do we test?

Every person working in any role in software development is aware that one needs to implement and ship new or changed application features at a fast cadence. There is constant pressure to implement new code and ship it as quickly as possible to production. But business analysts that write the feature specifications and software engineers that write the actual code implementing the specifications are just human beings. Human beings working under a lot of pressure tend to make mistakes. These

mistakes can be subtle, or they can be quite substantial. Those mistakes will manifest themselves in the application running in production. Our customers will discover them, and this will have consequences.

Manual versus automated testing

Most companies that write commercial applications will have a team of manual software testers. These people will take the newest version of the application that product engineering has prepared for them and execute a suite of manual regression tests against this application. If a manual tester discovers a bug, they will report it in a tool such as Jira as a bug ticket, where they will ideally write down all the necessary details that matter for the developer who will have to fix the bug. This includes the exact version of the application tested, the steps that the tester took before the bug was detected, and some evidence of the bug, such as screenshots, error messages, stack traces, and log entries. These tickets written by manual testers will become part of the backlog of product engineering.

Product engineering will then, together with the testers, triage all the new bug tickets on a regular basis, say daily, and decide how quickly a particular bug needs to be addressed. Usually, the classification of P1, P2, P3, and P4 is used, where P1 is a defect of the highest severity that needs to be fixed immediately, and P4 is a bug that is of low priority and can be dealt with whenever the team has time.

If the application is a typical enterprise application consisting of many services all running in the cloud, then the testers need a special environment where they can perform their regression testing. This environment is often called **user acceptance testing**, or **UAT** for short. A full test suite for such an enterprise application usually consists of several hundred test cases. To perform a single test case takes a manual tester a considerable amount of time. It is not unheard of that a team of dedicated manual testers needs a couple of weeks to perform a full test run. During this time, the UAT environment is blocked. No new version can be deployed to this environment, because otherwise the testers would have to restart their regression testing. Each change in the application can introduce new bugs, and we can only be certain to catch them all if we execute the whole suite of regression tests on each new version.

Only after the manual testers have run through all regression tests, and only if no more severe bugs have been discovered, can the current version of the application be shipped to production.

I bet you can imagine that having UAT blocked for several weeks at a time can introduce some significant problems in the software development process. Your many product engineering teams will have accumulated a lot of new code in the form of new features and bug fixes that are blocked from being shipped to production since the manual testers are still testing the previous version. But accumulating a lot of code changes does, at the same time, increase risk. To ship a piece of software that has undergone many changes is riskier than if we continuously ship new versions with minimal changes to production.

The only real solution to this problem is to shorten the regression test cycle. We need to shorten it from weeks to minutes or a small number of hours. This way, we can test and ship small batches of changes in a continuous fashion. But no human being is able to test so fast. The solution is to exclusively

use automated testing. And yes, I mean it: we should rely exclusively on automated regression and acceptance testing.

What did we learn? Manual testing is not scalable, it is super boring, since the testers have to repeat the same tests over and over again, and it is error prone, since everything humans do is not automated and thus not exactly repeatable every time.

Does this mean we have to fire all manual testers? Not necessarily. Manual testers should not perform acceptance and regression tests but rather exploratory tests. Manual testers are human beings, and they should leverage that fact and their creativity to discover yet undiscovered potential defects in the application. As the term *exploratory testing* implies, these tests are not following a particular script, but are rather random and only guided by the professional experience of the tester and their understanding of the business domain for which the application has been written. If the tester discovers a bug, they write a ticket for it, which then will be triaged and flown into the backlog of the development teams.

Why do we test in containers?

There are several reasons why it is often useful to run tests in containers:

- **Isolation**: Running tests in containers can provide a level of isolation between the test environment and the host system, which can be useful for ensuring that the test results are consistent and repeatable.
- **Environment consistency**: Containers allow you to package the entire test environment (including dependencies, libraries, and configuration) in a self-contained unit, which can help to ensure that the test environment is consistent across different development environments.
- **Ease of use**: Containers can make it easier to set up and run tests, as you don't have to manually install and configure all of the required dependencies and libraries on the host system.
- **Portability**: Containers can be easily moved between different environments, which can be useful for running tests in different environments or on different platforms.
- **Scalability**: Containers can make it easier to scale up your test infrastructure by allowing you to run tests in parallel or on multiple machines.

Overall, running tests in containers can help to improve the reliability, consistency, and scalability of the testing process and can make it easier to set up and maintain a testing environment that is isolated from the host system.

Different types of testing

This section gives an overview of different types of testing that can be performed on applications running in containers, including unit tests, integration tests, and acceptance tests.

Unit tests

A unit test's primary objective is to validate the functionality of a *unit*, or tiny, isolated portion of code. In order to check that the code is accurate and operates as expected, developers frequently build unit tests as they create or modify the code. These tests are then routinely executed as part of the development process.

With no reliance on other resources or components, unit tests are made to test distinct pieces of code in isolation. This enables developers to find and quickly solve bugs in their code and makes them quick and simple to run.

Typically, tools and testing frameworks that facilitate the creation, running, and reporting of unit tests are used to generate unit tests. These tools frequently offer capabilities such as automatic test discovery, test execution, and test results reporting, and they enable developers to create unit tests using a particular syntax or structure.

A thorough testing approach should include unit tests, since they enable developers to verify that their code is valid and works as intended at the most granular level. Normally, they are executed as a part of a **continuous integration** (**CI**) process, which is a workflow in which code changes are automatically executed each time they are committed to a version control system.

Integration tests

Software testing called *integration testing* examines how well various systems or components function together as a whole. It usually follows unit testing and entails examining how various parts of an application or system interact with one another.

Integration tests are created to examine how well various units or components interact together. They are frequently used to confirm that an application's or system's various components can function as intended. Testing the integration of several software components or the integration of a software program with external resources such as databases or APIs are examples of this.

As many components or systems need to be set up and configured in order to execute the tests, integration tests are typically more complicated and time-consuming than unit tests. In order to enable the execution and reporting of the tests, they could also call for the employment of specialist testing tools and frameworks.

Integration tests, like unit tests, are a crucial component of a thorough testing approach, because they enable developers to confirm that several systems or components can function together as intended.

Acceptance tests

Software testing of this kind, known as *acceptance testing*, ensures that a system or application is suitable for its intended use and that it satisfies all of the requirements. It usually comes after all other types of testing (such as unit testing and integration testing) and is the last step in the testing procedure.

Acceptance tests are typically developed and carried out by a different team or group of testers who are tasked with assessing the system or application from the viewpoint of the end user. These tests are intended to make sure that the system or program is simple to use, fits the demands of the intended users, and is user-friendly.

Functional testing (to ensure that the application or system performs the required functions correctly), usability testing (to make sure that the application or system is easy to use), and performance testing are just a few examples of the different types of testing that may be included in acceptance tests (to verify that the application or system performs well under different load conditions).

Acceptance testing is a crucial step in the software development process, since it enables developers to confirm that the system or application is ready for deployment and satisfies the needs of the intended customers. Although it is highly recommended to employ automated acceptance testing technologies to assist the testing process, it is often carried out manually by testers.

In this chapter, we will look at a special type of acceptance test called *black box tests*. The main differentiator compared to unit and integration tests is that these black box tests look at the system under tests from a decidedly business-oriented perspective. Ideally, acceptance tests, and with it black box tests, reflect the acceptance criteria to be found in the feature specifications written by business analysts or product owners. Most often, acceptance tests are written in a way that they look at the component to be tested as a black box. The internals of this component do not and should not matter. The test code only ever accesses the component or system under test via its public interfaces. Typically, public interfaces are APIs or messages that the component consumes or produces.

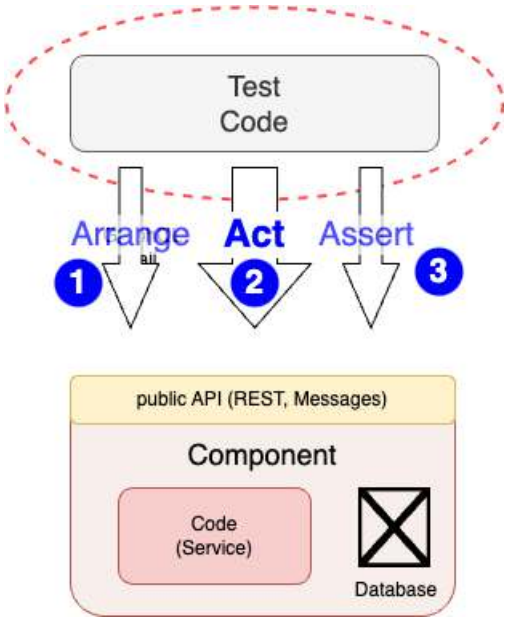


Figure 7.1 – Acceptance test interacting with the system under test

In the preceding figure, we can see how the test code is structured in the popular format of **Arrange-Act-Assert**, or **AAA**. First, we set up the boundary conditions (arrange). Next, we specify the action to exercise on the system under test (act). Finally, we verify that the outcome of the action is as expected (assert). The **system under test** (**SUT**) is the component that has a public interface in the form of either a REST API and/or messages that it consumes from a message bus. The SUT, in most cases, also has a database where it stores it state.

In the next section, we will present tools and technologies used for testing.

Commonly used tools and technologies

Let's now discuss the tools and technologies that are commonly used for testing applications running in containers, such as Docker, Kubernetes, and **continuous integration and delivery** (**CI/CD**) platforms.

Implementing a sample component

In this section, we want to implement a sample component that we are later going to use to demonstrate how we can write and execute tests for, and, specifically, how we can combine the advantage of automated tests and the use of Docker containers. We will implement the sample component using recent versions of Java and Spring Boot.

This sample component represents a simple REST API with some CRUD logic behind it. The tasks of creating and managing lists of animal species and associated races are simple enough to not warrant more complicated modeling. For simplicity, we are working with the in-memory database H2. This means that upon each restart of the component, the previous data is wiped out. If you want to change this, you can configure H2 to use a backing file for persistence instead:

1. Use the **spring initializr** page at `https://start.spring.io` to bootstrap the Java project. After configuring everything, the page should look like this:

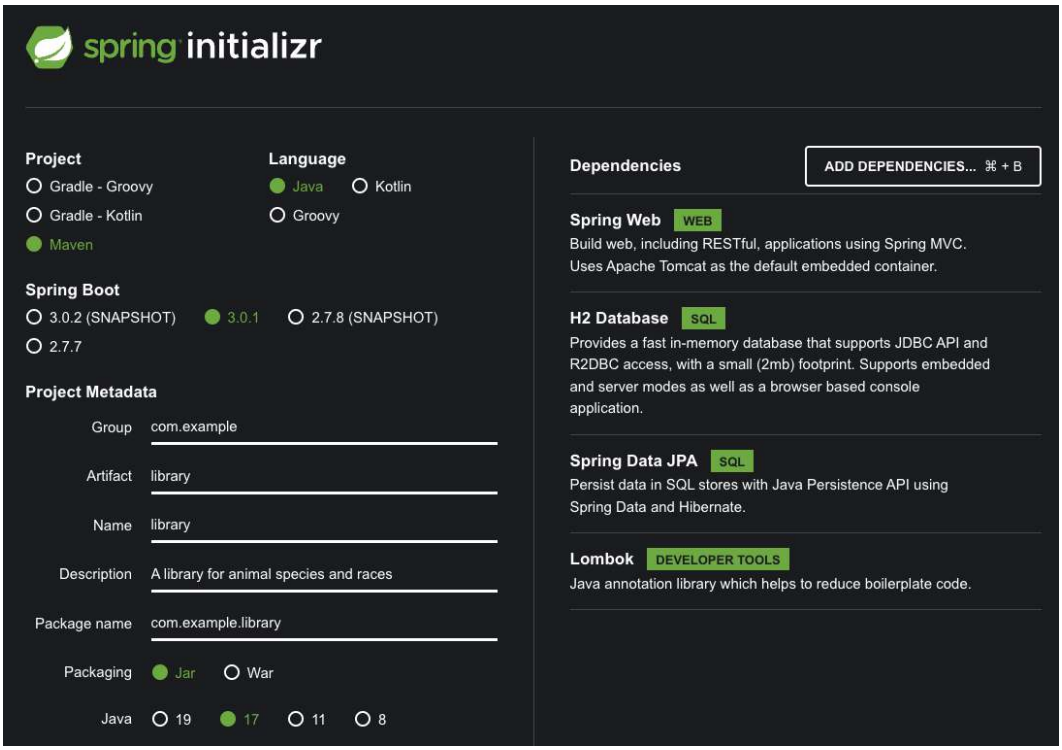


Figure 7.2 – Bootstrapping the library project

   Note how we have added the four dependencies listed on the right-hand side of the preceding figure.

2. Download the bootstrap code and unzip the file into the chapter folder, `.../ch07`. You should now have a subfolder called `library` containing the code we can use as a starting point to implement our API.
3. Open the project in VS Code.

4. Locate the `LibraryApplication.java` file in the `src/main/java/com/example/library` folder. It's the typical start class containing the `main` function for a Spring Boot-based Java application.
5. Inside this folder, create three subfolders called `controllers`, `models`, and `repositories`, respectively. They will contain the logic for our library.

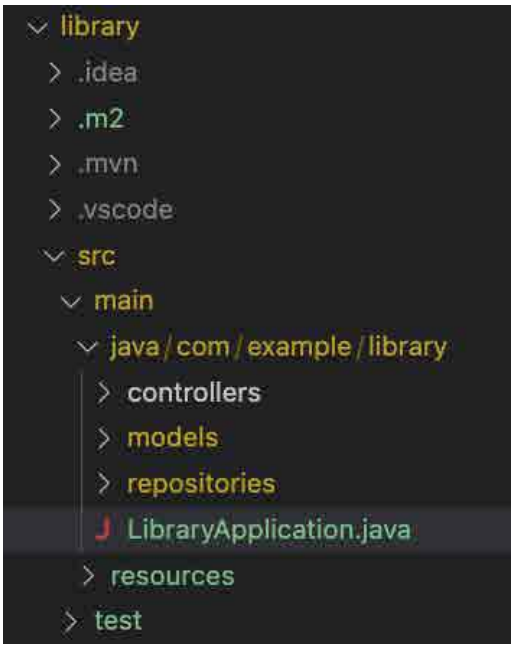


Figure 7.3 – Project structure of the library API

6. Let's first define the models we're using in our application. To the `models` folder, add the following simple data classes:

    I. To a file called `Race.java`, add the following content:

```java
package com.example.library.models;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
public class Race {
  @Id
  private Integer id;
  private Integer speciesId;
  private String name;
  private String description;
}
```

Figure 7.4 – The Race data class

II. To a file called `Species.java`, add this content:

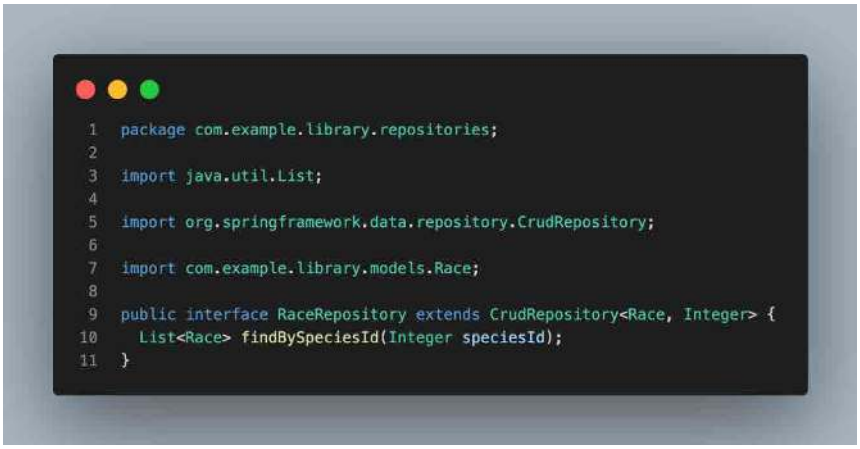
```java
package com.example.library.models;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
public class Species {
  @Id
  private int id;
  private String name;
  private String description;
}
```

Figure 7.5 – The Species data class

Note how we use the `@Entity` annotation to mark these classes as (database) entities, and we decorate their respective `id` properties with the `@Id` annotation to tell Spring Boot that this property represents the unique ID of each entity.

7. Next, we are going to implement the repositories we're going to use to persist data to and retrieve data from our database. To the `repositories` folder, add the following:

   A. A file called `RaceRepository.java` with this content:



```
1  package com.example.library.repositories;
2
3  import java.util.List;
4
5  import org.springframework.data.repository.CrudRepository;
6
7  import com.example.library.models.Race;
8
9  public interface RaceRepository extends CrudRepository<Race, Integer> {
10   List<Race> findBySpeciesId(Integer speciesId);
11 }
```

Figure 7.6 – Code for the race repository

Note how on line 10, we add a custom `findBySpeciesId` method, which will allow us to retrieve all races assigned to a given `speciesId`.

   B. A file called `SpeciesRepository.java` with the following content:

```
1  package com.example.library.repositories;
2
3  import org.springframework.data.repository.CrudRepository;
4
5  import com.example.library.models.Species;
6
7  public interface SpeciesRepository extends CrudRepository<Species, Integer> {
8
9  }
```

Figure 7.7 – Code for the species repository

8. Then, we define the two REST controllers through which we can interact with the application. To the `controllers` folder, add the following:

   A. A file called `RacesController.java` with this content:

```java
package com.example.library.controllers;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import com.example.library.models.Race;
import com.example.library.repositories.RaceRepository;

@RestController
public class RacesController {
  private final RaceRepository repository;

  public RacesController(RaceRepository repository) {
    this.repository = repository;
  }

  @GetMapping("/races")
  public Iterable<Race> index() {
    return repository.findAll();
  }

  @GetMapping("/races/{id}")
  public ResponseEntity<Race> getById(@PathVariable("id") Integer id) {
    var race = repository.findById(id);
    return race
      .map(r -> ResponseEntity.ok().body(r))
      .orElseGet(() -> ResponseEntity.notFound().build());
  }

  @GetMapping("/species/{speciesId}/races")
  public Iterable<Race> getBySpecies(@PathVariable("speciesId") Integer speciesId) {
    return repository.findBySpeciesId(speciesId);
  }

  @PostMapping("/races")
  public Race add(@RequestBody Race race) {
    return repository.save(race);
  }

  @PutMapping("/races/{id}")
  public ResponseEntity<Race> update(@PathVariable("id") Integer id, @RequestBody Race updatedRace) {
    var race = repository.findById(id);
    if(!race.isPresent())
      return ResponseEntity.notFound().build();

    updatedRace.setId(id);
    repository.save(updatedRace);
    return ResponseEntity.ok().body(updatedRace);
  }
}
```

Figure 7.8 – Code for the races controller

You can find the full code here: `https://github.com/PacktPublishing/The-Ultimate-Docker-Container-Book/blob/main/sample-solutions/ch07/library/src/main/java/com/example/library/controllers/RacesController.java`.

B. A file called `SpeciesController.java` with this code:

```java
package com.example.library.controllers;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import com.example.library.models.Species;
import com.example.library.repositories.SpeciesRepository;

@RestController
public class SpeciesController {
  private final SpeciesRepository repository;

  public SpeciesController(SpeciesRepository repository) {
    this.repository = repository;
  }

  @GetMapping("/species")
  public Iterable<Species> index() {
    return repository.findAll();
  }

  @GetMapping("/species/{id}")
  public ResponseEntity<Species> getById(@PathVariable("id") Integer id) {
    var species = repository.findById(id);
    return species
      .map(r -> ResponseEntity.ok().body(r))
      .orElseGet(() -> ResponseEntity.notFound().build());
  }

  @PostMapping("/species")
  public Species add(@RequestBody Species species) {
    return repository.save(species);
  }

  @PutMapping("/species/{id}")
  public ResponseEntity<Species> update(@PathVariable("id") Integer id, @RequestBody Species updatedSpecies) {
    var race = repository.findById(id);
    if(!race.isPresent())
      return ResponseEntity.notFound().build();

    updatedSpecies.setId(id);
    repository.save(updatedSpecies);
    return ResponseEntity.ok().body(updatedSpecies);
  }
}
```

Figure 7.9 – Code for the species controller

You can find the full code here: `https://github.com/PacktPublishing/The-Ultimate-Docker-Container-Book/blob/main/sample-solutions/ch07/library/src/main/java/com/example/library/controllers/SpeciesController.java`.

9. Finally, we need to do some application configuration. We can do so in the `application.properties` file, which you can find in the `src/main/resources` folder. Add this content to it, which configures the database we are going to use for this example:

Figure 7.10 – Application configuration

We are using the H2 in-memory database with a username of `sa` and no password. We are also making sure to enable the H2 console in our application to have an easy way to inspect the data from our browser (line 6).

10. Now open the `LibraryApplication` class and click the **Run** link above the main method to start the application. Observe the output generated in the terminal:

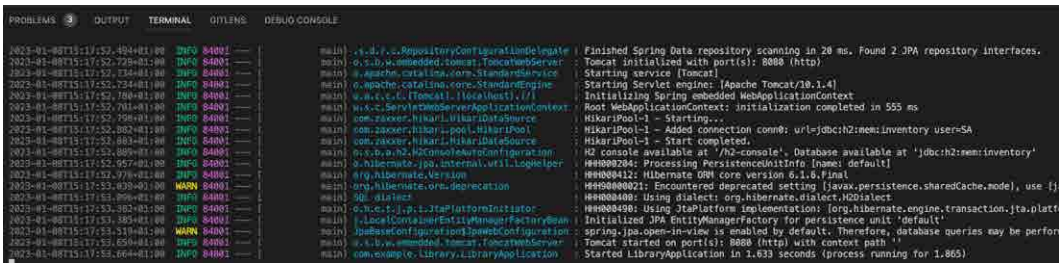


Figure 7.11 – Logging the output of the running library application

Read through the log output and try to make sense of each line. The second-to-last line of the preceding output is telling us that the application can be accessed at port `8080`, which is the default for Spring Boot applications. Also note the line where it says `H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:inventory'`. This indicates that we can now open a browser at `localhost:8080/h2-console` to open the H2 console and, through it, access our in-memory database.

11. Use the Thunder client in VS Code, Postman, or the `curl` command in the terminal to add a species to the database. Here we are using `curl`:

```
$ curl -X POST -d '{"id": 1, "name": "Elephant"}' \
    -H 'Content-Type: application/json' \
    localhost:8080/species
```

The response should look like this:

```
{"id":1,"name":"Elephant","description":null}
```

12. Use `curl` (or any other tool) again to list the species stored in the system:

```
$ curl localhost:8080/species
```

The output should look like this:

```
[{"id":1,"name":"Elephant","description":null}]
```

It is a JSON array with exactly one element.

13. Try all the other `REST` calls that the two controllers we implemented support, such as `PUT` to update an existing species and `GET`, `POST`, and `PUT` for the `/races` endpoint.
14. When done, make sure to stop the application.

Next, we need to package the application into a container and run it:

1. Add a Dockerfile to the root of the library project with this content:

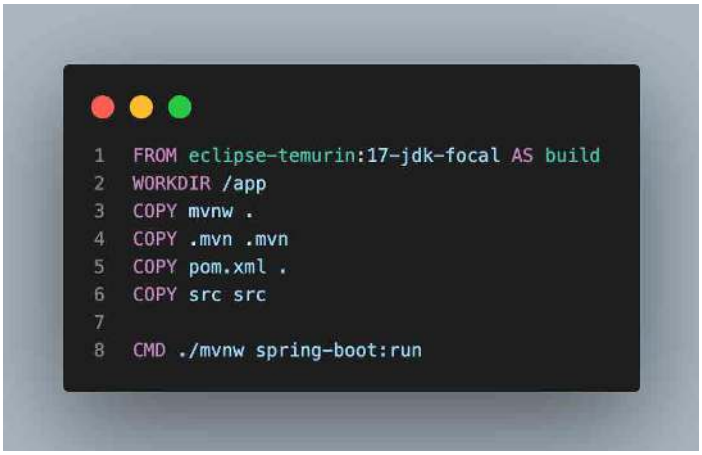


Figure 7.12 – Dockerfile for the library component

2. Create a Docker image using this Dockerfile with this command executed from within the `ch07` folder:

```
$ docker image build -t library library
```

3. Run a container with this command:

```
$ docker container run -d --rm \
    -p 8080:8080 library
```

4. Test that the component now running inside a container still works as expected by using the same commands as in the previous section.
5. When done, stop the container with the library component. We suggest that you use the Docker plugin of VS Code to do so or the dashboard of Docker Desktop.

Now that we have a working example application, we can continue and discuss how we can test this REST API using unit, integration, and black box tests. Let's start with the unit and/or integration tests.

Implementing and running unit and integration tests

Now that we have a working component, it is time to write some tests for it. In this section, we concentrate on unit and integration tests. Spring Boot makes it really simple to get started:

1. To the `src/test/java/com/example/library` folder, add a `LibraryUnitTests.java` file with the following content:

```java
package com.example.library;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class LibraryUnitTests {

  // *** This is our system under test (SUT) ***
  class Calculator {
    public int add(int a, int b) {
      return a + b;
    }
  }

  @Test
  public void assertCanAddNumbers() {
    // arrange
    var calc = new Calculator();
    var expected = 6;
    // act
    var result = calc.add(1, 5);
    // assert
    assertEquals(expected, result);
  }
}
```

Figure 7.13 – Sample unit test written for the library project

Note how we have added a private `Calculator` class to our `Test` class. This is for demonstration purposes only and makes it easier to show how to write a unit test. Normally, one would test classes and their methods that are part of the code base.

Tip

It is a good idea to always structure your tests in a similar way and make it easier for others (and yourself) to read and comprehend those tests. In this case, we have chosen the triple-A (AAA) syntax consisting of Arrange, Act, and Assert. Alternatively, you could use the Given-When-Then syntax.

2. If you have the **Test Runner for Java** extension installed on your VS Code editor, you should now see a green triangle next to the test method (line 19 in the preceding figure). Click it to run the test. As a result, you should see something like this:

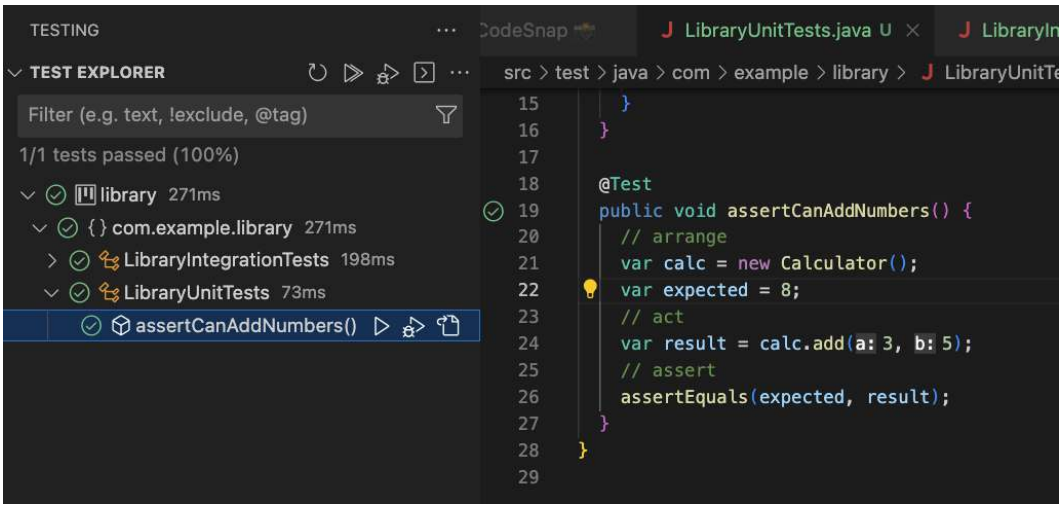


Figure 7.14 – Results of a first test run

Note

Alternatively, you can run the tests from the command line with this command:

```
$ ./mvnw test
```

3. Now let's add a sample integration test. For this, add a file called `LibraryIntegrationTests.java` in the same folder as where you have put the unit tests. We will implement a test using the `MockMvc` helper class provided by Spring Boot to simulate that our application runs on a web server and we're accessing it through its REST endpoints. Add the following content to the test class:

```java
package com.example.library;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@SpringBootTest
@AutoConfigureMockMvc
class LibraryIntegrationTests {

  @Autowired
  private MockMvc mockMvc;

  @Test
  void testAddSpecies() throws Exception {
    var body = "{\"id\":\"1\", \"name\":\"Elephant\"}";
    mockMvc.perform(post("/species")
      .contentType(MediaType.APPLICATION_JSON)
      .content(body))
      .andExpect(status().isOk())
      .andExpect(jsonPath("$.id").exists())
      .andExpect(jsonPath("$.name").value("Elephant"));
  }
}
```

Figure 7.15 – Sample Integration Test written for the library project

4. Run the preceding test the same way as you did with the unit test. Make sure the test passes.

We have finished our preparation and are now ready to package the component into a container and run the unit and integration tests inside the same container. To do this, follow these steps:

1. Let's add a Dockerfile with the following content to the root of our library project. The content is the same that we already used in the previous Java example:

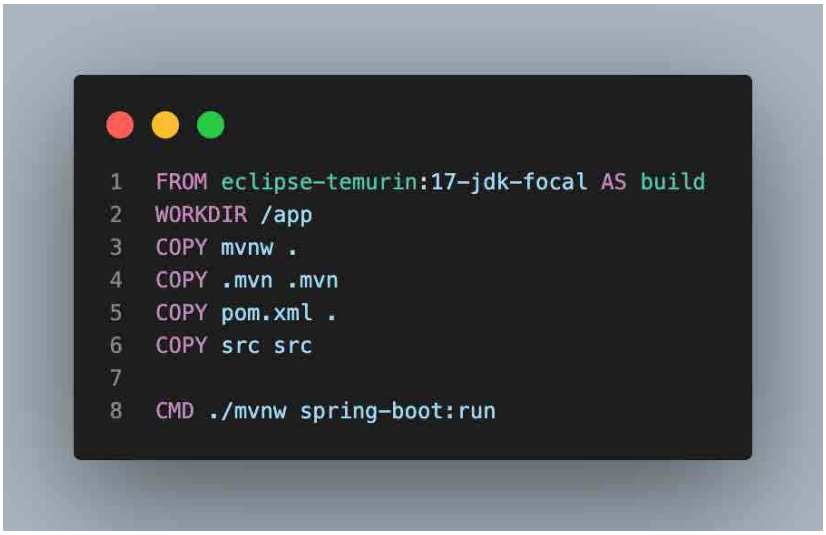


Figure 7.16 – Dockerfile for the library project

2. Then, let's build an image using this Dockerfile:

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

3. Run the tests in the container with the following command:

```
$ docker container run --rm \
    -v $HOME/.m2:/root/.m2
    library ./mvnw test
```

Note the volume mapping we are using. We are sharing our local Maven repository at `$HOME/.m2` with the container, so when building the application, Maven does not have to download all dependencies first as they are already in our local cache. This improves the overall experience massively.

Also note how we override the `CMD` command in our Dockerfile (line 8 in the preceding figure) with `./mvnw test` to run the tests instead of running the application.

4. Observe the output generated. The last few lines of the output should look like this, indicating that tests were run:

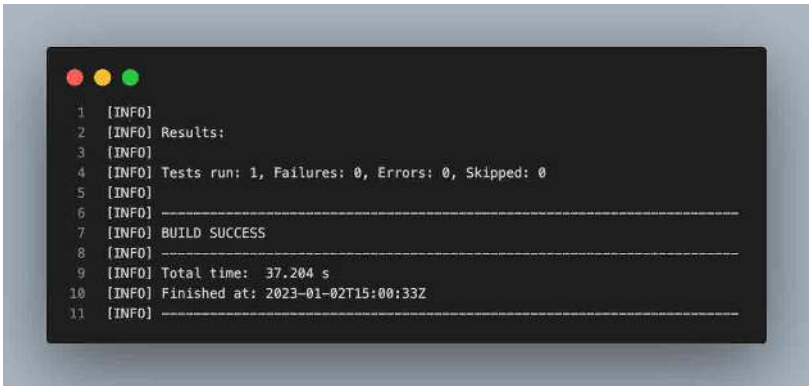


Figure 7.17 – Output of a test run inside the container

In the same way that you have now run the unit and integration tests inside a container locally on your laptop, you can also run it during the CI phase of your CI/CD pipeline. A simple shell script is enough to automate what you just did manually.

Implementing and running black box tests

Since black box tests have to deal with the SUT as a closed system, the tests should not run inside the same container as the component itself. It is instead recommended to run the test code in its own dedicated test container.

It is also recommended to not intermingle the code of black box tests and the component but to keep them strictly separate. We will demonstrate this by writing the tests in a different language than the component. This time, we will use C#. Any language will do such as Kotlin, Node.js, or Python.

In this example, we will use .NET and C# to implement the component tests:

1. From within the `ch07` folder, execute the following command to create a test project:

   ```
   $ dotnet new xunit -o library-component-tests
   ```

   This will create a test project in the `library-component-tests` subfolder using the popular `xunit` test library.

2. Try to run the tests with the following command:

   ```
   $ dotnet test library-component-tests
   ```

   The (shortened) output should look like this:

   ```
   Passed!  - Failed:  0, Passed:  1, Skipped:  0, Total:  1, …
   ```

   This indicates that all tests passed. Of course, by default, there exists only an empty sample test in the project at this time.

3. Open this project in VS Code with the following:

```
$ code library-component-tests
```

4. Locate the `UnitTest1.cs` file and open it. At the top of the file, add this statement:

```
using System.Text.Json;
```

5. Right after the `namespace` declaration, add this record definition:

```
public record Species(int id, string name, string description);
```

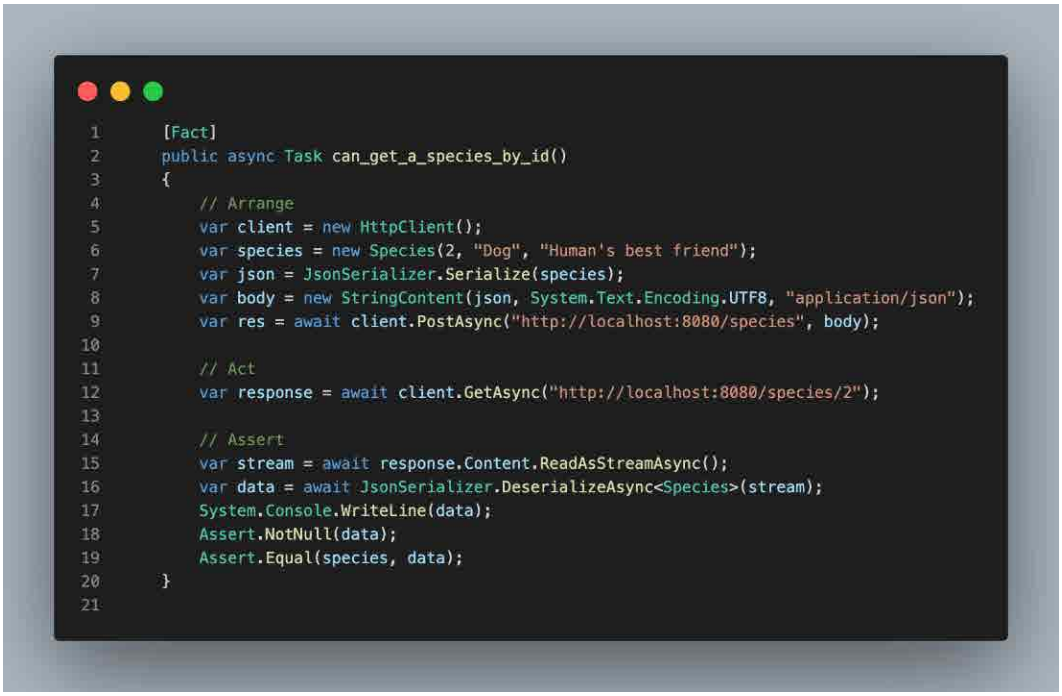
6. Now add a new method called `can_add_species`, looking like this:

```
[Fact]
    public async Task can_add_species()
    {
        // Arrange
        var client = new HttpClient();
        var url = "http://localhost:8080/species";
        var species = new Species(1, "Elephant", "The big gray mammal");
        var json = JsonSerializer.Serialize(species);
        var body = new StringContent(json, System.Text.Encoding.UTF8, "application/json");

        // act
        var response = await client.PostAsync(url, body);

        // assert
        Assert.Equal(System.Net.HttpStatusCode.OK, response.StatusCode);
    }
```

Figure 7.18 – Component test to add a species

Here we are using the `HttpClient` class to post a data object of type `Species` to the `/species` endpoint. We are then asserting that the HTTP response code for the operation is `OK (200)`. Note how we are using the AAA convention to structure our test.

7. Add another method called `can_get_a_species_by_id` with the following content:



```
[Fact]
public async Task can_get_a_species_by_id()
{
    // Arrange
    var client = new HttpClient();
    var species = new Species(2, "Dog", "Human's best friend");
    var json = JsonSerializer.Serialize(species);
    var body = new StringContent(json, System.Text.Encoding.UTF8, "application/json");
    var res = await client.PostAsync("http://localhost:8080/species", body);

    // Act
    var response = await client.GetAsync("http://localhost:8080/species/2");

    // Assert
    var stream = await response.Content.ReadAsStreamAsync();
    var data = await JsonSerializer.DeserializeAsync<Species>(stream);
    System.Console.WriteLine(data);
    Assert.NotNull(data);
    Assert.Equal(species, data);
}
```

Figure 7.19 – Component test to read a species by ID

8. Before you proceed and run the tests, make sure the `library` component is running and listening at port `8080`. Otherwise, the tests will fail, since nobody is listening at the expected endpoints. Use this command:

```
$ docker container run --rm \
    -v $HOME/.m2:/root/.m2
    library
```

9. Run the tests with this command:

```
$ dotnet test
```

   Make sure the two tests pass.

10. **Optional**: Add additional component tests testing the other endpoints of the `library` component.
11. When done, stop the `library` component.

Next, we are going to show how we can run the tests in a container:

1. Add a Dockerfile with the following content to the root of the .NET test project:

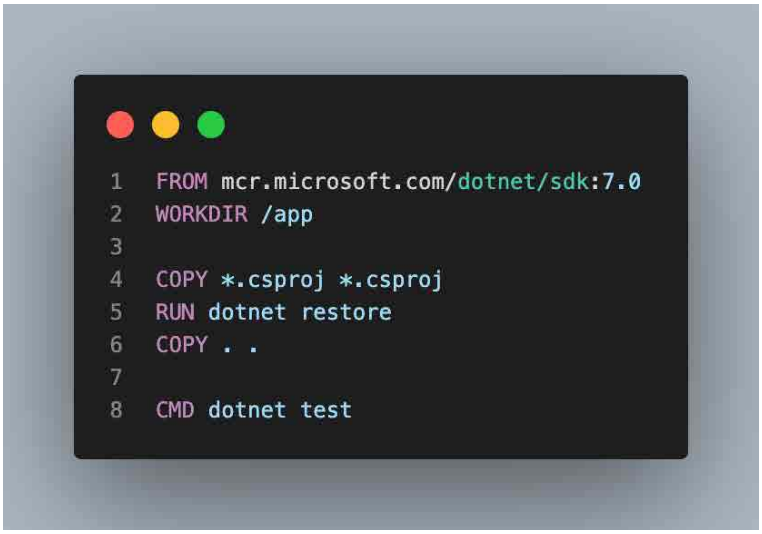


Figure 7.20 – Dockerfile for the component tests

2. Create an image with this Dockerfile. From within the `ch07` folder, use this command:

```
$ docker image build -t library-component-tests \
    library-component-tests
```

3. Double-check that we already have a Docker image created for the library component. If not, use this command to do so from within the `ch07` folder:

```
$ docker image build -t library library
```

4. Now that we have a Docker image for the library component and one for the component tests, we need to run a container of each:

   I. To run the library component, use this:

```
$ docker container run -d --rm \
    -p 8080:8080 library
```

   II. To run the component tests, use this command:

```
$ docker container run library-component-tests
```

   Observe that the tests are executed and are all passing.

5. When done, remove the two containers. Use your Docker plugin in VS Code or the dashboard of Docker Desktop to do so.

Well, that was quite a run. We have shown how to write unit and integration tests for a component written in Java and using Spring Boot 3. We ran the tests natively on our laptop and also inside a container. Then we showed how to create some black box tests in .NET 7, C# and ran them against our library component. We did this again natively on our laptop and then ran the component and the black box tests each in their own container.

Next, we are going to discuss how to best set up a testing environment.

Best practices for setting up a testing environment

In this section, we want to list a few best practices for setting up a testing environment for applications running in containers, including considerations for network isolation, data management, and resource constraints:

- **Use a separate testing environment**: It is generally a good idea to use a separate testing environment for running tests in containers rather than running tests on the same host as your production environment. This can help to prevent any potential issues or disruptions from affecting your production environment.
- **Isolate the testing network**: To ensure that your testing environment is isolated from your production environment, it is a good idea to use a separate network for testing. This can be achieved by using a separate virtual network or by using network namespaces or overlays in your container runtime.
- **Manage test data carefully**: When testing applications in containers, it is important to manage test data carefully to ensure that your tests are reliable and repeatable. This can involve using test data generation tools, snapshotting the test data, or using a separate test database.
- **Use resource constraints**: To ensure that your tests are reliable and consistent, it is a good idea to use resource constraints (e.g., CPU, memory) to limit the resources available to your containers. This can help to prevent resource contention and ensure that your tests are not impacted by external factors such as the load on the host system.
- **Use a container orchestration tool**: To manage a large number of containers and ensure that they are deployed and scaled consistently, it is a good idea to use a container orchestration tool such as Kubernetes or Docker Swarm. These tools can help to automate the process of deploying and scaling containers and can provide features such as automatic rollbacks and self-healing.
- **Monitor the testing environment**: To ensure that your testing environment is running smoothly and to identify any issues that may arise, it is a good idea to use monitoring tools to track the performance and resource usage of your containers. This can help you to identify and fix any issues that may affect the reliability of your tests.

Now when testing, you may face some troubles and hard-to-explain test failures. In the next section, we're going to provide a few tips about what you can do in such a situation.

Tips for debugging and troubleshooting issues

As we are running automated tests in our containerized environments, we may from time to time face seemingly weird behaviors and mysteriously failing tests. Here are some tips for debugging and troubleshooting issues that may arise when testing applications in containers:

- **Check the logs**: The first step in debugging any issue is to check the logs for any error messages or other clues about the cause of the problem. In a containerized environment, you can use tools such as `docker container logs` to view the logs for a specific container.
- **Use a debugger**: If the error message or log output is not sufficient to diagnose the problem, you can use a debugger to inspect the state of the application at runtime. Many IDEs, such as VS Code, which we use all the time, Visual Studio, and IntelliJ, have built-in support for debugging applications running in containers.
- **Inspect the container environment**: If the issue appears to be related to the container environment itself (e.g., a missing dependency or configuration issue), you can use tools such as `docker container exec` to run commands inside the container and inspect its environment.
- **Use a container runtime debugger**: Some container runtimes, such as Docker, provide tools for debugging issues with the container itself (e.g., resource usage and networking issues). These tools can be helpful for diagnosing issues that are specific to the container runtime.
- **Use a containerized debugging environment**: If you are having difficulty reproducing the issue in a local development environment, you can use a containerized debugging environment (e.g., a debugger container) to replicate the production environment more closely.
- **Check for known issues**: If you are using third-party libraries or dependencies in your application, it is worth checking whether there are any known issues or bugs that could be causing the problem. Many libraries and dependencies maintain lists of known issues and workarounds on their website or in their documentation.
- **Get help**: If you are unable to diagnose the issue on your own, don't hesitate to seek help from the community, for example, from Stack Overflow or the maintainers of the libraries and tools you are using. There are many resources available online.

Now let's discuss a few challenges that may occur during testing and what we should consider when testing.

Challenges and considerations when testing applications running in containers

Next to all the many advantages that testing applications running in containers brings to the table, we need to also have a brief discussion of the challenges and considerations involved in this type of testing, such as dealing with dependencies and managing test data:

- **Isolation**: Testing applications in containers can provide a level of isolation between the test environment and the host system, which can be useful for ensuring that the test results are consistent and repeatable. However, this isolation can also make it more difficult to debug issues and identify the root cause of problems, as you may not have access to the host system and its resources.
- **Environment consistency**: Ensuring that the test environment is consistent across different development environments can be a challenge when using containers. Differences in the host system, container runtime, and network configuration can all impact the behavior of the application and the test results.
- **Data management**: Managing test data in a containerized environment can be challenging, as you may need to ensure that the test data is consistent and available to all containers, or that it is properly isolated and not shared between tests.
- **Resource constraints**: Testing applications in containers can be resource-intensive, as you may need to run multiple containers in parallel to test different scenarios. This can lead to resource contention and may require careful resource management to ensure that your tests are reliable and consistent.
- **Integration testing**: Testing the integration between multiple containers can be challenging, as you may need to coordinate the startup and shutdown of multiple containers and ensure that they can communicate with each other.
- **Performance testing**: Testing the performance of applications running in containers can be difficult, as the performance may be impacted by the host system, the container runtime, and the network configuration.

Overall, testing applications running in containers requires careful planning and consideration to ensure that the test environment is consistent and reliable, and to ensure that the test results are meaningful and actionable.

Before we end this chapter, let's look at a few case studies where companies are using containerized tests.

Case studies

In this last section of the chapter, we present a few case studies and examples of organizations that have successfully implemented testing strategies for applications running in containers:

1. An automated testing technique was introduced by a well-known online shop to boost the effectiveness and efficiency of its software development process. The company was able to considerably reduce the time and effort needed to test its applications by automating the execution of functional, integration, and acceptance tests. As a result, it was able to provide customers with new features and upgrades more rapidly and reliably.
2. Automated testing was used by a financial services company to enhance the dependability and stability of their trading platform. The business was able to find and fix problems early in the development process by automating the execution of unit, integration, and acceptance tests, minimizing the risk of downtime and enhancing customer satisfaction.
3. Automated testing was used by a healthcare organization to guarantee the precision and dependability of its **electronic medical record** (**EMR**) system. The business was able to swiftly identify and address problems by automating the execution of functional and acceptability tests, increasing the EMR system's dependability and trustworthiness, and lowering the risk of mistakes and patient harm.

The advantages of automated testing, such as better quality, quicker development and deployment cycles, increased reliability, and higher customer happiness, are illustrated by these case studies.

Summary

In this chapter, we learned about the benefits of testing applications running in containers, discussed the different types of testing, presented some of the tools and technologies commonly used for testing, as well as best practices for setting up a testing environment. We also presented a list of tips for debugging and troubleshooting issues, talked about challenges and considerations when testing applications running in containers, and concluded the chapter with a list of case studies.

In the next chapter, we will introduce miscellaneous tips, tricks, and concepts useful when containerizing complex distributed applications or when using Docker to automate sophisticated tasks.

Questions

To assess your learning, please try to answer the following questions before you proceed to the next chapter:

1. How do we run unit tests for an application inside a container?
2. Should the Docker images that we use in production contain test code? Justify your answer.
3. Where do we typically run unit and integration tests that run inside a container?
4. List a few advantages of running unit and integration tests in containers.
5. What are a few challenges you may face if running tests in containers?

Answers

Here are sample answers to the questions of this chapter:

1. We have learned how to run an application in a container. We have seen examples written in Node.js, Python, Java, and .NET C#. We have learned how the Dockerfile must look to create an image. Specifically, we have learned how to define the startup command to execute when a container is created from such an image. In the case of a Java application, this could be as follows:

   ```
   CMD java -jar /app/my-app.jar
   ```

   For a Node.js application, it could be as follows:

   ```
   CMD node index.js
   ```

   To run the unit tests for the application, we just have to use a different startup command.

2. We strongly advise against shipping test code to a production environment. Tests bloat the Docker image, which has several negative side effects, such as the following:

   - Providing a bigger surface for hacker attacks
   - Longer startup times for the container since it takes longer to load an image from storage into the memory of the container host
   - Longer download times and higher network usage due to the increased size of the image

3. Unit and integration tests are typically run on the developer's local machine before they push code to a code repository such as GitHub. Once the code is pushed to GitHub or any other remote code repository, usually the CI/CD pipeline kicks in and the CI stage is executed. Part of this stage is the execution of all unit and integration tests against the application. Usually, this is performed on a so-called build agent. In many cases, this is a sandbox environment where Docker containers can be run. Thus, the CI stage uses the same technique to run the tests in the build agent as a developer would do locally. It is important to note that tests other than some special smoke tests are never run in a production environment, since this could have undesired side effects.
4. One of the most important advantages of running tests in containers is the isolation aspect. We can run the tests on any environment able to run containers and do not have to worry about installing frameworks or libraries on the hosting machine first.

   Another important advantage is that running tests in containers makes them repeatable out of the box. Each time a container containing the application code and the tests are started, the boundary conditions are the same. With this, we guarantee consistency in the test execution. Were we to run the tests natively on the host, we would have a harder time guaranteeing this consistency.
5. Some challenges we may face when running our tests inside containers are as follows:

   - It may be harder to troubleshoot and debug failing tests
   - Integration testing can be more challenging when several containers are involved in the necessary setup
   - Resources such as CPU, RAM, and network bandwidth can be limited in a containerized environment (via cgroup settings) and thus negatively impact your test runs

8

Increasing Productivity with Docker Tips and Tricks

This chapter introduces miscellaneous tips, tricks, and concepts that are useful when containerizing complex distributed applications or when using Docker to automate sophisticated tasks. You will also learn how to leverage containers to run your whole development environment in them. Here is the list of topics we are going to discuss:

- Keeping your Docker environment clean
- Using a `.dockerignore` file
- Executing simple admin tasks in a container
- Limiting the resource usage of a container
- Avoiding running a container as `root`
- Running Docker from within Docker
- Optimizing your build process
- Scanning for vulnerabilities and secrets
- Running your development environment in a container

After reading this chapter, you will have learned how to do the following:

- Successfully restore your Docker environment after it has been messed up completely
- Use a `.dockerignore` file to speed up builds, reduce image size, and enhance security
- Run various tools to perform tasks on your computer without installing them
- Limit the number of resources a containerized application uses during runtime
- Harden your system by not running containers as root

- Enable advanced scenarios by running Docker inside of a Docker container
- Accelerate and improve the build process of your custom Docker images
- Scan your Docker images for common vulnerabilities and exposures and the accidental inclusion of secrets
- Run a whole development environment inside a container running locally or remotely

Let's get started!

Technical requirements

In this chapter, if you want to follow along with the code, you need Docker Desktop installed on your local machine as well as the Visual Studio Code editor.

Before we start, let's create a folder for the samples that we will be using during this part of the book. Open a new terminal window and navigate to the folder you clone the sample code to. Usually, this is `~/The-Ultimate-Docker-Container-Book`:

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

Create a new subfolder for *Chapter 8* called `ch08` and navigate to it:

```
$ mkdir ch08 && cd ch08
```

Now that you are ready, let's start with the tips and tricks on how to keep our Docker environment clean.

You can find the sample code here: `https://github.com/PacktPublishing/The-Ultimate-Docker-Container-Book/tree/main/sample-solutions/ch08`.

Keeping your Docker environment clean

First, we want to learn how we can delete dangling images. A dangling Docker image is an unused image that has no association with any tagged images or containers. It usually occurs when a new image is built using the same tag as an existing image. Instead of removing the old image, Docker preserves it but removes the tag reference, leaving the image without a proper tag.

Dangling images are not referenced by any container or tagged image, and they consume disk space without providing any benefit. They can accumulate over time, especially in environments with frequent image builds and updates. Thus, it is better to remove them from time to time. Here is the command to do so:

```
$ docker image prune -f
```

Please note that we have added the `-f` (or `--force`) parameter to the `prune` command. This is to prevent the CLI from asking you to confirm that you really want to delete those superfluous layers.

Stopped containers can waste precious resources too. If you're sure that you don't need these containers anymore, then you should remove them. You can remove them individually with the following command:

```
$ docker container rm <container-id|container-name>
```

You can also remove them as a batch by using this command:

```
$ docker container prune --force
```

In the command for removing them individually, `<container-id|container-name>` means that we can either use the container ID or its name to identify the container.

Unused Docker volumes can also quickly fill up disk space. It is good practice to tender your volumes, specifically in a development or **Continuous Integration** (**CI**) environment where you create a lot of mostly temporary volumes. But I have to warn you, Docker volumes are meant to store data. Often, this data must live longer than the life cycle of a container. This is specifically true in a production or production-like environment where the data is often mission-critical. Hence, be 100% sure of what you're doing when using the following command to prune volumes on your Docker host:

```
$ docker volume prune
   WARNING! This will remove all local volumes not used by at least
one container.
   Are you sure you want to continue? [y/N]
```

We recommend using this command without the `-f` (or `--force`) flag. It is a dangerous and terminal operation and it's better to give yourself a second chance to reconsider your action. Without the flag, the CLI outputs the warning you see in the preceding command. You have to explicitly confirm by typing `y` and pressing the *Enter* key.

On production or production-like systems, you should abstain from the preceding command and rather delete unwanted volumes one at a time by using this command:

```
$ docker volume rm <volume-name>
```

I should also mention that there is a command to prune Docker networks. But since we have not yet officially introduced networks, I will defer this to *Chapter 10*, *Using Single-Host Networking*.

In the next section, we are going to show how we can exclude some folders and files from being included in the build context for a Docker image.

Using a .dockerignore file

The `.dockerignore` file is a text file that tells Docker to ignore certain files and directories when building a Docker image from a Dockerfile. This is similar to how the `.gitignore` file works in Git.

The primary benefit of using a `.dockerignore` file is that it can significantly speed up the Docker build process. When Docker builds an image, it first sends all of the files in the current directory (known as the "build context") to the Docker daemon. If this directory contains large files or directories that aren't necessary for building the Docker image (such as log files, local environment variables, cache files, etc.), these can be ignored to speed up the build process.

Moreover, using a `.dockerignore` file can help to improve security and maintain clean code practices. For instance, it helps prevent potentially sensitive information (such as `.env` files containing private keys) from being included in the Docker image. It can also help to keep the Docker image size minimal by avoiding unnecessary files, which is particularly beneficial when deploying the image or transferring it across networks.

Here's an example of a `.dockerignore` file:

```
# Ignore everything
**
# Allow specific directories
!my-app/
!scripts/
# Ignore specific files within allowed directories
my-app/*.log
scripts/temp/
```

In this example, all files are ignored except those in the `my-app/` and `scripts/` directories. However, log files within `my-app/` and all files in the `scripts/temp/` subdirectory are ignored. This level of granularity provides developers with fine control over what is included in the Docker build context.

In conclusion, the use of a `.dockerignore` file is a best practice for Docker builds, helping to speed up builds, reduce image size, and enhance security by excluding unnecessary or sensitive files from the build context. In the next section, we are going to show how to execute simple administrative tasks within a Docker container.

Executing simple admin tasks in a container

In this section, we want to provide a few examples of tasks you may want to run in a container instead of natively on your computer.

Running a Perl script

Let's assume you need to strip all leading whitespaces from a file and you found the following handy Perl script to do exactly that:

```
$ cat sample.txt | perl -lpe 's/^\s*//'
```

As it turns out, you don't have Perl installed on your working machine. What can you do? Install Perl on the machine? Well, that would certainly be an option, and it's exactly what most developers or system admins do. But wait a second, you already have Docker installed on your machine. Can't we use Docker to circumvent the need to install Perl? And can't we do this on any operating system supporting Docker? Yes, we can. This is how we're going to do it:

1. Navigate to the chapter's code folder:

```
$ cd ~/The-Ultimate-Docker-Container-Book/ch08
```

2. Create a new subfolder called `simple-task`, and navigate to it:

```
$ mkdir simple-task && cd simple-task
```

3. Open VS Code from within this folder:

```
$ code .
```

4. In this folder, create a `sample.txt` file with the following content:

```
            1234567890
               This is some text
                another line of text
          more text
final line
```

   Please note the whitespaces at the beginning of each line. Save the file.

5. Now, we can run a container with Perl installed in it. Thankfully, there is an official Perl image on Docker Hub. We are going to use the slim version of the image. The primary difference between the normal Perl Docker image and the slim version lies in their size and the components included in the images. Both images provide the Perl runtime environment, but they are optimized for different use cases:

```
$ docker container run --rm -it \
    -v $(pwd):/usr/src/app \
    -w /usr/src/app \
    perl:slim sh -c "cat sample.txt | perl -lpe 's/^\s*//'"
```

   The preceding command runs a Perl container (`perl:slim`) interactively, maps the content of the current folder into the `/usr/src/app` folder of the container, and sets the working folder inside the container to `/usr/src/app`. The command that is run inside the container is as follows:

```
sh -c "cat sample.txt | perl -lpe 's/^\s*//'"
```

   It basically spawns a Bourne shell and executes our desired Perl command.

6. Analyze the output generated by the preceding command. It should look like this:

```
1234567890
This is some text
another line of text
more text
final line
```

That is, all trailing blanks have been removed.

Without needing to install Perl on our machine, we were able to achieve our goal. The nice thing is that, after the script has run, the container is removed from your system without leaving any traces because we used the `--rm` flag in the `docker container run` command, which automatically removes a stopped container.

> **Tip**
>
> If that doesn't convince you yet because, if you're on macOS, you already have Perl installed, then consider you're looking into running a Perl script named `your-old-perl-script.pl` that is old and not compatible with the newest release of Perl that you happen to have installed on your system. Do you try to install multiple versions of Perl on your machine and potentially break something? No, you just run a container with the (old) version of Perl that is compatible with your script, as in this example:
>
> ```
> $ docker container run -it --rm \
>     -v $(pwd):/usr/src/app \
>     -w /usr/src/app \
>     perl:<old-version> perl your-old-perl-script.pl
> ```
>
> Here, `<old-version>` corresponds to the tag of the version of Perl that you need to run your script.

In the next section, we are going to demonstrate how to run a Python script.

Running a Python script

A lot of people use quick and dirty Python scripts or mini apps to automate tasks that are not easily coded with, say, Bash. Now, if the Python script has been written in Python 3.x and you only happen to have Python 2.7 installed or no version at all on your machine, then the easiest solution is to execute the script inside a container. Let's assume a simple example where the Python script counts lines, words, and letters in a given file and outputs the result to the console:

1. Still in the `simple-task` folder, add a `stats.py` file and add the following content:

```python
import sys

fname = sys.argv[1]
lines = 0
words = 0
letters = 0
for line in open(fname):
    lines += 1
    letters += len(line)
    pos = 'out'
    for letter in line:
        if letter != ' ' and pos == 'out':
            words += 1
            pos = 'in'
        elif letter == ' ':
            pos = 'out'

print("Lines:", lines)
print("Words:", words)
print("Letters:", letters)
```

Figure 8.1 – Python script to calculate statistics of a sample text

2. After saving the file, you can run it with the following command:

```
$ docker container run --rm -it \
    -v $(pwd):/usr/src/app \
    -w /usr/src/app \
     python:3-alpine python stats.py sample.txt
```

3. Note that, in this example, we are reusing the `sample.txt` file from the previous *Running a Perl script* section. The output in my case is as follows:

```
Lines: 5
Words: 13
Letters: 121
```

The beauty of this approach is that the Perl script before and this last Python script will now run on any computer with any OS installed, as long as the machine is a Docker host and hence, can run containers.

Next, we are going to learn how to limit the number of resources a container running on the system can consume.

Limiting the resource usage of a container

One of the great features of a container, apart from encapsulating application processes, is the possibility of limiting the resources a single container can consume at most. This includes CPU and memory consumption. Let's have a look at how limiting the amount of memory (RAM) works:

```
$ docker container run --rm -it \
       --name stress-test \
       --memory 512M \
       ubuntu:22.04 /bin/bash
```

Once inside the container, install the `stress` tool, which we will use to simulate memory pressure:

```
/# apt-get update && apt-get install -y stress
```

Open another terminal window and execute the `docker stats` command to observe the resource consumption of all running Docker containers. You should see something like this:

Figure 8.2 – The Docker stats showing a resource-limited container

Look at `MEM USAGE` and `LIMIT`. Currently, the container uses only `36.57MiB` memory and has a limit of `512MiB`. The latter corresponds to what we have configured for this container. Now, let's use the `stress` tool to simulate three workers, which will allocate memory using the `malloc()` function in blocks of `256MiB`. Run this command inside the container to do so:

```
/# stress -m 3
```

The preceding command puts stress on the system's memory by creating three child processes that will `malloc()` and touch memory until the system runs out of memory. In the terminal running Docker stats, observe how the value for `MEM USAGE` approaches but never exceeds `LIMIT`. This is exactly the behavior we expected from Docker. Docker uses Linux **cgroups** to enforce those limits.

> **What are cgroups?**
>
> Linux **cgroups**, short for **control groups**, is a kernel-level feature that allows you to organize processes into hierarchical groups, and to allocate, restrict, and monitor system resources such as CPU, memory, disk I/O, and network among these groups. Cgroups provide a way to manage and limit the resource usage of processes, ensuring fair distribution and preventing individual processes from monopolizing system resources.

We could similarly limit the amount of CPU a container can consume with the `--cpu` switch.

With this operation, engineers can avoid the noisy neighbor problem on a busy Docker host, where a single container starves all of the others by consuming an excessive amount of resources.

Avoiding running a container as root

Most applications or application services that run inside a container do not need `root` access. To tighten security, it is helpful in those scenarios to run these processes with minimal necessary privileges. These applications should not be run as `root` nor assume that they have `root`-level privileges.

Once again, let's illustrate what we mean with an example. Assume we have a file with top-secret content. We want to secure this file on our Unix-based system using the `chmod` tool so that only users with `root` permissions can access it. Let's assume I am logged in as `demo` on the dev host and hence my prompt is `demo@dev $`. I can use `sudo su` to impersonate a superuser. I have to enter the superuser password though:

```
demo@dev $ sudo su
   Password: <root password>
   root@dev $
```

Now, as the `root` user, I can create this file called `top-secret.txt` and secure it:

```
root@dev $ echo "You should not see this." > top-secret.txt
root@dev $ chmod 600 ./top-secret.txt
root@dev $ exit
demo@dev $
```

If I try to access the file as user `demo`, the following happens:

```
cat: ./top-secret.txt: Permission denied
```

I get a `Permission denied` message, which is what we wanted. No other user except `root` can access this file. Now, let's build a Docker image that contains this secured file and when a container is created from it, tries to output the content of the `secrets` file. The Dockerfile could look like this:

```
FROM ubuntu:22.04
COPY ./top-secret.txt /secrets/
# simulate use of restricted file
CMD cat /secrets/top-secret.txt
```

We can build an image from that Dockerfile (as `root`!) with the following:

```
demo@dev $ sudo su
Password: <root password>
root@dev $ docker image build -t demo-image .
root@dev $ exit
demo@dev $
```

Then, by running a container with the image built in the previous step, we get the following:

```
demo@dev $ docker container run demo-image
```

The preceding command will generate this output:

```
You should not see this.
```

OK, so although I am impersonating the `demo` user on the host and running the container under this user account, the application running inside the container automatically runs as `root`, and hence has full access to protected resources. That's bad, so let's fix it! Instead of running with the default, we define an explicit user inside the container. The modified Dockerfile looks like this:

```
FROM ubuntu:22.04
RUN groupadd -g 3000 demo-group |
       && useradd -r -u 4000 -g demo-group demo-user
USER demo-user
COPY ./top-secret.txt /secrets/
# simulate use of restricted file
CMD cat /secrets/top-secret.txt
```

We use the `groupadd` tool to define a new group, `demo-group`, with the ID `3000`. Then, we use the `useradd` tool to add a new user, `demo-user`, to this group. The user has the ID `4000` inside the container. Finally, with the `USER demo-user` statement, we declare that all subsequent operations should be executed as `demo-user`.

Rebuild the image—again, as `root`—and then try to run a container from it:

```
demo@dev $ sudo su
Password: <root password>
root@dev $ docker image build -t demo-image .
root@dev $ exit
demo@dev $ docker container run demo-image \
     cat: /secrets/top-secret.txt:
Permission denied
```

And as you can see on the last line, the application running inside the container runs with restricted permissions and cannot access resources that need root-level access. By the way, what do you think would happen if I ran the container as `root`? Try it out!

In the next section, we are going to show how we can automate Docker from within a container.

Running Docker from within Docker

At times, we may want to run a container hosting an application that automates certain Docker tasks. How can we do that? Docker Engine and the Docker CLI are installed on the host, yet the application runs inside the container. Well, from early on, Docker has provided a means to bind-mount Linux sockets from the host into the container. On Linux, sockets are used as very efficient data communications endpoints between processes that run on the same host. The Docker CLI uses a socket to communicate with Docker Engine; it is often called the **Docker socket**. If we can give access to the Docker socket to an application running inside a container, then we can just install the Docker CLI inside this container, and we will then be able to run an application in the same container that uses this locally installed Docker CLI to automate container-specific tasks.

Important note

Here, we are not talking about running Docker Engine inside the container but rather only the Docker CLI and bind-mounting the Docker socket from the host into the container so that the CLI can communicate with Docker Engine running on the host computer. This is an important distinction.

Running Docker Engine inside a container is generally not recommended due to several reasons, including security, stability, and potential performance issues. This practice is often referred to as **Docker-in-Docker** or **DinD**. The main concerns are as follows:

- **Security**: Running Docker Engine inside a container requires elevated privileges, such as running the container in privileged mode or mounting the Docker socket. This can expose the host system to potential security risks, as a compromised container could gain control over the host's Docker daemon and escalate privileges, affecting other containers and the host itself.
- **Stability**: Containers are designed to be isolated, lightweight, and ephemeral. Running Docker Engine inside a container can create complex dependencies and increase the chances of conflicts or failures, particularly when managing storage, networking, and process namespaces between the host and the nested container environment.
- **Performance**: Running Docker Engine inside a container can introduce performance overhead, as it adds another layer of virtualization, particularly in terms of storage and networking. This can lead to increased latency and reduced throughput, particularly when managing large numbers of containers or when working with high-performance applications.
- **Resource management**: Docker-in-Docker can make it challenging to manage and allocate resources effectively, as nested containers may not inherit resource limits and restrictions from their parent container, leading to potential resource contention or over-commitment on the host.

To illustrate the concept, let's look at an example using the preceding technique. We are going to use a copy of the `library` component we built in the previous chapter (*Chapter 7*) for this:

1. Navigate to the chapter folder:

```
$ cd ~/The-Ultimate-Docker-Container-Book/ch08
```

2. Copy the `library` component from the `ch07` directory to this folder:

```
$ cp -r ../ch07/library .
```

3. Open the component in VS Code:

```
$ code library
```

4. Add a new file called `pipeline.sh` to the root of the project and add the following code to it, which automates the building, testing, and pushing of a Docker image:

```
#! /bin/bash

# *** Sample script to build, test & push         ***
# *** containerized Java/Spring Boot applications ***

# build the Docker image
docker image build -t $HUB_USER/$REPOSITORY:$TAG .
# Run all unit tests
docker container run $HUB_USER/$REPOSITORY:$TAG ./mvnw test
# Login to Docker Hub
docker login -u $HUB_USER -p $HUB_PWD
# Push the image to Docker Hub
docker image push $HUB_USER/$REPOSITORY:$TAG
```

Figure 8.3 – Script to build, test, and push a Java application

Note that we're using four environment variables: `$HUB_USER` and `$HUB_PWD` being the credentials for Docker Hub, and `$REPOSITORY` and `$TAG` being the name and tag of the Docker image we want to build. Eventually, we will have to pass values for those environment variables in the `docker container run` command, so that they are available for any process running inside the container.

5. Save the file and make it an executable:

```
$ chmod +x ./pipeline.sh
```

We want to run the `pipeline.sh` script inside a builder container. Since the script uses the Docker CLI, our builder container must have the Docker CLI installed, and to access Docker Engine, the builder container must have the Docker socket bind-mounted.

Let's start creating a Docker image for such a builder container:

1. Add a file called `Dockerfile.builder` to the root of the project and add the following content to it:

```
1  FROM eclipse-temurin:17-jdk-focal AS build
2
3  RUN apt-get update \
4      && apt-get install -y ca-certificates curl gnupg lsb-release \
5      && mkdir -p /etc/apt/keyrings \
6      && curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
7         gpg --dearmor -o /etc/apt/keyrings/docker.gpg \
8      && echo \
9         "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
10        https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" \
11        | tee /etc/apt/sources.list.d/docker.list > /dev/null \
12     && apt-get update \
13     && apt-get install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin
14
15 WORKDIR /app
16 COPY mvnw .
17 COPY .mvn .mvn
18 COPY pom.xml .
19 COPY src src
20 COPY pipeline.sh .
21
22 CMD ./mvnw spring-boot:run
```

Figure 8.4 – Dockerfile for the builder

Note the long `RUN` command on line 3 onward. This is needed to install Docker in the container. For more details about this command, you may want to consult the Docker online documentation here: `https://docs.docker.com/engine/install/ubuntu/`.

2. Building a Docker image with this Dockerfile is straightforward:

```
$ docker image build -f Dockerfile.builder -t builder .
```

3. We are now ready to try the `builder` command with a real Java application; for example, let's take the sample app we defined in the `ch08/library` folder. Make sure you replace `<user>` and `<password>` with your own credentials for Docker Hub:

```
docker container run --rm \
    --name builder \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -v "$PWD":/usr/src/app \
    -e HUB_USER=<user> \
    -e HUB_PWD=<password> \
    -e REPOSITORY=ch08-library \
    -e TAG=1.0 \
    builder
```

Figure 8.5 – Docker run command for the builder

Notice how, in the preceding command, we mounted the Docker socket into the container with `-v /var/run/docker.sock:/var/run/docker.sock`. If everything goes well, you should have a container image built for the sample application, the test should have been run, and the image should have been pushed to Docker Hub. This is only one of the many use cases where it is very useful to be able to bind-mount the Docker socket.

A special notice to those of you who want to try Windows containers on a Windows computer: on Docker Desktop for Windows, you can create a similar environment by bind-mounting Docker's **named pipe** instead of a socket. A named pipe on Windows is roughly the same as a socket on a Unix- based system. Assuming you're using a PowerShell terminal, the command to bind-mount a named pipe when running a Windows container hosting Jenkins looks like this:

```
PS> docker container run `
       --name jenkins `
       -p 8080:8080 `
       -v \\.\pipe\docker_engine:\\.\pipe\docker_engine `
       friism/jenkins
```

Note the special syntax, `\\.\pipe\docker_engine`, to access Docker's named pipe.

In this section, we have shown how to run Docker from within Docker by mounting the Docker socket into the respective container.

Next, we are going to revisit the topic of how to make your Docker build as fast as possible to reduce friction in the development cycle.

Optimizing your build process

The Docker build process can and should be optimized. This will remove a lot of friction in the software development life cycle.

Many Docker beginners make the following mistake when crafting their first Dockerfile:

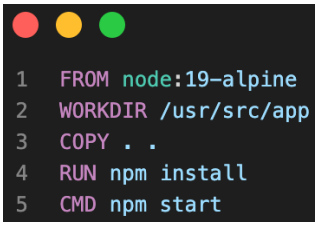


Figure 8.6 – Unoptimized Dockerfile for a Node.js application

Can you spot the weak point in this typical Dockerfile for a Node.js application? In *Chapter 4*, *Creating and Managing Container Images*, we learned that an image consists of a series of layers. Each (logical) line in a Dockerfile creates a layer, except the lines with the `CMD` and/or `ENTRYPOINT` keywords. We also learned that the Docker builder tries to do its best by caching layers and reusing them if they have not changed between subsequent builds. But the caching only uses cached layers that occur before the first changed layer. All subsequent layers need to be rebuilt. That said, the preceding structure of the Dockerfile invalidates – or as we often hear said – *busts* the image layer cache!

Why? Well, from experience, you certainly know that the `npm install` command can be a pretty expensive operation in a typical Node.js application with many external dependencies. The execution of this command can take from seconds to many minutes. That said, each time one of the source files changes, and we know that happens frequently during development, line 3 in the Dockerfile causes the corresponding image layer to change. Hence, the Docker builder cannot reuse this layer from the cache, nor can it reuse the subsequent layer created by `RUN npm install`. Any minor change in code causes a complete rerun of `npm install`. That can be avoided. The `package.json` file containing the list of external dependencies rarely changes. With all of that information, let's fix the Dockerfile:

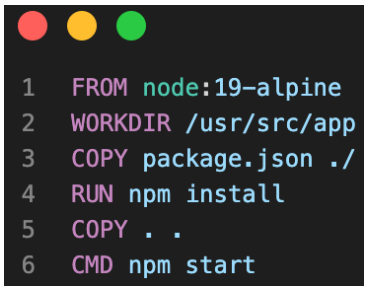


Figure 8.7 – Optimized Dockerfile for a Node.js application

This time, on line 3, we only copy the `package.json` file into the container, which rarely changes. Hence, the subsequent `npm install` command has to be executed equally rarely. The `COPY` command on line 5 is then a very fast operation and hence rebuilding an image after some code has changed only needs to rebuild this last layer. Build times reduce to merely a fraction of a second.

The very same principle applies to most languages or frameworks, such as Python, .NET, or Java. Avoid busting your image layer cache!

Scanning for vulnerabilities and secrets

What exactly are vulnerabilities, or to be more accurate, **Common Vulnerabilities and Exposures** (**CVE**)?

A database of information security problems that have been made publicly known is called **Common Vulnerabilities and Exposures**. A number uniquely identifies each vulnerability from the list of all other entries in the database. This list is continuously reviewed and updated by experts who include any new vulnerabilities or exposures as soon as they are found.

Now, we can scan the various layers of our Docker images using specialist software, such as Snyk, to find software libraries that are known to have such CVE. If we find that our image is flawed, we should and can repair the issue by switching to a more recent version of the flawed library. The image will then need to be rebuilt.

But our work is not yet done. Security experts frequently find new CVE, as was already mentioned previously. As a result, a software library that was previously secure may suddenly be vulnerable as a result of newly revealed CVE.

This means that we must ensure that all of our active Docker images are routinely inspected, notify our developers and security experts about the issue, and take other steps to ensure a speedy resolution of the issue.

There are a few ways to scan a Docker image for vulnerabilities and secrets:

- Use a vulnerability scanner such as Clair, Anchore, or Trivy. These tools can scan a Docker image and check it against a database of known vulnerabilities in order to identify any potential security risks.
- Use a tool such as Aquasec or Sysdig to scan the image for secrets. These tools can detect and alert on sensitive information such as private keys, passwords, and other sensitive data that may have been accidentally committed to the image.
- Use a combination of both tools, for example, Docker Bench for Security, which checks for dozens of common best practices around deploying Docker containers in production.
- Use a tool such as OpenSCAP, which can perform vulnerability scans, security configuration assessments, and compliance checks on a Docker image.

It's important to note that it's always good practice to keep your images updated and only use official and trusted images.

In the next section, we will investigate how we can discover vulnerabilities inside our Docker images.

Using Snyk to scan a Docker image

Snyk is a security platform that can be used to scan Docker images for vulnerabilities. Here is an example of how to use Snyk to scan a Docker image for vulnerabilities:

1. First, we have to install the Snyk CLI on our machine. We can do this by running the following command:

```
$ npm install -g snyk
```

2. Once Snyk is installed, we can authenticate with our Snyk account by running the following command and following the prompts:

```
$ snyk auth
```

3. Next, we can run the following command to scan a specific Docker image for vulnerabilities:

```
$ snyk test --docker <image-name>
```

   The preceding command will perform a vulnerability scan on the specified Docker image and print the results in the console. The results will show the number of vulnerabilities found, the severity of each vulnerability, and the package and version that is affected.

4. We can also use the `--file` flag to scan a Dockerfile instead of a built image:

```
$ snyk test --file=path/to/Dockerfile
```

5. Additionally, we can also use the `--org` flag to specify an organization, if we're a member of multiple organizations:

```
$ snyk test --docker <image-name> --org=my-org
```

6. Finally, we can use the `--fix` flag to automatically fix the vulnerabilities found by running the following command:

```
$ snyk protect --docker <image-name>
```

   Please note that this feature is only available for images that are built using a Dockerfile and it will update the Dockerfile with the new package versions, and you will need to rebuild the image to take advantage of the fix.

> **Note**
> The Snyk free plan is limited to a certain number of scans, and it does not include the *Protect* feature. You will have to upgrade to a paid plan to have access to this feature.

Using docker scan to scan a Docker image for vulnerabilities

In this section, we are once again going to use Snyk to scan a Docker image for vulnerabilities. Snyk should be included with your Docker Desktop installation:

1. Check by using this command:

```
$ docker scan --version
```

   The output should look similar to this:

```
Version:    v0.22.0
Git commit: af9ca12
Provider:   Snyk (1.1054.0)
```

2. Let's try to scan a sample `whoami` application from the author's Docker Hub account. First, make sure you have the `whoami` image in your local cache:

```
$ docker image pull gnschenker/whoami:1.0
```

3. Scan the image for vulnerabilities:

```
$ docker scan gnschenker/whoami:1.0
```

4. You will be asked the following:

```
Docker Scan relies upon access to Snyk, a third party provider,
do you consent to proceed using Snyk? (y/N)
```

   Please answer this with `y`.

   The result of the preceding scan looks like this on my computer:

```
⊗ →  ch08 git:(main) ✗ docker scan gnschenker/whoami:1.0

Testing gnschenker/whoami:1.0...

✗ Medium severity vulnerability found in openssl/libcrypto1.1
  Description: Inadequate Encryption Strength
  Info: https://security.snyk.io/vuln/SNYK-ALPINE315-OPENSSL-2941810
  Introduced through: openssl/libcrypto1.1@1.1.1n-r0, openssl/libssl1.1@1.1.1n-r0, apk-tools/apk-tools@2.12.7-r3,
 libretls/libretls@3.3.4-r3, ca-certificates/ca-certificates@20211220-r0, krb5-conf/krb5-conf@1.0-r2
  From: openssl/libcrypto1.1@1.1.1n-r0
  From: openssl/libssl1.1@1.1.1n-r0 > openssl/libcrypto1.1@1.1.1n-r0
  From: apk-tools/apk-tools@2.12.7-r3 > openssl/libcrypto1.1@1.1.1n-r0
  and 7 more...
  Fixed in: 1.1.1q-r0

✗ High severity vulnerability found in krb5/krb5-libs
  Description: Integer Overflow or Wraparound
  Info: https://security.snyk.io/vuln/SNYK-ALPINE315-KRB5-3136433
  Introduced through: krb5/krb5-libs@1.19.3-r0, krb5-conf/krb5-conf@1.0-r2
  From: krb5/krb5-libs@1.19.3-r0
  From: krb5-conf/krb5-conf@1.0-r2 > krb5/krb5-libs@1.19.3-r0
  Fixed in: 1.19.4-r0

✗ Critical severity vulnerability found in zlib/zlib
  Description: Out-of-bounds Write
  Info: https://security.snyk.io/vuln/SNYK-ALPINE315-ZLIB-2976173
  Introduced through: zlib/zlib@1.2.12-r0, apk-tools/apk-tools@2.12.7-r3
  From: zlib/zlib@1.2.12-r0
  From: apk-tools/apk-tools@2.12.7-r3 > zlib/zlib@1.2.12-r0
  Fixed in: 1.2.12-r2


Package manager:   apk
Project name:      docker-image|gnschenker/whoami
Docker image:      gnschenker/whoami:1.0
Platform:          linux/arm64

Tested 23 dependencies for known vulnerabilities, found 3 vulnerabilities.

For more free scans that keep your images secure, sign up to Snyk at https://dockr.ly/3ePqVcp
```

Figure 8.8 – Scanning the gnschenker/whoami:1.0 Docker image

As you can see, there were three vulnerabilities found in this version of the image: one of *medium*, one of *high*, and one of *critical* severity. It is clear that we should address critical vulnerabilities as soon as possible. Let's do this now:

1. First, we copy over the original `whoami` project including the Dockerfile we used to build this image. You can find the copy in your `~/The-Ultimate-Docker-Container-Book/sample-solutions/ch14` folder.
2. Open the Dockerfile and inspect it. We used version `6.0-alpine` for both the .NET SDK and the runtime. Let's see whether Microsoft has updated the vulnerabilities in this version already.
3. Navigate to your `.../ch08/whoami` folder.
4. Build a new version of the Docker image with this command:

   ```
   $ docker image build -t gnschenker/whoami:1.0.1 .
   ```

   Note, you may want to replace `gnschenker` with your own Docker account name.

5. Scan the new image:

```
$ docker scan gnschenker/whoami:1.0.1
```

This time, the output should look like this:

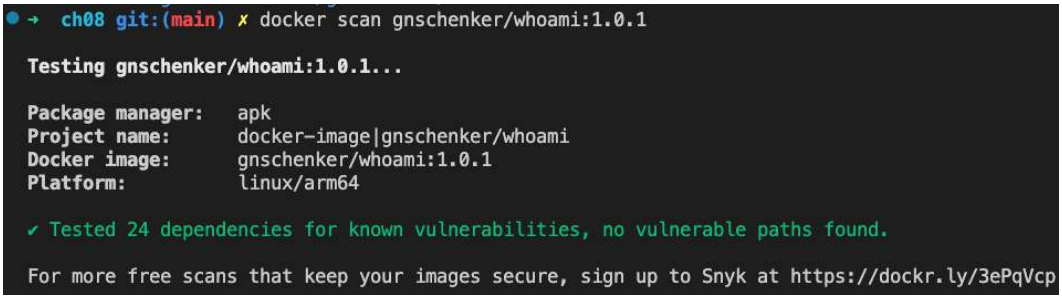


Figure 8.9 – Scanning the rebuilt whoami Docker image

As you can see, this time, the image is free from any vulnerabilities. We should now instruct our DevOps to use this new version of the image. We can use a rolling update in production and should be just fine, as the application itself did not change.

In the next section, we are going to learn how to run a complete development environment inside a container.

Running your development environment in a container

Imagine that you only have access to a workstation with Docker Desktop installed, but no possibility to add or change anything else on this workstation. Now you want to do some proofs of concept and code some sample applications using Java. Unfortunately, Java and SpringBoot are not installed on your computer. What can you do? What if you could run a whole development environment inside a container, including a code editor and debugger? What if, at the same time, you could still have your code files on your host machine?

Containers are awesome, and genius engineers have come up with solutions for exactly this kind of problem.

> **Note**
>
> Microsoft and the community are continuously updating VS Code and the plugins. Thus your version of VS Code may be newer than the one used during the writing of this book. As such, expect a slightly different experience. Refer to the official documentation for more details on how to work with Dev containers: `https://code.visualstudio.com/docs/devcontainers/containers`.

We will be using Visual Studio Code, our favorite code editor, to show how to run a complete Java development environment inside a container:

1. But first, we need to install the necessary VS Code extension. Open VS Code and install the extension called **Remote Development**:

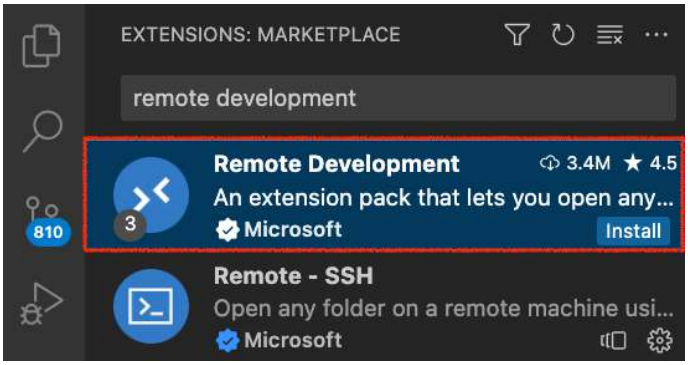


Figure 8.10 – Adding the Remote Development extension to VS Code

2. Then, click the green quick actions status bar item in the lower-left of the Visual Studio Code window. In the popup, select **Remote-Containers | Open Folder in Container...**:

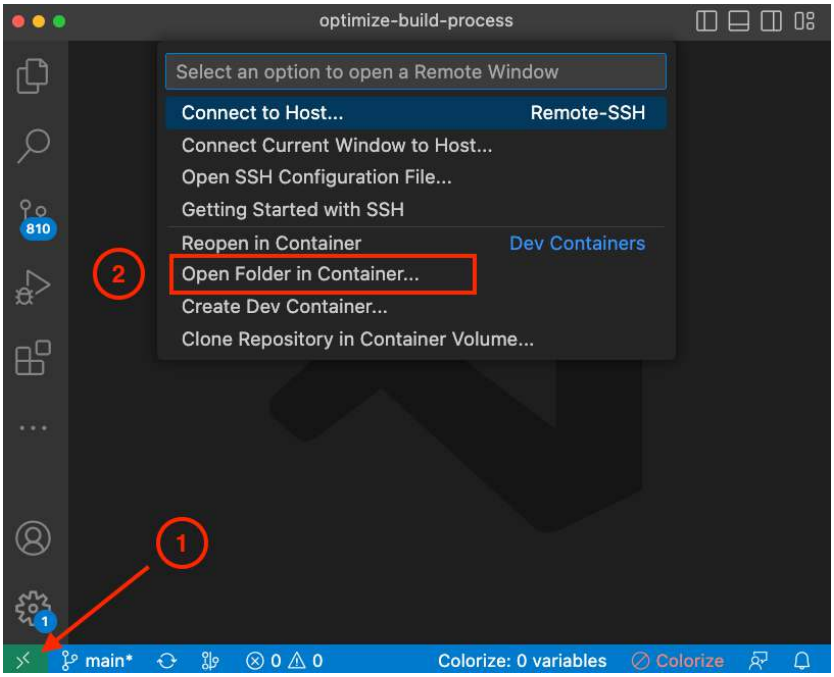


Figure 8.11 – Open Folder in Container

3. Select the project folder you want to work with in the container. In our case, we selected the `~/The-Ultimate-Docker-Container-Book/ch08/library` folder.
4. A popup will appear asking you to define how you want to create the development container. From the list, select **From 'Dockerfile'**:

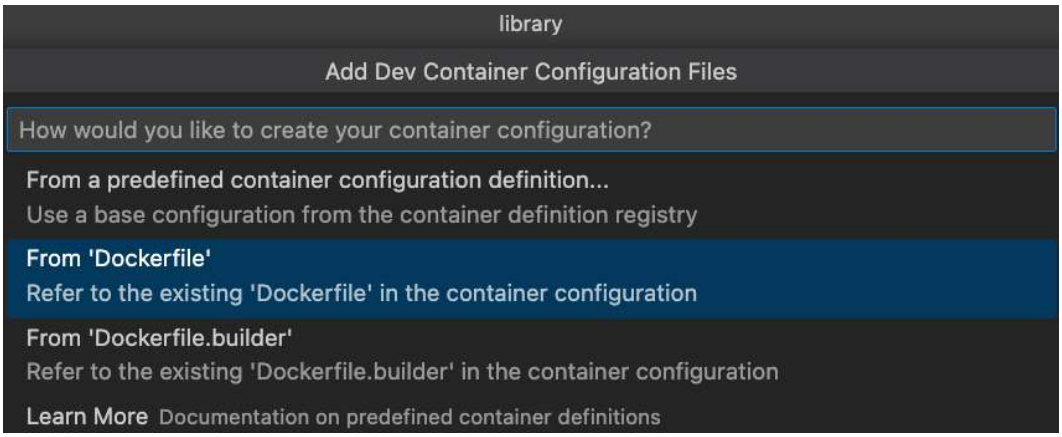


Figure 8.12 – Selecting the method to create the development container

5. When asked to add additional features to install, just click **OK** to continue. At this time, we do not need anything special.

   VS Code will now start preparing the environment, which, the very first time, can take a couple of minutes or so.
6. Once the environment is ready, you should notice that in the lower-left corner, the prompt has changed to the following:

   ```
   Dev Container: Existing Dockerfile @ <folder-path>
   ```

   This indicates that VS Code has indeed run a container based on the Dockerfile found in the library folder and is allowing you to work within it.
7. You will be asked to install the extension pack for Java since VS Code has recognized that this is a Java project. Click **Install**. Note, the VS Code server is running inside the `dev` container and only the UI is still running on your laptop. Thus, the extension pack will be installed for the engine inside the container. You will notice this when you open the **EXTENSIONS** panel and find a list of remote extensions under **DEV CONTAINER**. In our case, by installing the Java extensions pack, we now have the following eight remote extensions installed:

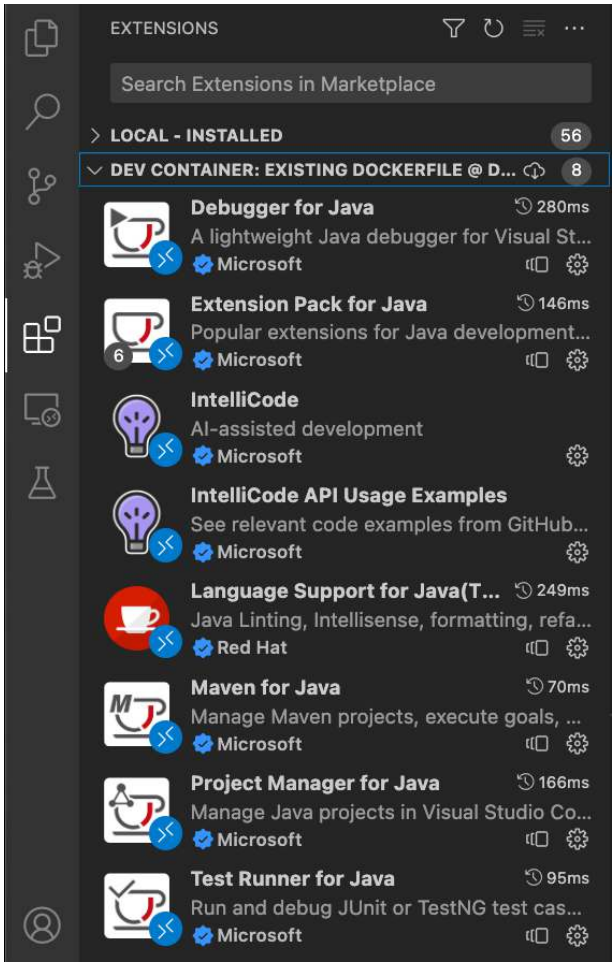


Figure 8.13 – Remote extensions installed on the dev container

8. Open a Terminal inside VS Code with *Shift* + *Ctrl* + ‘ and notice the prompt revealing that the terminal session is inside the `dev` container and that we are *not* running directly on our Docker host:

```
root@c96b82891be7:/workspaces/.../ch08/library#
```

Note that for readability, we have shortened the preceding prompt.

9. Now, try to run the Java application by locating the `main` method in the `LibraryApplication` class and clicking the **Run** link just above the method. The application should start as normal, but notice that our context is inside the dev container and not directly on our working machine.

Alternatively, we could have started the application from the command line with this command:

```
$ ./mvnw spring-boot:run
```

10. Now, add a file called `DefaultController.java` to the `controllers` folder and give it this content:

```
package com.example.library.controllers;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class DefaultController {
  @GetMapping("/")
  public String index() {
    return "Library component";
  }
}

```

Figure 8.14 – Adding a default controller while working inside the dev container

11. Restart the application and open a browser as `http://localhost:8080`. The message `Library component` should be displayed as expected.
12. When done experimenting, click on the green area in the lower-left corner of VS Code and select **Open folder locally** from the pop-up menu to quit the dev container and open the project locally.
13. Observe that a new folder, `.devcontainer`, has been added to the project containing a `devcontainer.json` file. This file contains the configuration needed to run a dev container from this project. Please read the documentation of VS Code to familiarize yourself with the possibilities this file offers to you.

These have been a few tips and tricks for pros that are useful in the day-to-day usage of containers. There are many more. Google them. It is worth it.

Summary

In this chapter, we presented miscellaneous tips, tricks, and concepts that are useful when containerizing complex distributed applications or when using Docker to automate sophisticated tasks. We also learned how to leverage containers to run a whole development environment inside of them.

In the next chapter, we will introduce the concept of a distributed application architecture and discuss the various patterns and best practices that are required to run a distributed application successfully.

Questions

Here are a few questions you should try to answer to assess your progress:

1. Name the reasons why you would want to run a complete development environment inside a container.
2. Why should you avoid running applications inside a container as `root`?
3. Why would you ever bind-mount the Docker socket into a container?
4. When pruning your Docker resources to make space, why do you need to handle volumes with special care?
5. Why would you want to run certain admin tasks inside a Docker container and not natively on the host machine?

Answers

Here are sample answers for the questions in this chapter:

1. You could be working on a workstation with limited resources or capabilities, or your workstation could be locked down by your company so that you are not allowed to install any software that is not officially approved. Sometimes, you might need to do proofs of concept or experiments using languages or frameworks that are not yet approved by your company (but might be in the future if the proof of concept is successful).
2. Bind-mounting a Docker socket into a container is the recommended method when a containerized application needs to automate some container-related tasks. This can be an application such as an automation server (such as Jenkins) that you are using to build, test, and deploy Docker images.
3. Most business applications do not need `root`-level authorizations to do their job. From a security perspective, it is therefore strongly recommended to run such applications with the least necessary access rights to their job. Any unnecessary elevated privileges could possibly be exploited by hackers in a malicious attack. By running the application as a non-`root` user, you make it more difficult for potential hackers to compromise your system.
4. Volumes contain data and the lifespan of data most often needs to go far beyond the life cycle of a container, or an application, for that matter. Data is often mission-critical and needs to be stored safely for days, months, or even years. When you delete a volume, you irreversibly delete the data associated with it. Hence, make sure you know what you're doing when deleting a volume.

5. There are several reasons why you might want to run certain admin tasks inside a Docker container, rather than natively on the host machine:

   - **Isolation**: Containers provide a level of isolation from the host machine, so running admin tasks inside a container can help to prevent conflicts with other processes or dependencies on the host machine.
   - **Portability**: Containers are designed to be lightweight and portable, which allows for easy deployment of admin tasks across different environments. This can be particularly useful for tasks that need to be run in multiple environments or on multiple machines.
   - **Consistency**: Containers provide a consistent environment for running admin tasks, regardless of the underlying host machine's configuration. This can be useful for ensuring that tasks are run in a predictable and repeatable manner, which can help to minimize errors and improve efficiency.
   - **Versioning**: Containers allow for easy versioning of admin tasks, which allows for rollbacks and roll forward of the tasks. This can be useful for testing, troubleshooting, and production environments.
   - **Security**: Running admin tasks inside a container can help to improve security by isolating the task from the host machine, and by making it easier to limit the permissions and access that the task has.
   - **Scalability**: Containers can be easily scaled up and down, allowing you to increase or decrease the resources that the admin task needs.

   Please note that this is not a comprehensive list and different use cases may require different approaches. It's important to weigh the pros and cons of running admin tasks inside a container versus natively on the host machine and to choose the approach that best fits your particular use case.

Part 3: Orchestration Fundamentals

By the end of *Part 3*, you will be familiar with the concepts of a Dockerized distributed application and container orchestrators, and be able to use Docker Swarm to deploy and run your applications.

- *Chapter 9*, *Learning about Distributed Application Architecture*
- *Chapter 10*, *Using Single-Host Networking*
- *Chapter 11*, *Managing Containers with Docker Compose*
- *Chapter 12*, *Shipping Logs and Monitoring Containers*
- *Chapter 13*, *Introducing Container Orchestration*
- *Chapter 14*, *Introducing Introducing Docker Swarm*
- *Chapter 15*, *Deploying and Running a Distributed Application on Docker Swarm*

9

Learning about Distributed Application Architecture

This chapter introduces the concept of distributed application architecture and discusses the various patterns and best practices that are required to run a distributed application successfully. It will also discuss the additional requirements that need to be fulfilled to run such an application in production. You might be wondering, what does this have to do with Docker containers? And you are right to ask. At first glance, these are not related to each other. But as you will soon see, when introducing containers that host an application or application service, your application will quickly consist of several containers that will be running on different nodes of a cluster of computers or VMs; and voilà – you are dealing with a distributed application. We thought that it makes sense to provide you with a sense of the complexity that distributed applications introduce and help you avoid the most common pitfalls.

Here is the list of topics we are going to discuss:

- What is a distributed application architecture?
- Patterns and best practices
- Running in production

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

- Draft a high-level architecture diagram of a distributed application while pointing out key design patterns
- Identify the possible pitfalls of a poorly designed distributed application
- Name commonly used patterns for dealing with the problems of a distributed system
- Name at least four patterns that need to be implemented for a production-ready distributed application

Let's get started!

What is a distributed application architecture?

In this section, we are going to explain what we mean when we talk about distributed application architecture. First, we need to make sure that all the words or acronyms we use have a meaning and that we are all talking in the same language.

Defining the terminology

In this and subsequent chapters, we will talk a lot about concepts that might not be familiar to everyone. To make sure we are all talking the same language, let's briefly introduce and describe the most important of these concepts or words:

Keyword	Description
VM	A virtual machine (VM) is a software simulation of a physical computer that runs on a host computer. It provides a separate operating system and resources, allowing multiple operating systems to run on a single physical machine.
Cluster	A cluster is a group of connected servers that work together as a single system to provide high availability, scalability, and increased performance for applications. The nodes in a cluster are connected through a network and share resources to provide a unified, highly available solution.
Node	A cluster node is a single server within a cluster computing system. It provides computing resources and works together with other nodes to perform tasks as a unified system, providing high availability and scalability for applications.
Network	A network is a group of interconnected devices that can exchange data and information. Networks can be used to connect computers, servers, mobile devices, and other types of devices and allow them to communicate with each other and share resources, such as printers and storage. More specifically in our case, these are physical and software-defined communication paths between individual nodes of a cluster and programs running on those nodes.
Port	A port is a communication endpoint in a network-attached device, such as a computer or server. It allows the device to receive and send data to other devices on the network through a specific network protocol, such as TCP or UDP. Each port has a unique number that is used to identify it, and different services and applications use specific ports to communicate.
Service	Unfortunately, this is a very overloaded term and its real meaning depends on the context that it is used in. If we use the term service in the context of an application, such as an application service, then it usually means that this is a piece of software that implements a limited set of functionalities that are then used by other parts of the application. As we progress through this book, other types of services that have slightly different definitions will be discussed.

Naively said, a distributed application architecture is the opposite of a monolithic application architecture, but it is not unreasonable to look at this monolithic architecture first. Traditionally, most business applications are written in such a way that the result can be seen as a single, tightly coupled program that runs on a named server somewhere in a data center. All its code is compiled into a single binary, or a few very tightly coupled binaries that need to be co-located when running the application. The fact that the server – or more generally – the host, that the application is running on has a well-defined name or static IP address is also important in this context. Let's look at the following diagram, which illustrates this type of application architecture a bit more precisely:

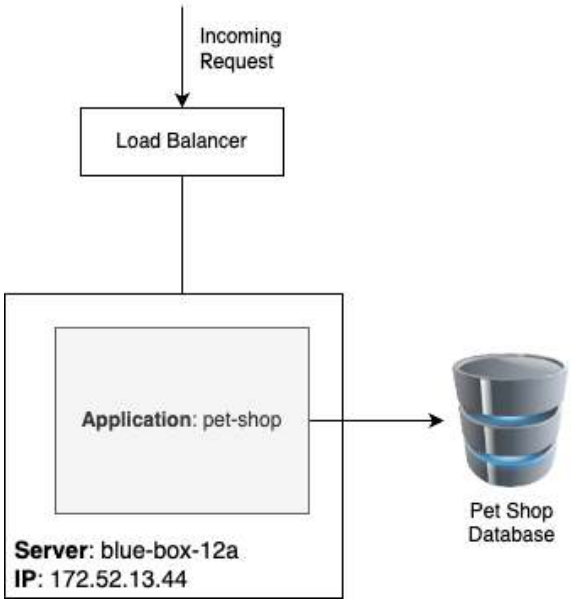


Figure 9.1 – Monolithic application architecture

In the preceding diagram, we can see a server named `blue-box-12a` with an IP address of `172.52.13.44` running an application called `pet-shop`, which is a monolith consisting of a main module and a few tightly coupled libraries.

Now, let's look at the following diagram:

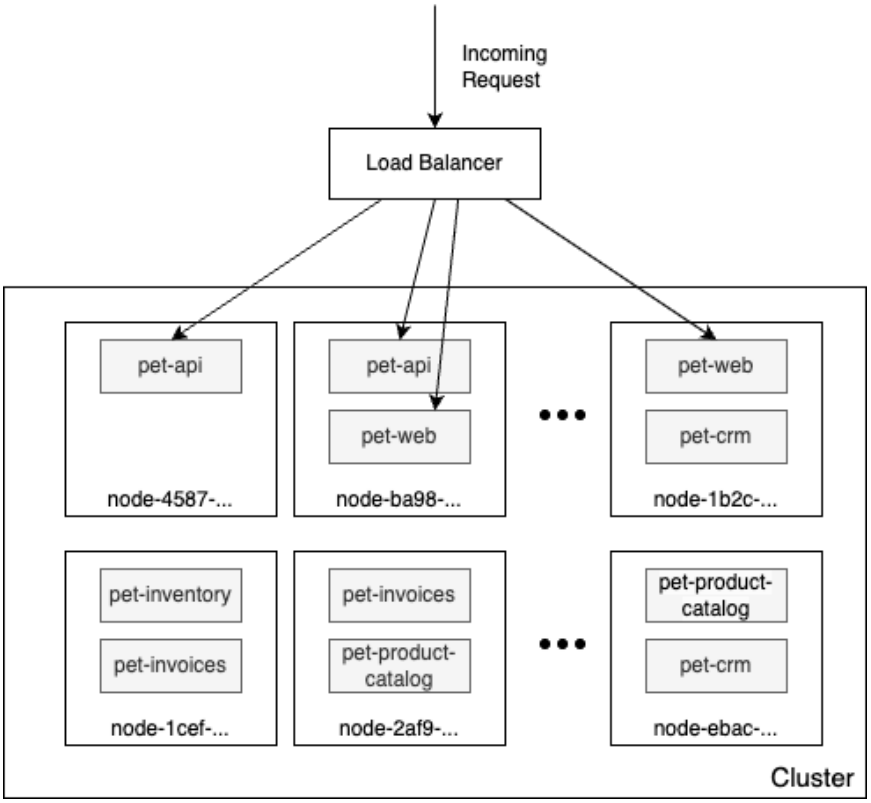


Figure 9.2 – Distributed application architecture

Here, all of a sudden, we do not have just a single named server anymore; instead, we have a lot of them, and they do not have human-friendly names, but rather some unique IDs that can be something such as a **Universal Unique Identifier** (**UUID**). Now, the pet shop application does not consist of a single monolithic block anymore, but rather a plethora of interacting, yet loosely coupled, services such as `pet-api`, `pet-web`, and `pet-inventory`. Furthermore, each service runs in multiple instances in this cluster of servers or hosts.

You might be wondering why we are discussing this in a book about Docker containers, and you are right to ask. While all the topics we're going to investigate apply equally to a world where containers do not (yet) exist, it is important to realize that containers and container orchestration engines help address all these problems in a much more efficient and straightforward way. Most of the problems that used to be very hard to solve in a distributed application architecture become quite simple in a containerized world.

Patterns and best practices

A distributed application architecture has many compelling benefits, but it also has one very significant drawback compared to a monolithic application architecture – the former is way more complex. To tame this complexity, the industry has come up with some important best practices and patterns. In the following sections, we are going to investigate some of the most important ones in more detail.

Loosely coupled components

The best way to address a complex subject has always been to divide it into smaller subproblems that are more manageable. As an example, it would be insanely complex to build a house in a single step. It is much easier to build a house from simple parts that are then combined into the final result.

The same also applies to software development. It is much easier to develop a very complex application if we divide this application into smaller components that interoperate and make up the overall application. Now, it is much easier to develop these components individually if they are loosely coupled with each other. What this means is that component A makes no assumptions about the inner workings of, say, components B and C, and is only interested in how it can communicate with those two components across a well-defined interface.

If each component has a well-defined and simple public interface through which communication with the other components in the system and the outside world happens, then this enables us to develop each component individually, without implicit dependencies on other components. During the development process, other components in the system can easily be replaced by stubs or mocks to allow us to test our components.

Stateful versus stateless

Every meaningful business application creates, modifies, or uses data. In IT, a synonym for data is **state**. An application service that creates or modifies persistent data is called a **stateful component**. Typical stateful components are database services or services that create files. On the other hand, application components that do not create or modify persistent data are called **stateless components**.

In a distributed application architecture, stateless components are much simpler to handle than stateful components. Stateless components can easily be scaled up and down. Furthermore, they can be quickly and painlessly torn down and restarted on a completely different node of the cluster – all because they have no persistent data associated with them.

Given this, it is helpful to design a system in a way that most of the application services are stateless. It is best to push all the stateful components to the boundaries of the application and limit how many are used. Managing stateful components is hard.

Service discovery

As we build applications that consist of many individual components or services that communicate with each other, we need a mechanism that allows the individual components to find each other in the cluster. Finding each other usually means that you need to know on which node the target component is running, and on which port it is listening for communication. Most often, nodes are identified by an **IP address** and a **port**, which is just a number in a well-defined range.

Technically, we could tell **Service A**, which wants to communicate with a target, **Service B**, what the IP address and port of the target are. This could happen, for example, through an entry in a configuration file:

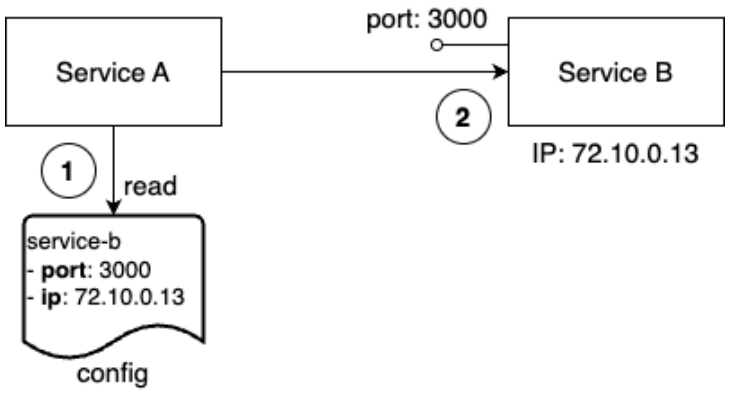


Figure 9.3 – Components are hardwired

While this might work very well in the context of a monolithic application that runs on one or only a few well-known and curated servers, it falls apart in a distributed application architecture. First of all, in this scenario, we have many components, and keeping track of them manually becomes a nightmare. This is not scalable. Furthermore, typically, Service A should or will never know on which node of the cluster the other components run. Their location may not even be stable as component B could be moved from node *X* to another node, *Y*, due to various reasons external to the application. Thus, we need another way in which Service A can locate Service B, or any other service, for that matter. Commonly, an external authority that is aware of the topology of the system at any given time is used.

This external authority or service knows all the nodes and their IP addresses that currently pertain to the cluster; it knows about all the services that are running and where they are running. Often, this kind of service is called a **DNS service**, where DNS stands for **Domain Name System**. As we will see, Docker has a DNS service implemented as part of its underlying engine. Kubernetes – the number one container orchestration system, which we'll discuss in *Chapter 13*, *Introducing Container Orchestration* – also uses a DNS service to facilitate communication between components running in a cluster:

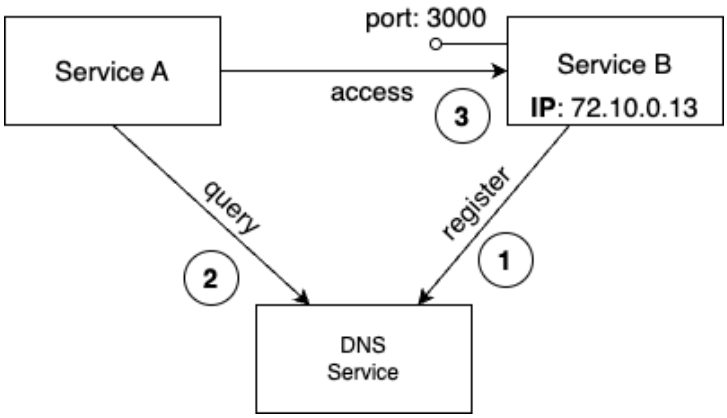


Figure 9.4 – Components consulting an external locator service

In the preceding diagram, we can see how Service A wants to communicate with Service B, but it can't do this directly. First, it has to query the external authority, a registry service (here, this is called **DNS Service**), about the whereabouts of Service B. The registry service will answer with the requested information and hand out the IP address and port number that Service A can use to reach Service B. Service A then uses this information and establishes communication with Service B. Of course, this is a naive picture of what's happening at a low level, but it is a good picture to help us understand the architectural pattern of service discovery.

Routing

Routing is the mechanism of sending packets of data from a source component to a target component. Routing is categorized into different types. The so-called OSI model (see the reference to this in the *Further reading* section at the end of this chapter for more information) is used to distinguish between different types of routing. In the context of containers and container orchestration, routing at layers 2, 3, 4, and 7 are relevant. We will look at routing in more detail in subsequent chapters. For now, let's just say that layer 2 routing is the most low-level type of routing, which connects a MAC address to another MAC address, while layer 7 routing, which is also called application-level routing, is the most high-level one. The latter is, for example, used to route requests that have a target identifier – that is, a URL such as `https://acme.com/pets` – to the appropriate target component in our system.

Load balancing

Load balancing is used whenever **Service A** needs to communicate with **Service B**, such as in a request-response pattern, but the latter is running in more than one instance, as shown in the following diagram:

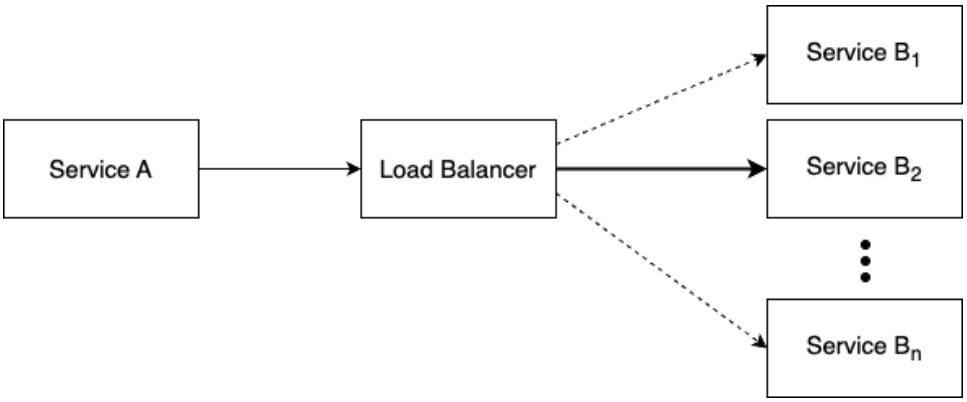


Figure 9.5 – The request of Service A is being load balanced to Service B

If we have multiple instances of a service such as Service B running in our system, we want to make sure that every one of those instances gets an equal amount of workload assigned to it. This task is a generic one, which means that we don't want the caller to have to do the load balancing but, rather, an external service that intercepts the call and takes over the role of deciding which of the target service instances to forward the call to. This external service is called a load balancer. Load balancers can use different algorithms to decide how to distribute incoming calls to target service instances. The most common algorithm that's used is called round-robin. This algorithm assigns requests repetitively, starting with instance 1, then 2, until instance *n*. After the last instance has been served, the load balancer starts over with instance number 1.

In the preceding example, a load balancer also facilitates high availability since a request from Service A will be forwarded to a healthy instance of Service B. The load balancer also takes the role of periodically checking the health of each instance of B.

Defensive programming

When developing a service for a distributed application, it is important to remember that this service is not going to be standalone and that it's dependent on other application services or even on external services provided by third parties, such as credit card validation services or stock information services, to just name two. All these other services are external to the service we are developing. We have no control over their correctness or their availability at any given time. Thus, when coding, we always need to assume the worst and hope for the best. Assuming the worst means that we have to deal with potential failures explicitly.

Retries

When there is a possibility that an external service might be temporarily unavailable or not responsive enough, then the following procedure can be used. When the call to the other service fails or times out, the calling code should be structured in such a way that the same call is repeated after a short wait time. If the call fails again, the wait should be a bit longer before the next trial. The calls should be repeated up to a maximum number of times, each time increasing the wait time. After that, the service should give up and provide a degraded service, which could mean returning some stale cached data or no data at all, depending on the situation.

Logging

Important operations that are performed on a service should always be logged. Logging information needs to be categorized to be of any real value. A common list of categories includes *debug*, *info*, *warning*, *error*, and *fatal*. Logging information should be collected by a central log aggregation service and not stored on an individual node of the cluster. Aggregated logs are easy to parse and filter for relevant information. This information is essential to quickly pinpoint the root cause of a failure or unexpected behavior in a distributed system consisting of many moving parts, running in production.

Error handling

As we mentioned earlier, each application service in a distributed application is dependent on other services. As developers, we should always expect the worst and have appropriate error handling in place. One of the most important best practices is to fail fast. Code the service in such a way that unrecoverable errors are discovered as early as possible and, if such an error is detected, have the service fail immediately. But don't forget to log meaningful information to `STDERR` or `STDOUT`, which can be used by developers or system operators later to track malfunctions in the system. Also, return a helpful error to the caller, indicating as precisely as possible why the call failed.

One sample of fail fast is always checking the input values provided by the caller. Are the values in the expected ranges and complete? If not, then do not try to continue processing; instead, immediately abort the operation.

Redundancy

A mission-critical system has to be available at all times, around the clock, 365 days a year. Downtime is not acceptable since it might result in a huge loss of opportunities or reputation for the company. In a highly distributed application, the likelihood of a failure of at least one of the many involved components is non-neglectable. We can say that the question is not whether a component will fail, but rather when a failure will occur.

To avoid downtime when one of the many components in the system fails, each part of the system needs to be redundant. This includes the application components, as well as all infrastructure parts. What that means is that if we have a payment service as part of our application, then we need to run this service redundantly. The easiest way to do that is to run multiple instances of this very service on different nodes of our cluster. The same applies, say, to an edge router or a load balancer. We cannot afford for these to ever go down. Thus, the router or load balancer must be redundant.

Health checks

We have mentioned various times that in a distributed application architecture, with its many parts, the failure of an individual component is highly likely and that it is only a matter of time until it happens. For that reason, we must run every single component of the system redundantly. Load balancers then distribute the traffic across the individual instances of a service.

But now, there is another problem. How does the load balancer or router know whether a certain service instance is available? It could have crashed, or it could be unresponsive. To solve this problem, we can use so-called **health checks**. The load balancer, or some other system service on behalf of it, periodically polls all the service instances and checks their health. The questions are basically, *Are you still there? Are you healthy?* The answer to each question is either *Yes* or *No*, or the health check times out if the instance is not responsive anymore.

If the component answers with *No* or a timeout occurs, then the system kills the corresponding instance and spins up a new instance in its place. If all this happens in a fully automated way, then we say that we have an auto-healing system in place. Instead of the load balancer periodically polling the status of the components, responsibility can also be turned around. The components could be required to periodically send live signals to the load balancer. If a component fails to send live signals over a predefined, extended period, it is assumed to be unhealthy or dead.

There are situations where either of the described ways is more appropriate.

Circuit breaker pattern

A circuit breaker is a mechanism that is used to avoid a distributed application going down due to the cascading failure of many essential components. Circuit breakers help us avoid one failing component tearing down other dependent services in a domino effect. Like circuit breakers in an electrical system, which protect a house from burning down due to the failure of a malfunctioning plugged-in appliance by interrupting the power line, circuit breakers in a distributed application interrupt the connection from **Service A** to **Service B** if the latter is not responding or is malfunctioning.

This can be achieved by wrapping a protected service call in a circuit breaker object. This object monitors for failures. Once the number of failures reaches a certain threshold, the circuit breaker trips. All subsequent calls to the circuit breaker will return with an error, without the protected call being made at all:

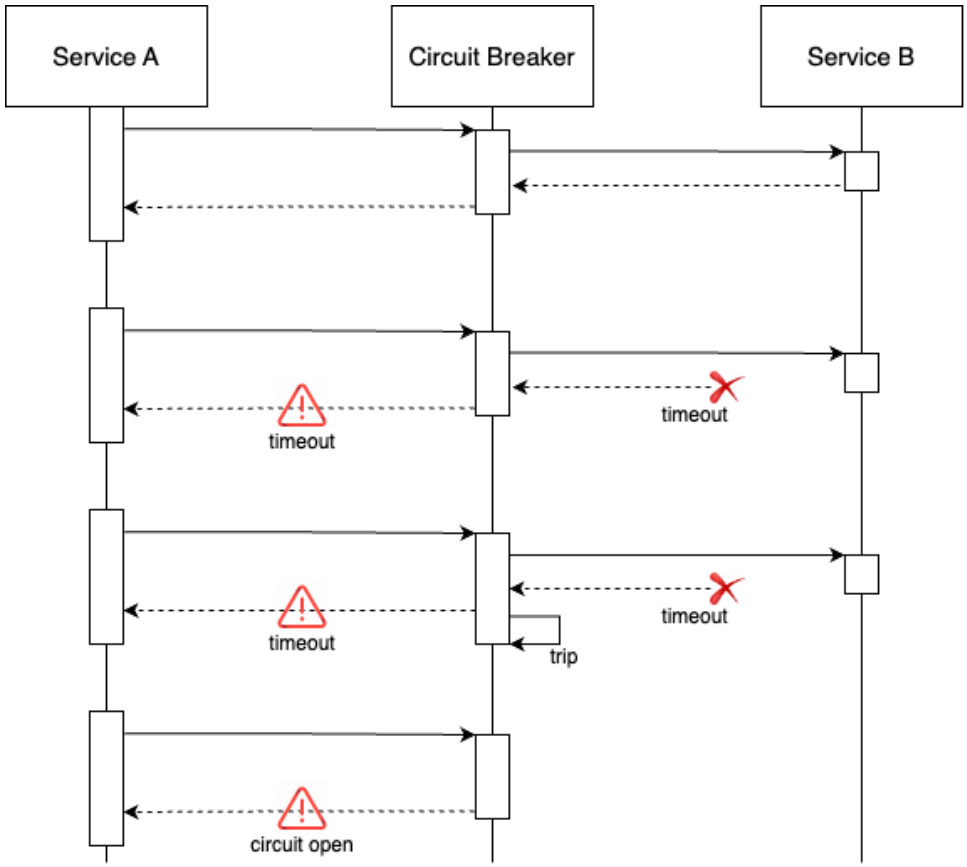


Figure 9.6 – Circuit breaker pattern

In the preceding diagram, we have a circuit breaker that tips over after the second timeout is received when calling **Service B**.

Rate limiter

In the context of a circuit breaker, a rate limiter is a technique that's used to control the rate at which requests are processed by a system or service. By limiting the number of requests allowed within a specific time window, rate limiters help prevent overloading and ensure the stability and availability of the service. This mechanism proves useful in mitigating the impact of sudden traffic spikes, protecting backend systems from being overwhelmed, and avoiding cascading failures throughout a distributed system. By integrating rate limiting with circuit breakers, systems can effectively maintain optimal performance and gracefully handle unexpected surges in demand.

Bulkhead

In addition to that, and still in the context of a circuit breaker, a bulkhead is a resilience pattern that's used to isolate components or resources within a system, ensuring that a failure in one area does not cause a cascading effect on the entire system. By partitioning resources and segregating operations into separate, independent units, bulkheads help prevent a single point of failure from bringing down the entire service. This mechanism is useful in maintaining system stability, improving fault tolerance, and ensuring that critical operations can continue functioning, even in the event of localized failures. When combined with circuit breakers, bulkheads contribute to a more robust and resilient system, capable of handling failures and maintaining overall system performance.

Running in production

To successfully run a distributed application in production, we need to consider a few more aspects beyond the best practices and patterns that were presented in the preceding sections. One specific area that comes to mind is introspection and monitoring. Let's go through the most important aspects in detail.

Logging

Once a distributed application is in production, it is not possible to live debug it. But how can we then find out what the root cause of the application malfunctioning is? The solution to this problem is that the application produces abundant and meaningful logging information while running. We briefly discussed this topic in an earlier section. But due to its importance, it is worth reiterating. Developers need to instrument their application services in such a way that they output helpful information, such as when an error occurs or a potentially unexpected or unwanted situation is encountered. Often, this information is output to `STDOUT` and `STDERR`, where it is then collected by system daemons that write the information to local files or forward it to a central log aggregation service.

If there is sufficient information in the logs, developers can use those logs to track down the root cause of the errors in the system.

In a distributed application architecture, with its many components, logging is even more important than in a monolithic application. The paths of execution of a single request through all the components of the application can be very complex. Also, remember that the components are distributed across a cluster of nodes. Thus, it makes sense to log everything of importance and add things to each log entry, such as the exact time when it happened, the component in which it happened, and the node on which the component ran, to name just a few. Furthermore, the logging information should be aggregated in a central location so that it is readily available for developers and system operators to analyze.

Tracing

Tracing is used to find out how an individual request is funneled through a distributed application and how much time is spent overall on the request and in every individual component. This information, if collected, can be used as one of the sources for dashboards that show the behavior and health of the system.

Monitoring

Operation engineers like to have dashboards showing live key metrics of the system, which show them the overall health of the application at a glance. These metrics can be nonfunctional metrics, such as memory and CPU usage, the number of crashes of a system or application component, and the health of a node, as well as functional and, hence, application-specific metrics, such as the number of checkouts in an ordering system or the number of items out of stock in an inventory service.

Most often, the base data that's used to aggregate the numbers that are used for a dashboard is extracted from logging information. This can either be system logs, which are mostly used for non-functional metrics, or application-level logs, for functional metrics.

Application updates

One of the competitive advantages for a company is to be able to react promptly to changing market situations. Part of this is being able to quickly adjust an application to fulfill new and changed needs or to add new functionality. The faster we can update our applications, the better. Many companies these days roll out new or changed features multiple times per day.

Since application updates are so frequent, these updates have to be non-disruptive. We cannot allow the system to go down for maintenance when upgrading. It all has to happen seamlessly and transparently.

Rolling updates

One way of updating an application or an application service is to use rolling updates. The assumption here is that the particular piece of software that has to be updated runs in multiple instances. Only then can we use this type of update.

What happens is that the system stops one instance of the current service and replaces it with an instance of the new service. As soon as the new instance is ready, it will be served traffic. Usually, the new instance is monitored for some time to see whether it works as expected; if it does, the next instance of the current service is taken down and replaced with a new instance. This pattern is repeated until all the service instances have been replaced.

Since there are always a few instances running at any given time, current or new, the application is operational all the time. No downtime is needed.

Blue-green deployments

In blue-green deployments, the current version of the application service, called **blue**, handles all the application traffic. We then install the new version of the application service, called **green**, on the production system. This new service is not wired with the rest of the application yet.

Once the green service has been installed, we can execute **smoke tests** against this new service. If those succeed, the router can be configured to funnel all traffic that previously went to blue to the new service, green. The behavior of the green service is then observed closely and, if all success criteria are met, the blue service can be decommissioned. But if, for some reason, the green service shows some unexpected or unwanted behavior, the router can be reconfigured to return all traffic to the blue service. The green service can then be removed and fixed, and a new blue-green deployment can be executed with the corrected version:

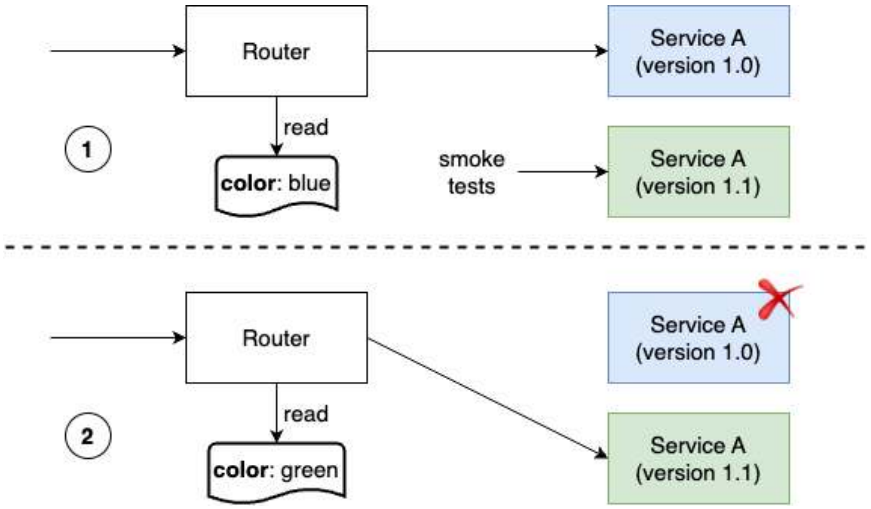


Figure 9.7 – Blue-green deployment

Next, let's look at canary releases.

Canary releases

Canary releases are releases where we have the current version of the application service and the new version installed on the system in parallel. As such, they resemble blue-green deployments. At first, all traffic is still routed through the current version. We then configure a router so that it funnels a small percentage, say 1%, of the overall traffic to the new version of the application service. Subsequently, the behavior of the new service is monitored closely to find out whether it works as expected. If all the criteria for success are met, then the router is configured to funnel more traffic, say 5% this time, through the new service. Again, the behavior of the new service is closely monitored and, if it is successful, more and more traffic is routed to it until we reach 100%. Once all the traffic has been routed to the new service and it has been stable for some time, the old version of the service can be decommissioned.

Why do we call this a canary release? It is named after coal miners who would use canary birds as an early warning system in mines. Canaries are particularly sensitive to toxic gas and if such a bird died, the miners knew they had to abandon the mine immediately.

Irreversible data changes

If part of our update process is to execute an irreversible change in our state, such as an irreversible schema change in a backing relational database, then we need to address this with special care. It is possible to execute such changes without downtime if we use the right approach. It is important to recognize that, in such a situation, we cannot deploy the code changes that require the new data structure in the data store at the same time as the changes to the data. Rather, the whole update has to be separated into three distinct steps. In the first step, we roll out a backward-compatible schema and data change. If this is successful, then we roll out the new code in the second step. Again, if that is successful, we clean up the schema in the third step and remove the backward compatibility:

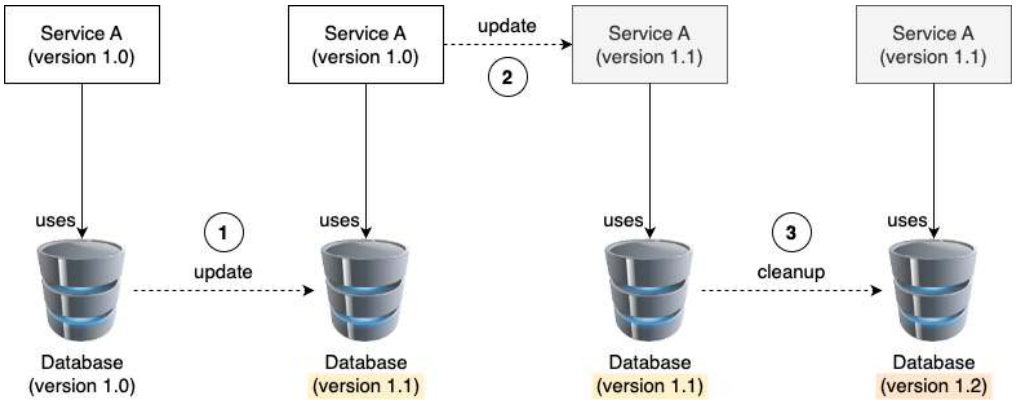


Figure 9.8 – Rolling out an irreversible data or schema change

The preceding diagram shows how the data and its structure are updated, how the application code is updated, and how the data and data structure are cleaned up.

Changing the data structure at scale

Over time, an application may produce an enormous amount of data. Changing the data structure at scale refers to the process of altering the format, organization, or layout of large amounts of data stored in a database or other type of data storage system. This can involve adding, removing, or modifying fields, tables, or other elements within the data structure. The goal is to optimize the data for a specific use case or business requirement while preserving the data's accuracy and integrity. This process typically involves analyzing the existing data structure, planning and testing the changes, and then executing the update in a controlled manner. In large-scale data structure changes, it is important to have a well-defined strategy, a robust testing and validation process, and adequate resources, including technical expertise and backup systems, to minimize the risk of data loss or corruption during the migration process.

In a dynamic data migration scenario, data is constantly updated in real time as it is being used, making the migration process more complex and challenging. This type of migration requires a more sophisticated approach to ensure data consistency and integrity throughout the migration process. The solution should be able to keep track of changes made to the data in the source system and replicate them in the target system while minimizing downtime and data loss. This may involve using specialized tools, such as data replication or mirroring software, or employing a multi-step process that includes data synchronization and reconciliation. Additionally, it is essential to have robust testing and validation procedures in place, as well as a clear rollback plan, to minimize the risk of data loss or corruption during the migration process.

Rollback and roll forward

If we have frequent updates for our application services that run in production, sooner or later, there will be a problem with one of those updates. Maybe a developer, while fixing a bug, introduced a new one, which was not caught by all the automated, and maybe manual, tests, so the application is misbehaving. In this case, we must roll back the service to the previous good version. In this regard, a rollback is a recovery from a disaster.

Again, in a distributed application architecture, it is not a question of whether a rollback will ever be needed, but rather when a rollback will have to occur. Thus, we need to be sure that we can always roll back to a previous version of any service that makes up our application. Rollbacks cannot be an afterthought; they have to be a tested and proven part of our deployment process.

If we are using blue-green deployments to update our services, then rollbacks should be fairly simple. All we need to do is switch the router from the new green version of the service back to the previous blue version.

If we adhere to continuous delivery and the main branch of our code is always in a deployable state, then we can also consider rolling forward instead of rolling back. Often, it is faster to fix a production issue and roll out the fix immediately instead of trying to roll back our system to a previous state. The technique of rolling forward is of particular interest if the previous change introduced some backward incompatibility.

Summary

In this chapter, we learned what a distributed application architecture is and what patterns and best practices are helpful or needed to successfully run a distributed application. We also discussed what is needed to run such an application in production.

In the next chapter, we will dive into networking limited to a single host. We are going to discuss how containers living on the same host can communicate with each other and how external clients can access containerized applications if necessary.

Further reading

The following articles provide more in-depth information regarding what was covered in this chapter:

- *Circuit breakers*: `http://bit.ly/1NU1sgW`
- *The OSI model explained*: `http://bit.ly/1UCcvMt`
- *Blue-green deployments*: `http://bit.ly/2r2IxNJ`

Questions

Please answer the following questions to assess your understanding of this chapter's content:

1. When and why does every part in a distributed application architecture have to be redundant? Explain this in a few short sentences.
2. Why do we need DNS services? Explain this in three to five sentences.
3. What is a circuit breaker and why is it needed?
4. What are some of the important differences between a monolithic application and a distributed or multi-service application?
5. What is a blue-green deployment?

Answers

Here are the possible answers to this chapter's questions:

1. In a distributed application architecture, every piece of the software and infrastructure needs to be redundant in a production environment, where the continuous uptime of the application is mission-critical. A highly distributed application consists of many parts and the likelihood of one of the pieces failing or misbehaving increases with the number of parts. It is guaranteed that, given enough time, every part will eventually fail. To avoid outages of the application, we need redundancy in every part, be it a server, a network switch, or a service running on a cluster node in a container.
2. In highly distributed, scalable, and fault-tolerant systems, individual services of the application can move around due to scaling needs or due to component failures. Thus, we cannot hardwire different services with each other. Service A, which needs access to Service B, should not have to know details about Service B, such as its IP address. It should rely on an external provider for this information. TheDNS is such a provider of location information. Service A just tells it that it wants to talk to Service B and the DNS service will figure out the details.

3. A circuit breaker is a means to avoid cascading failures if a component in a distributed application is failing or misbehaving. Similar to a circuit breaker in electric wiring, a software-driven circuit breaker cuts the communication between a client and a failed service. The circuit breaker will directly report an error back to the client component if the failed service is called. This allows the system to recover or heal from failure.
4. A monolithic application is easier to manage than a multi-service application since it consists of a single deployment package. On the other hand, a monolith is often harder to scale to account for increased demand in one particular area of the application. In a distributed application, each service can be scaled individually and each service can run on optimized infrastructure, while a monolith needs to run on infrastructure that is OK for all or most of the features implemented in it. However, over time, this has become less of a problem since very powerful servers and/or VMs are made available by all major cloud providers. These are relatively cheap and can handle the load of most average line-of-business or web applications with ease.

   Maintaining and updating a monolith, if not well modularized, is much harder than a multi-service application, where each service can be updated and deployed independently. The monolith is often a big, complex, and tightly coupled pile of code. Minor modifications can have unexpected side effects. (Micro) Services, in theory, are self-contained, simple components that behave like black boxes. Dependent services know nothing about the inner workings of the service and thus do not depend on it.

> **Note**
>
> The reality is often not so nice – in many cases, microservices are hard coupled and behave like distributed monoliths. Sadly, the latter is the worst place a team or a company can be in as it combines the disadvantages of both worlds, with the monolith on one side and the distributed application on the other.

5. A blue-green deployment is a form of software deployment that allows for zero downtime deployments of new versions of an application or an application service. If, say, Service A needs to be updated with a new version, then we call the currently running blue version. The new version of the service is deployed into production, but not yet wired up with the rest of the application. This new version is called green. Once the deployment succeeds and smoke tests have shown it's ready to go, the router that funnels traffic to the blue version is reconfigured to switch to the green version. The behavior of the green version is observed for a while and if everything is OK, the blue version is decommissioned. On the other hand, if the green version causes difficulties, the router can simply be switched back to the blue version, and the green version can be fixed and later redeployed.

10
Using Single-Host Networking

In the previous chapter, we learned about the most important architectural patterns and best practices that are used when dealing with distributed application architecture.

In this chapter, we will introduce the Docker container networking model and its single-host implementation in the form of the bridge network. This chapter also introduces the concept of **Software Defined Networks** (**SDNs**) and how they are used to secure containerized applications. Furthermore, we will demonstrate how container ports can be opened to the public and thus make containerized components accessible to the outside world. Finally, we will introduce Traefik, a reverse proxy, which can be used to enable sophisticated HTTP application-level routing between containers.

This chapter covers the following topics:

- Dissecting the container network model
- Network firewalling
- Working with the bridge network
- The host and null network
- Running in an existing network namespace
- Managing container ports
- HTTP-level routing using a reverse proxy

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

- Create, inspect, and delete a custom bridge network
- Run a container attached to a custom bridge network
- Isolate containers from each other by running them on different bridge networks
- Publish a container port to a host port of your choice
- Add Traefik as a reverse proxy to enable application-level routing

Technical requirements

For this chapter, the only thing you will need is a Docker host that is able to run Linux containers. You can use your laptop with Docker Desktop for this purpose.

To start with, let's first create a folder for this chapter where we are going to store the code for our examples:

1. Navigate to the folder where you have cloned the repository accompanying this book. Usually, this is the following:

```
$ cd ~/The-Ultimat-Docker-Container-Book
```

2. Create a subfolder for this chapter and navigate to it:

```
$ mkdir ch10 && cd ch10
```

Let's get started!

Dissecting the container network model

So far, we have mostly worked with single containers, but in reality, a containerized business application consists of several containers that need to collaborate to achieve a goal. Therefore, we need a way for individual containers to communicate with each other. This is achieved by establishing pathways, which we can use to send data packets back and forth between containers. These pathways are called networks. Docker has defined a very simple networking model, the so-called **container network model** (**CNM**), to specify the requirements that any software that implements a container network has to fulfill. The following is a graphical representation of the CNM:

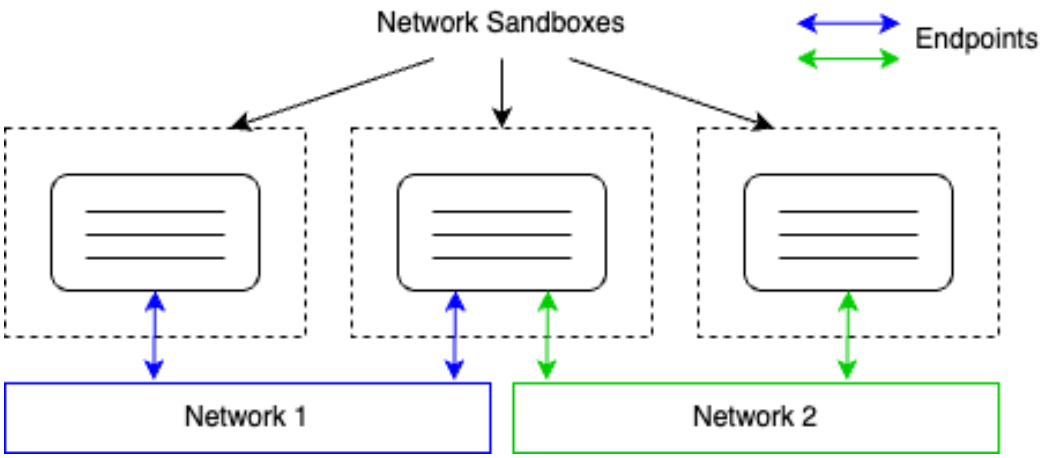


Figure 10.1 – The Docker CNM

The CNM has three elements – sandboxes, endpoints, and networks:

- **Network Sandboxes**: The sandbox perfectly isolates a container from the outside world. No inbound network connection is allowed into the sandboxed container, but it is very unlikely that a container will be of any value in a system if absolutely no communication with it is possible. To work around this, we have element number two, which is the endpoint.
- **Endpoint**: An endpoint is a controlled gateway from the outside world into the network's sandbox, which shields the container. The endpoint connects the network sandbox (but not the container) to the third element of the model, which is the network.
- **Network**: The network is the pathway that transports the data packets of an instance of communication from endpoint to endpoint or, ultimately, from container to container.

It is important to note that a network sandbox can have zero to many endpoints, or, said differently, each container living in a network sandbox can either be attached to no network at all or it can be attached to multiple different networks at the same time. In the preceding diagram, the middle one of the three **Network Sandboxes** is attached to both **Network 1** and **Network 2** using an endpoint.

This networking model is very generic and does not specify where the individual containers that communicate with each other over a network run. All containers could, for example, run on the same host (local) or they could be distributed across a cluster of hosts (global).

Of course, the CNM is just a model describing how networking works among containers. To be able to use networking with our containers, we need real implementations of the CNM. For both local and global scopes, we have multiple implementations of the CNM. In the following table, we've given a short overview of the existing implementations and their main characteristics. The list is in no particular order:

Network	**Company**	**Scope**	**Description**
Bridge	Docker	Local	Simple network based on Linux bridges to allow networking on a single host
Macvlan	Docker	Local	Configures multiple layer-2 (that is, MAC) addresses on a single physical host interface
Overlay	Docker	Global	Multi-node capable container network based on Virtual Extensible LAN (VXLan)
Weave Net	Weaveworks	Global	Simple, resilient, multi-host Docker networking
Contiv Network Plugin	Cisco	Global	Open source container networking

Table 10.1 – Network types

All network types not directly provided by Docker can be added to a Docker host as a plugin.

In the next section, we will describe how network firewalling works.

Network firewalling

Docker has always had the mantra of security first. This philosophy had a direct influence on how networking in a single- and multi-host Docker environment was designed and implemented. SDNs are easy and cheap to create, yet they perfectly firewall containers that are attached to this network from other non-attached containers, and from the outside world. All containers that belong to the same network can freely communicate with each other, while others have no means to do so.

In the following diagram, we have two networks called **front** and **back**. Attached to the **front** network, we have containers **c1** and **c2**, and attached to the **back** network, we have containers **c3** and **c4**. **c1** and **c2** can freely communicate with each other, as can **c3** and **c4**, but **c1** and **c2** have no way to communicate with either **c3** or **c4**, and vice versa:

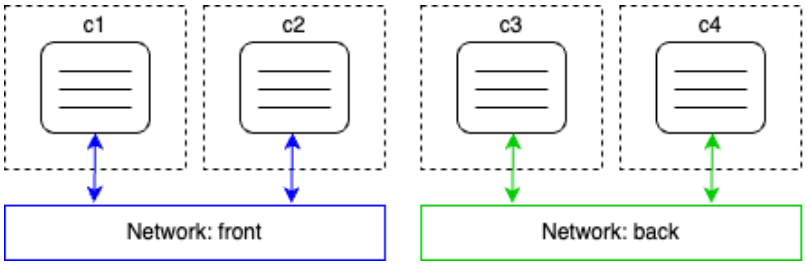


Figure 10.2 – Docker networks

Now, what about a situation in which we have an application consisting of three services: `webAPI`, `productCatalog`, and `database`? We want `webAPI` to be able to communicate with `productCatalog`, but not with the database, and we want `productCatalog` to be able to communicate with the database service. We can solve this situation by placing `webAPI` and the database on different networks and attaching `productCatalog` to both of these networks, as shown in the following diagram:

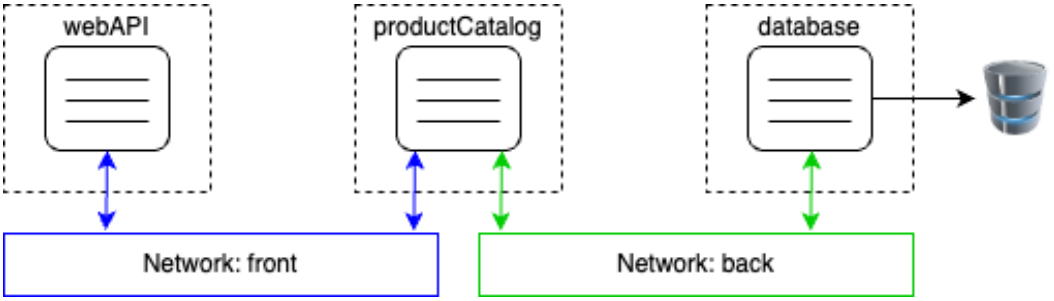


Figure 10.3 – Container attached to multiple networks

Since creating SDNs is cheap, and each network provides added security by isolating resources from unauthorized access, it is highly recommended that you design and run applications so that they use multiple networks and only run services on the same network that absolutely need to communicate with each other. In the preceding example, there is absolutely no need for the `webAPI` component to ever communicate directly with the `database` service, so we have put them on different networks. If the worst-case scenario happens and a hacker compromises `webAPI`, they won't be able to access the database from there without also hacking the `productCatalog` service.

Now we are ready to discuss the first implementation of the CNM, the bridge network.

Working with the bridge network

The Docker bridge network is the first implementation of the CNM that we're going to look at in detail. This network implementation is based on the Linux bridge.

When the Docker daemon runs for the first time, it creates a Linux bridge and calls it `docker0`. This is the default behavior and can be changed by changing the configuration.

Docker then creates a network with this Linux bridge and calls it the network bridge. All the containers that we create on a Docker host and that we do not explicitly bind to another network lead to Docker automatically attaching the containers to this bridge network.

To verify that we indeed have a network called `bridge` of the `bridge` type defined on our host, we can list all the networks on the host with the following command:

```
$ docker network ls
```

This should provide an output similar to the following:

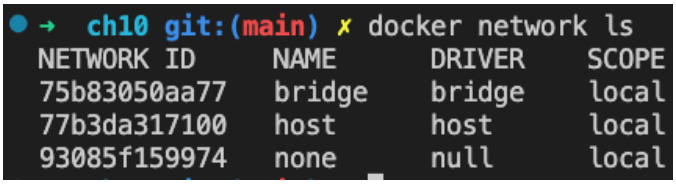


Figure 10.4 – Listing all the Docker networks available by default

In your case, the IDs will be different, but the rest of the output should look the same. We do indeed have a first network called `bridge` using the `bridge` driver. The scope being `local` just means that this type of network is restricted to a single host and cannot span across multiple hosts. In *Chapter 14*, *Introducing Docker Swarm*, we will also discuss other types of networks that have a global scope, meaning they can span whole clusters of hosts.

Now, let's look a little bit deeper into what this bridge network is all about. For this, we are going to use the Docker `inspect` command:

```
$ docker network inspect bridge
```

When executed, this outputs a big chunk of detailed information about the network in question. This information should look as follows:

```
● → ch10 git:(main) ✗ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "75b83050aa772bdfed2f0de1ce5243276e22bd35eda514a24ae6fa4f330e587b",
        "Created": "2023-02-15T16:23:15.807986084Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
```

Figure 10.5 – Output generated when inspecting the Docker bridge network

We saw the `ID`, `Name`, `Driver`, and `Scope` values when we listed all the networks, so that is nothing new, but let's have a look at the **IP address management** (**IPAM**) block.

IPAM is a piece of software that is used to track the IP addresses that are used on a computer. The important part of the IPAM block is the *config* node with its values for the subnet and gateway. The subnet for the bridge network is defined by default as `172.17.0.0/16`. This means that all containers attached to this network will get an IP address assigned by Docker that is taken from the given range, which is `172.17.0.2` to `172.17.255.255`. The `172.17.0.1` address is reserved for the router of this network whose role in this type of network is taken by the Linux bridge. We can expect that the very first container that will be attached to this network by Docker will get the `172.17.0.2` address. All subsequent containers will get a higher number; the following diagram illustrates this fact:

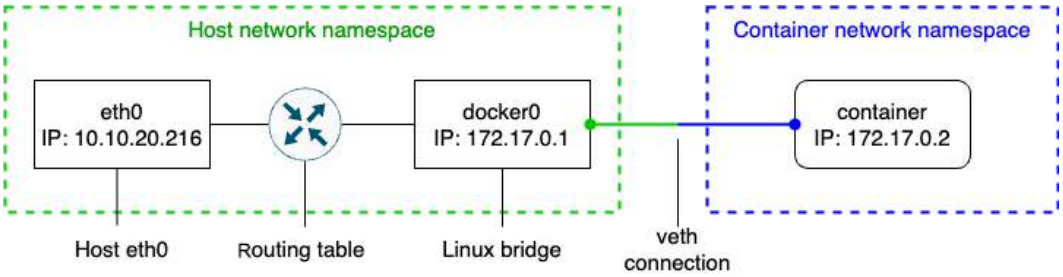


Figure 10.6 – The bridge network

In the preceding diagram, we can see the network namespace of the host, which includes the host's `eth0` endpoint, which is typically an NIC if the Docker host runs on bare metal or a virtual NIC if the Docker host is a VM. All traffic to the host comes through `eth0`. The Linux bridge is responsible for routing the network traffic between the host's network and the subnet of the bridge network.

What is a NIC?

A **Network Interface Card** (**NIC**), sometimes referred to as a network interface connector, is a hardware component that enables a computer or device to connect to a network. It serves as an interface between the computer and the network, allowing data to be transmitted and received. NICs are typically built-in components on motherboards or installed as expansion cards and support various types of network connections, such as Ethernet, Wi-Fi, or fiber-optic connections.

By default, only egress traffic is allowed, and all ingress is blocked. What this means is that while containerized applications can reach the internet, they cannot be reached by any outside traffic. Each container attached to the network gets its own **virtual ethernet** (**veth**) connection to the bridge. This is illustrated in the following diagram:

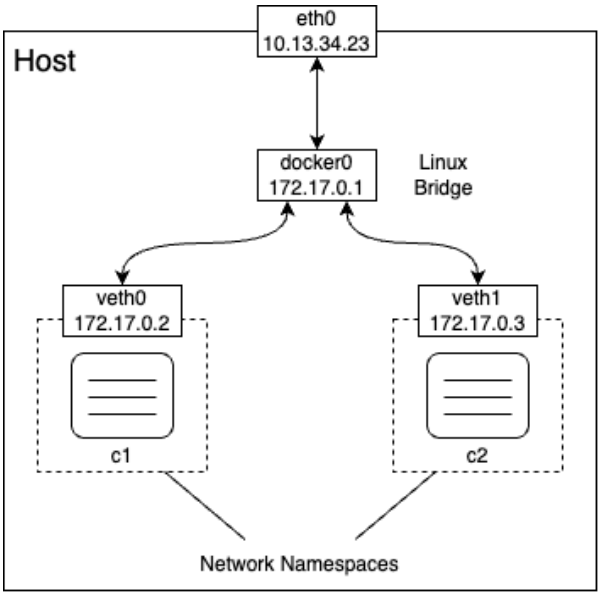


Figure 10.7 – Details of the bridge network

The preceding diagram shows us the world from the perspective of the host. We will explore what this situation looks like from within a container later on in this section. We are not limited to just the bridge network, as Docker allows us to define our own custom bridge networks. This is not just a feature that is nice to have; it is a recommended best practice not to run all containers on the same network. Instead, we should use additional bridge networks to further isolate containers that have no need to communicate with each other. To create a custom bridge network called `sample-net`, use the following command:

```
$ docker network create --driver bridge sample-net
```

If we do this, we can then inspect what subnet Docker has created for this new custom network, as follows:

```
$ docker network inspect sample-net | grep Subnet
```

This returns the following value:

```
"Subnet": "172.18.0.0/16",
```

Evidently, Docker has just assigned the next free block of IP addresses to our new custom bridge network. If, for some reason, we want to specify our own subnet range when creating a network, we can do so by using the `--subnet` parameter:

```
$ docker network create --driver bridge --subnet "10.1.0.0/16" test-
net
```

> **Note**
>
> To avoid conflicts due to duplicate IP addresses, make sure you avoid creating networks with overlapping subnets.

Now that we have discussed what a bridge network is and how we can create a custom bridge network, we want to understand how we can attach containers to these networks.

First, let's interactively run an Alpine container without specifying the network to be attached:

```
$ docker container run --name c1 -it --rm alpine:latest /bin/sh
```

In another Terminal window, let's inspect the `c1` container:

```
$ docker container inspect c1
```

In the vast output, let's concentrate for a moment on the part that provides network-related information. This can be found under the `NetworkSettings` node. I have it listed in the following output:

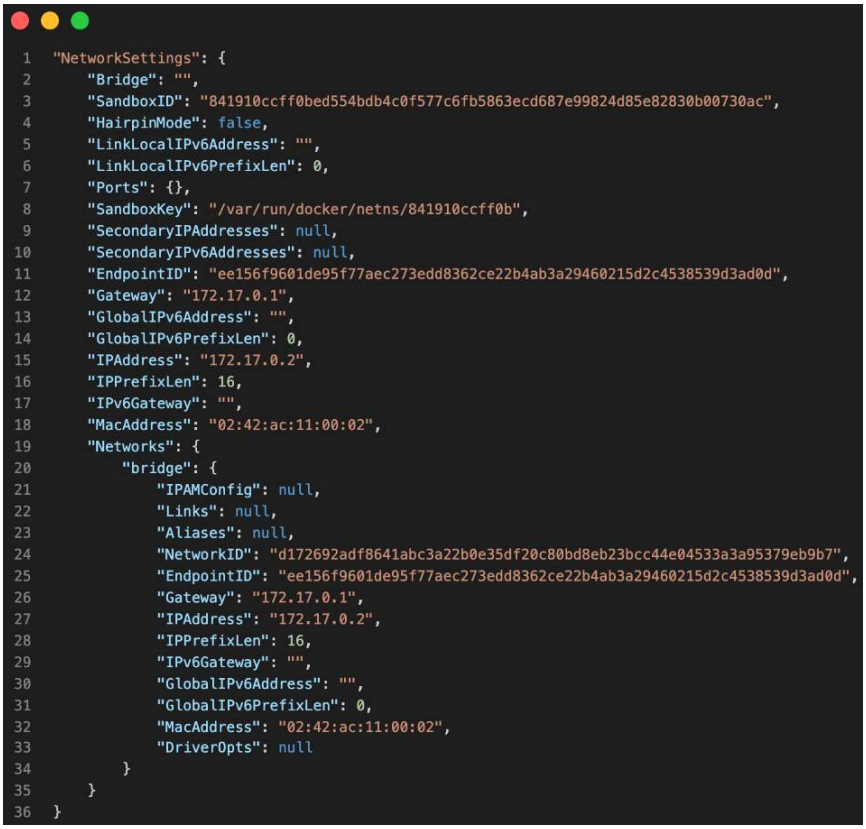


Figure 10.8 – The NetworkSettings section of the container metadata

In the preceding output, we can see that the container is indeed attached to the bridge network since `NetworkID` is equal to `d172692...`, which we can see from the preceding code being the ID of the bridge network. We can also see that the container was assigned the IP address of `172.17.0.2` as expected and that the gateway is at `172.17.0.1`.

Please note that the container also had a `MacAddress` associated with it. This is important as the Linux bridge uses the `MacAddress` for routing.

So far, we have approached this from the outside of the container's network namespace. Now, let's see what the situation looks like when we're not only inside the container but inside the container's network namespace. Inside the `c1` container, let's use the `ip` tool to inspect what's going on. Run the `ip addr` command and observe the output that is generated, as follows:

```
/ # ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN qlen 1000
    link/ipip 0.0.0.0 brd 0.0.0.0
3: ip6tnl0@NONE: <NOARP> mtu 1452 qdisc noop state DOWN qlen 1000
    link/tunnel6 00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00 brd 00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00
54: eth0@if55: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

Figure 10.9 – Container namespace, as seen by the IP tool

The interesting part of the preceding output is `54:`, that is, the `eth0` endpoint. The `veth0` endpoint that the Linux bridge created outside of the container namespace is mapped to `eth0` inside the container. Docker always maps the first endpoint of a container network namespace to `eth0`, as seen from inside the namespace. If the network namespace is attached to an additional network, then that endpoint will be mapped to `eth1`, and so on.

Since at this point, we're not really interested in any endpoint other than `eth0`, we could have used a more specific variant of the command, which would have given us the following:

```
/ # ip addr show eth0
54: eth0@if55: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

Figure 10.10 – eth0 endpoint as seen from inside of the container

In the output, we can also see what MAC address (`02:42:ac:11:00:02`) and what IP (`172.17.0.2`) have been associated with this container network namespace by Docker.

We can also get some information about how requests are routed by using the `ip route` command:

```
/ # ip route
```

This gives us the following output:

```
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0 scope link  src 172.17.0.2
```

This output tells us that all the traffic to the gateway at `172.17.0.1` is routed through the `eth0` device.

Now, let's run another container called `c2` on the same network and in `detach` mode:

```
$ docker container run --name c2 -d --rm alpine:latest ping 127.0.0.1
```

The `c2` container will also be attached to the bridge network since we have not specified any other network. Its IP address will be the next free one within the subnet, which is `172.17.0.3`, as we can readily test with the following command:

```
$ docker container inspect --format "{{.NetworkSettings.IPAddress}}"
c2
```

This results in the following output:

```
172.17.0.3
```

Now, we have two containers attached to the bridge network. We can try to inspect this network once again to find a list of all containers attached to it in the output:

```
$ docker network inspect bridge
```

This information can be found under the `Containers` node:

Figure 10.11 – The Containers section of the output of the Docker network inspect bridge

Once again, we have shortened the output to the relevant part for readability.

Now, let's create two additional containers, `c3` and `c4`, and attach them to `sample-net`, which we created earlier. For this, we'll use the `--network` parameter:

```
$ docker container run --name c3 --rm -d \
    --network sample-net \
    alpine:latest ping 127.0.0.1

$ docker container run --name c4 --rm -d \
    --network sample-net \
    alpine:latest ping 127.0.0.1
```

Let's inspect the `sample-net` network and confirm that `c3` and `c4` are indeed attached to it:

```
$ docker network inspect sample-net
```

This will give us the following output for the `Containers` section:


```
"Containers": {
    "852c4ff38de3c2029b25c4c6076a7bbb28aeea48eeea603d2c4eeccda0b50dc1": {
        "Name": "c3",
        "EndpointID": "36fff74b9b5138ffa04e1d031e5a0cd7a33b334289e9cf578c6dd885f3ad4a9f",
        "MacAddress": "02:42:ac:14:00:02",
        "IPv4Address": "172.20.0.2/16",
        "IPv6Address": ""
    },
    "cca95c07251705c33631fa8c08b3070fd3e79c1612dfa5c350b3fde97600a312": {
        "Name": "c4",
        "EndpointID": "0c4e0b4daf99e16d12957297d07d2f2c440c8252646d7a45794b0f7f3d64efbf",
        "MacAddress": "02:42:ac:14:00:03",
        "IPv4Address": "172.20.0.3/16",
        "IPv6Address": ""
    }
},
```


Figure 10.12 – The Containers section of the Docker network inspect test-net command

The next question we're going to ask ourselves is whether the `c3` and `c4` containers can freely communicate with each other. To demonstrate that this is indeed the case, we can `exec` into the `c3` container:

```
$ docker container exec -it c3 /bin/sh
```

Once inside the container, we can try to `ping` container `c4` by name and by IP address:

```
/ # ping c4
```

We should get this output:

```
PING c4 (172.20.0.3): 56 data bytes
64 bytes from 172.20.0.3: seq=0 ttl=64 time=3.092 ms
64 bytes from 172.20.0.3: seq=1 ttl=64 time=0.481 ms
...
```

Instead of the container name, here, we use `c4`'s IP address:

```
/ # ping 172.20.0.3
```

We should see the following result:

```
PING 172.20.0.3 (172.20.0.3): 56 data bytes
64 bytes from 172.20.0.3: seq=0 ttl=64 time=0.200 ms
64 bytes from 172.20.0.3: seq=1 ttl=64 time=0.172 ms
...
```

The answer in both cases confirms to us that the communication between containers attached to the same network is working as expected. The fact that we can even use the name of the container we want to connect to shows us that the name resolution provided by the Docker DNS service works inside this network.

Now, we want to make sure that the `bridge` and `sample-net` networks are firewalled from each other. To demonstrate this, we can try to `ping` the `c2` container from the `c3` container, either by its name or by its IP address. Let's start with pinging by name:

```
/ # ping c2
```

This results in the following output:

```
ping: bad address 'c2'
```

The following is the result of the ping using the IP address of the `c2` container instead:

```
/ # ping 172.17.0.3
```

It gives us this output:

```
PING 172.17.0.3 (172.17.0.3): 56 data bytes
^C
--- 172.17.0.3 ping statistics ---
11 packets transmitted, 0 packets received, 100% packet loss
```

The preceding command remained hanging and I had to terminate the command with *Ctrl* + *C*. From the output of pinging `c2`, we can also see that the name resolution does not work across networks. This is the expected behavior. Networks provide an extra layer of isolation, and thus security, to containers.

Earlier, we learned that a container can be attached to multiple networks. Let's first create a network called `test-net`. Note that the following command does not define the driver of the network; thus, the default driver is used, which happens to be the bridge driver:

```
$ docker network create test-net
```

Then, we attach a container, `c5`, to our `sample-net` network:

```
$ docker container run --name c5 --rm -d \
    --network sample-net
    alpine:latest ping 127.0.0.1
```

Then, we attach the `c6` container to the `sample-net` and `test-net` networks at the same time:

```
$ docker container run --name c6 --rm -d \
    --network sample-net \
    alpine:latest ping 127.0.0.1
$ docker network connect test-net c6
```

Now, we can test that `c6` is reachable from the `c5` container attached to the `test-net` network, as well as from the `c3` container attached to the `sample-net` network. The result will show that the connection indeed works.

If we want to remove an existing network, we can use the `docker network rm` command, but note that we cannot accidentally delete a network that has containers attached to it:

```
$ docker network rm test-net
```

It results in this output:

```
Error response from daemon: network test-net id 455c922e... has active
endpoints
```

Before we continue, let's clean up and remove all the containers:

```
$ docker container rm -f $(docker container ls -aq)
```

Now, we can remove the two custom networks that we created:

```
$ docker network rm sample-net
$ docker network rm test-net
```

Alternatively, we could remove all the networks that no container is attached to with the `prune` command:

```
$ docker network prune --force
```

I used the `--force` (or `-f`) argument here to prevent Docker from reconfirming that I really want to remove all unused networks.

Double-check with the `docker network ls` command that you are only left with the three default networks provided by Docker.

The next network types we are going to inspect a bit are the `host` and `null` network types.

The host and null networks

In this section, we are going to look at two predefined and somewhat unique types of networks, the host and the null networks. Let's start with the former.

The host network

There are occasions when we want to run a container in the network namespace of the host. This may be necessary when we need to run some software in a container that is used to analyze or debug the host network's traffic, but keep in mind that these are very specific scenarios. When running business software in containers, there is no good reason to ever run the respective containers attached to the host's network. For security reasons, it is strongly recommended that you do not run any such container attached to the host network in a production or production-like environment.

That said, how can we run a container inside the network namespace of the host? Simply by attaching the container to the `host` network:

1. Run an Alpine container and attach it to the `host` network:

```
$ docker container run --rm -it \
    --network host \
    alpine:latest /bin/sh
```

2. Use the `ip` tool to analyze the network namespace from within the container. You will see that we get exactly the same picture as we would if we were running the `ip` tool directly on the host. For example, I inspect the `eth0` device on my laptop with the following:

```
/ # ip addr show eth0
```

As a result, I get this:

```
/ # ip addr show eth0
8: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether de:36:11:0b:28:23 brd ff:ff:ff:ff:ff:ff
    inet 192.168.65.4 peer 192.168.65.5/32 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::dc36:11ff:fe0b:2823/64 scope link
       valid_lft forever preferred_lft forever
```

Figure 10.13 – Showing the eth0 device from inside a container

Here, I can see that `192.168.65.3` is the IP address that the host has been assigned and that the MAC address shown here also corresponds to that of the host.

3. We can also inspect the routes:

```
/ # ip route
```

On my MacBook Air M1, this is what I get:

```
/ # ip route
default via 192.168.65.5 dev eth0
172.17.0.0/16 dev docker0 scope link  src 172.17.0.1
192.168.65.5 dev eth0 scope link  src 192.168.65.4
```

Figure 10.14 – Routes from within a container

Before we move on to the next section of this chapter, I want to once again point out that running a container on the host network can be dangerous due to potential security vulnerabilities and conflicts:

- **Security risks**: By using the host network, the container has the same network access as the host machine. This means that if an application running within the container has a vulnerability that is exploited, the attacker could gain access to the host network and potentially compromise other services or data.
- **Port conflicts**: When a container uses the host network, it shares the same network namespace as the host. This means that if your containerized application and a host application listen on the same port, there can be conflicts.
- **Isolation**: One of the major benefits of using Docker is the isolation it provides at various levels (process, filesystem, or network). By using the host network, you lose this level of isolation, which could lead to unforeseen issues.

Therefore, it's generally recommended to use a user-defined network instead of the host network when running Docker containers, as it provides better isolation and reduces the risk of conflicts and security vulnerabilities.

The null network

Sometimes, we need to run a few application services or jobs that do not need any network connection at all to execute the task at hand. It is strongly advised that you run those applications in a container that is attached to the `none` network. This container will be completely isolated and is thus safe from any outside access. Let's run such a container:

```
$ docker container run --rm -it \
    --network none \
    alpine:latest /bin/sh
```

Once inside the container, we can verify that there is no `eth0` network endpoint available:

```
/ # ip addr show eth0
ip: can't find device 'eth0'
```

There is also no routing information available, as we can demonstrate by using the following command:

```
/ # ip route
```

This returns nothing.

In the following section, we are going to learn how we can run a container inside the existing network namespace of another container.

Running in an existing network namespace

Normally, Docker creates a new network namespace for each container we run. The network namespace of the container corresponds to the sandbox of the container network model we described earlier on. As we attach the container to a network, we define an endpoint that connects the container network namespace to the actual network. This way, we have one container per network namespace.

Docker provides an additional way for us to define the network namespace that a container runs in. When creating a new container, we can specify that it should be attached to (or maybe we should say included in) the network namespace of an existing container. With this technique, we can run multiple containers in a single network namespace:

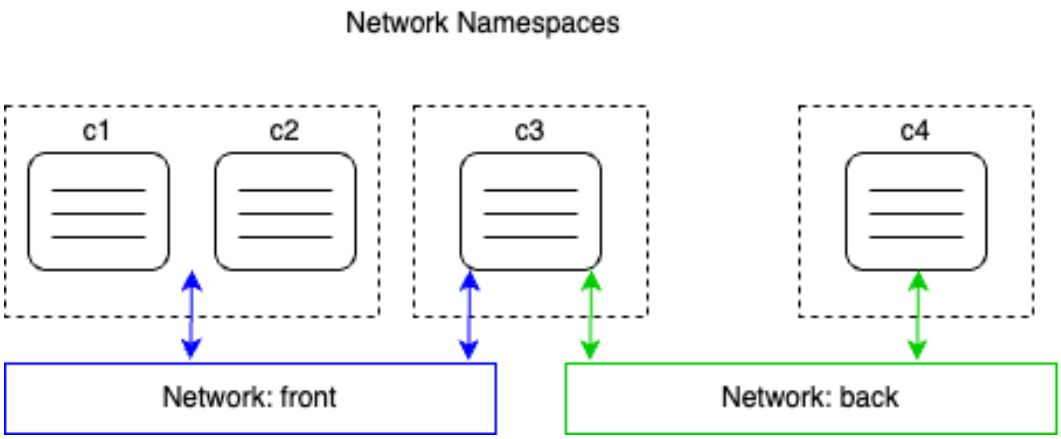


Figure 10.15 – Multiple containers running in a single network namespace

In the preceding diagram, we can see that in the leftmost network namespace, we have two containers. The two containers, since they share the same namespace, can communicate on `localhost` with each other. The network namespace (and not the individual containers) is then attached to the **front** network.

This is useful when we want to debug the network of an existing container without running additional processes inside that container. We can just attach a special utility container to the network namespace of the container to inspect. This feature is also used by Kubernetes when it creates a Pod. We will learn more about Kubernetes and Pods in subsequent chapters of this book.

Now, let's demonstrate how this works:

1. First, we create a new `bridge` network:

```
$ docker network create --driver bridge test-net
```

2. Next, we run a container attached to this network:

```
$ docker container run --name web -d \
    --network test-net \
    nginx:alpine
```

3. Finally, we run another container and attach it to the network of our web container:

```
$ docker container run -it --rm \
    --network container:web \
    alpine:latest /bin/sh
```

   Specifically, note how we define the network: `--network container:web`. This tells Docker that our new container will use the same network namespace as the container called `web`.

4. Since the new container is in the same network namespace as the web container running nginx, we're now able to access nginx on `localhost`! We can prove this by using the `wget` tool, which is part of the Alpine container, to connect to nginx. We should see the following:

```
/ # wget -qO - localhost
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
</html>
```

   Note that we have shortened the output for readability. Please also note that there is an important difference between running two containers attached to the same network and two containers running in the same network namespace. In both cases, the containers can freely communicate with each other, but in the latter case, the communication happens over `localhost`.

5. To clean up the container and network, we can use the following command:

```
$ docker container rm --force web
$ docker network rm test-net
```

In the next section, we are going to learn how to expose container ports on the container host.

Managing container ports

Now that we know how we can isolate firewall containers from each other by placing them on different networks, and that we can have a container attached to more than one network, we have one problem that remains unsolved. How can we expose an application service to the outside world? Imagine a container running a web server hosting our `webAPI` from before. We want customers from the internet to be able to access this API. We have designed it to be a publicly accessible API. To achieve this, we have to, figuratively speaking, open a gate in our firewall through which we can funnel external traffic to our API. For security reasons, we don't just want to open the doors wide; we want to have a single controlled gate that traffic flows through.

We can create this kind of gate by mapping a container port to an available port on the host. We're also calling this container port to publish a port. Remember that the container has its own virtual network stack, as does the host. Therefore, container ports and host ports exist completely independently and by default have nothing in common at all, but we can now wire a container port with a free host port and funnel external traffic through this link, as illustrated in the following diagram:

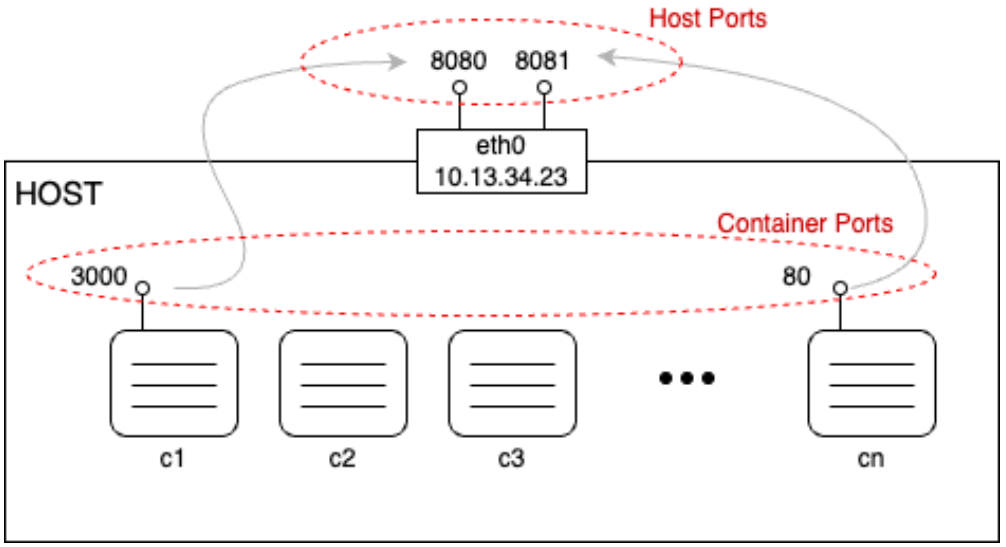


Figure 10.16 – Mapping container ports to host ports

But now, it is time to demonstrate how we can actually map a container port to a host port. This is done when creating a container. We have different ways of doing so:

1. First, we can let Docker decide which host port our container port should be mapped to. Docker will then select one of the free host ports in the range of `32xxx`. This automatic mapping is done by using the `-P` parameter:

```
$ docker container run --name web -P -d nginx:alpine
```

The preceding command runs an nginx server in a container. nginx is listening at port `80` inside the container. With the `-P` parameter, we're telling Docker to map all the exposed container ports to a free port in the `32xxx` range. We can find out which host port Docker is using by using the `docker container port` command:

```
$ docker container port web
80/tcp -> 0.0.0.0:32768
```

The nginx container only exposes port `80`, and we can see that it has been mapped to the host port `32768`. If we open a new browser window and navigate to `localhost:32768`, we should see the following screen:

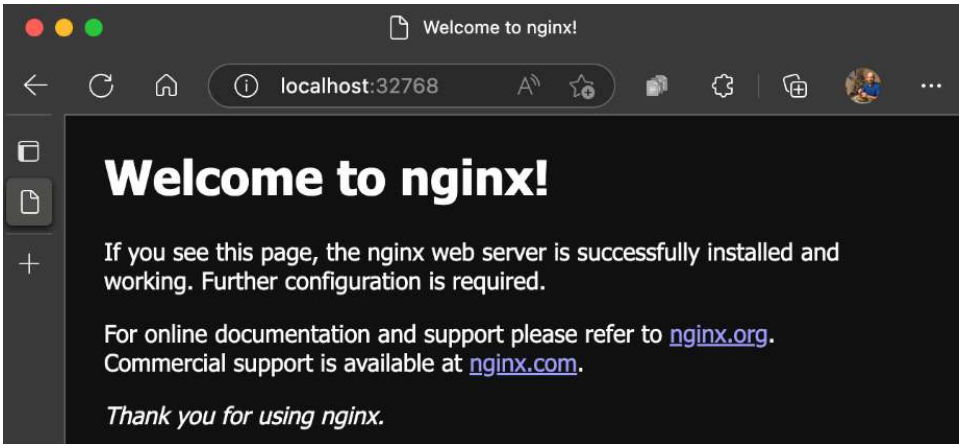


Figure 10.17 – The welcome page of nginx

2. An alternative way to find out which host port Docker is using for our container is to inspect it. The host port is part of the `NetworkSettings` node:

```
$ docker container inspect web | grep HostPort
        "HostPort": "32768"
```

3. Finally, the third way of getting this information is to list the container:

```
$ docker container ls
CONTAINER ID IMAGE ... PORTS NAMES
56e46a14b6f7 nginx:alpine ... 0.0.0.0:32768->80/tcp web
```

Please note that in the preceding output, the `/tcp` part tells us that the port has been opened for communication with the TCP protocol, but not for the UDP protocol. TCP is the default, and if we want to specify that we want to open the port for UDP, then we have to specify this explicitly. The special (IP) address, `0.0.0.0`, in the mapping tells us that traffic from any host IP address can now reach container port `80` of the web container.

4. Sometimes, we want to map a container port to a very specific host port. We can do this by using the `-p` parameter (or `--publish`). Let's look at how this is done with the following command:

```
$ docker container run --name web2 -p 8080:80 -d nginx:alpine
```

The value of the `-p` parameter is in the form of `<host port>:<container port>`. Therefore, in the preceding case, we map container port `80` to host port `8080`. Once the `web2` container runs, we can test it in the browser by navigating to `localhost:8080`, and we should be greeted by the same nginx welcome page that we saw in the previous example that dealt with automatic port mapping.

When using the UDP protocol for communication over a certain port, the publish parameter will look like so: `-p 3000:4321/udp`. Note that if we want to allow communication with both TCP and UDP protocols over the same port, then we have to map each protocol separately.

In the next section, we will talk about HTTP routing using a reverse proxy.

HTTP-level routing using a reverse proxy

Imagine you have been tasked with containerizing a monolithic application. The application has organically evolved over the years into an unmaintainable behemoth. Changing even a minor feature in the source code may break other features due to the tight coupling that exists in the code base. Releases are rare due to their complexity and require the whole team to be on board. The application has to be taken down during the release window, which costs the company a lot of money due to lost opportunities, not to mention their loss of reputation.

Management has decided to put an end to that vicious cycle and improve the situation by containerizing the monolith. This alone will lead to a massively decreased time between releases, as witnessed within the industry. As a later step, the company wants to break out every piece of functionality from the monolith and implement it as a microservice. This process will continue until the monolith has been completely starved.

But it is this second point that leads to some head-scratching for the team involved. How will we break down the monolith into loosely coupled microservices without affecting all the many clients of the monolith out there? The public API of the monolith, though very complex, has a well-structured design. Public URIs were carefully crafted and should not be changed at any cost. For example, there is a product catalog function implemented in the app that can be accessed via `https://acme.com/catalog?category=bicycles` so that we can access a list of bicycles offered by the company.

On the other hand, there is a URL called `https://acme.com/checkout` that we can use to initiate the checkout of a customer's shopping cart, and so on. I hope it is clear where we are going with this.

Containerizing the monolith

Let's start with the monolith. I have prepared a simple code base that has been implemented in Python 3.7 and uses Flask to implement the public REST API. The sample app is not really a full-blown application but just complex enough to allow for some redesign. The sample code can be found in the `ch10/e-shop` folder. Inside this folder is a subfolder called `monolith` containing the Python application. Follow these steps:

1. In a new Terminal window, navigate to that folder, install the required dependencies, and run the application:

```
$ cd ~/The-Ultimate-Docker-Container-Book
$ cd ch10/e-shop/monolith
$ pip install -r requirements.txt
$ export FLASK_APP=main.py
$ flask run
```

   The application will start and listen on localhost on port `5000`:

```
➜  monolith-solved git:(main) ✗ flask run
 * Serving Flask app 'main.py'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

Figure 10.18 – Running the Python monolith

2. We can use `curl` to test the app. Open another Terminal window and use the following command to retrieve a list of all the bicycles the company offers:

```
$ curl localhost:5000/catalog?type=bicycle
```

   This results in the following output:

```
[{"id": 1, "name": "Mountanbike Driftwood 24", "unitPrice":
199},
{"id": 2, "name": "Tribal 100 Flat Bar Cycle Touring Road Bike",
"unitPrice": 300}, {"id": 3, "name": "Siech Cycles Bike (58
cm)",
"unitPrice": 459}]
```

   Here, we have a JSON-formatted list of three types of bicycles. OK – so far, so good.

3. Now, let's change the `hosts` file, add an entry for `acme.com`, and map it to `127.0.0.1`, the loop-back address. This way, we can simulate a real client accessing the app at `http://acme.com/catalog?type=bicycle` instead of using `localhost`. You need to use `sudo` to edit the `/etc/hosts` file on a macOS or on Linux. You should add a line to the `hosts` file that looks like this:

```
127.0.0.1 acme.com
```

Windows host file

On Windows, you can edit the file by, for example, running Notepad as an administrator, opening the `c:\Windows\System32\Drivers\etc\hosts` file, and modifying it.

4. Save your changes and assert that it works by pinging `acme.com`:

```
$ ping acme.com
PING acme.com (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=55 time<1 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=55 time<1 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=55 time<1 ms
...
```

After all this, it is time to containerize the application. The only change we need to make to the application is ensuring that we have the application web server listening on `0.0.0.0` instead of `localhost`.

5. We can do this easily by modifying the application and adding the following start logic at the end of `main.py`:

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

6. Then, we can start the application as follows:

```
$ python main.py.
```

7. Now, add a Dockerfile to the monolith folder with the following content:

```
1  FROM python:3.7-alpine
2  WORKDIR /app
3  COPY requirements.txt ./
4  RUN pip install -r requirements.txt
5  COPY . .
6  EXPOSE 5000
7  CMD python main.py
```

Figure 10.19 – The Dockerfile for the monolith

8. In your Terminal window, from within the monolith folder, execute the following command to build a Docker image for the application:

```
$ docker image build -t acme/eshop:1.0 .
```

9. After the image has been built, try to run the application:

```
$ docker container run --rm -it \
    --name eshop \
    -p 5000:5000 \
    acme/eshop:1.0
```

Notice that the output from the app now running inside a container is indistinguishable from what we got when running the application directly on the host. We can now test whether the application still works as before by using the two `curl` commands to access the catalog and the checkout logic:

```
● → The-Ultimate-Docker-Container-Book git:(main) ✗ curl http://acme.com:5000/catalog\?type\=bicycle
[{"id": 1, "name": "Mountanbike Driftwood 24\"", "unitPrice": 199}, {"id": 2, "name": "Tribal 100 Flat Bar Cycl
e Touring Road Bike", "unitPrice": 300}, {"id": 3, "name": "Siech Cycles Bike (58 cm)", "unitPrice": 459}]%
● → The-Ultimate-Docker-Container-Book git:(main) ✗ curl http://acme.com:5000/checkout
Starting checkout of your shopping cart...%
○ → The-Ultimate-Docker-Container-Book git:(main) ✗
```

Figure 10.20 – Testing the monolith while running in a container

Evidently, the monolith still works exactly the same way as before, even when using the correct URL, that is, `http://acme.com`. Great! Now, let's break out part of the monolith's functionality into a Node.js microservice, which will be deployed separately.

Extracting the first microservice

The team, after some brainstorming, has decided that the catalog product is a good candidate for the first piece of functionality that is cohesive yet self-contained enough to be extracted from the monolith. They decide to implement the product catalog as a microservice implemented in Node.js.

You can find the code they came up with and the Dockerfile in the `catalog` subfolder of the project folder, that is, `e-shop`. It is a simple Express.js application that replicates the functionality that was previously available in the monolith. Let's get started:

1. In your Terminal window, from within the `catalog` folder, build the Docker image for this new microservice:

```
$ docker image build -t acme/catalog:1.0 .
```

2. Then, run a container from the new image you just built:

```
$ docker run --rm -it --name catalog -p 3000:3000 \
    acme/catalog:1.0
```

3. From a different Terminal window, try to access the microservice and validate that it returns the same data as the monolith:

```
$ curl http://acme.com:3000/catalog?type=bicycle
```

Please notice the differences in the URL compared to when accessing the same functionality in the monolith. Here, we are accessing the microservice on port `3000` (instead of `5000`).

But we said that we didn't want to have to change the clients that access our e-shop application. What can we do? Luckily, there are solutions to problems like this. We need to reroute incoming requests. We'll show you how to do this in the next section.

Using Traefik to reroute traffic

In the previous section, we realized that we would have to reroute incoming traffic with a target URL starting with `http://acme.com:5000/catalog` to an alternative URL such as `product-catalog:3000/catalog`. We will be using Traefik to do exactly that.

Traefik is a cloud-native edge router and it is open source, which is great for our specific case. It even has a nice web UI that you can use to manage and monitor your routes. Traefik can be combined with Docker in a very straightforward way, as we will see in a moment.

To integrate well with Docker, Traefik relies on the metadata found for each container or service. This metadata can be applied in the form of labels that contain the routing information:

1. First, let's look at how to run the `catalog` service. Here is the Docker `run` command:

```
$ docker container run --rm -d \
    --name catalog \
    --label traefik.enable=true \
    --label traefik.port=3000 \
    --label traefik.priority=10 \
    --label traefik.http.routers.catalog.rule=\
             "Host(\"acme.com\") && PathPrefix(\"/catalog\")" \
    acme/catalog:1.0
```

   Let's quickly look at the four labels we define:

   - `traefik.enable=true`: This tells Traefik that this particular container should be included in the routing (the default is `false`).
   - `traefik.port=3000`: The router should forward the call to port `3000` (which is the port that the Express.js app is listening on).
   - `traefik.priority=10`: This gives this route high priority. We will see why in a second.
   - `traefik.http.routers.catalog.rule="Host(\"acme.com\") && PathPrefix(\"/catalog\")"`: The route must include the hostname, `acme.com`, and the path must start with `/catalog` in order to be rerouted to this service. As an example, `acme.com/catalog?type=bicycles` would qualify for this rule.
   - Please note the special form of the fourth label. Its general form is `traefik.http.routers.<service name>.rule`.

2. Now, let's look at how we can run the `eshop` container:

```
$ docker container run --rm -d \
    --name eshop \
    --label traefik.enable=true \
    --label traefik.port=5000 \
    --label traefik.priority=1 \
    --label traefik.http.routers.eshop.rule=\
              "Host(\"acme.com\")" \
    acme/eshop:1.0
```

   Here, we forward any matching calls to port `5000`, which corresponds to the port where the `eshop` application is listening. Pay attention to the priority, which is set to `1` (low). This, in combination with the high priority of the catalog service, allows us to filter out all URLs starting with `/catalog` and redirect them to the `catalog` service, while all other URLs will go to the `eshop` service.

3. Now, we can finally run Traefik as the edge router that will serve as a reverse proxy in front of our application. This is how we start it:

```
$ docker run -d \
    --name traefik \
    -p 8080:8080 \
    -p 80:80 \
    -v /var/run/docker.sock:/var/run/docker.sock \
    traefik:v2.0 --api.insecure=true --providers.docker
```

   Note how we mount the Docker socket into the container using the `-v` (or `--volume`) parameter so that Traefik can interact with the Docker engine. We will be able to send web traffic to port `80` of Traefik, from where it will be rerouted according to our rules in the routing definitions found in the metadata of the participating container. Furthermore, we can access the web UI of Traefik via port `8080`.

4. Now that everything is running, that is, the monolith, the first microservice called `catalog`, and Traefik, we can test whether everything works as expected. Use `curl` once again to do so:

```
$ curl http://acme.com/catalog?type=bicycles
$ curl http://acme.com/checkout
```

   As we mentioned earlier, we are now sending all traffic to port `80`, which is the port Traefik is listening on. This proxy will then reroute the traffic to the correct destination.

5. Before proceeding, stop and remove all containers:

```
$ docker container rm -f traefik eshop catalog
```

That's it for this chapter.

Summary

In this chapter, we learned about how containers running on a single host can communicate with each other. First, we looked at the CNM, which defines the requirements of a container network, and then we looked at several implementations of the CNM, such as the bridge network. We then looked at how the bridge network functions in detail and also what kind of information Docker provides us with about the networks and the containers attached to those networks. We also learned about adopting two different perspectives, from both outside and inside the container.

In the next chapter, we're going to introduce Docker Compose. We will learn about creating an application that consists of multiple services, each running in a container, and how Docker Compose allows us to easily build, run, and scale such an application using a declarative approach.

Further reading

Here are some articles that describe the topics that were presented in this chapter in more detail:

- *Docker networking overview*: `http://dockr.ly/2sXGzQn`
- *What is a bridge?*: `https://bit.ly/2HyC3Od`
- *Using bridge networks*: `http://dockr.ly/2BNxjRr`
- *Using Macvlan networks*: `http://dockr.ly/2ETjy2x`
- *Networking using the host network*: `http://dockr.ly/2F4aI59`

Questions

To assess the skills that you have gained from this chapter, please try to answer the following questions:

1. Name the three core elements of the **container network model** (**CNM**).
2. How do you create a custom `bridge` network called, for example, `frontend`?
3. How do you run two `nginx:alpine` containers attached to the frontend network?
4. For the frontend network, get the following:
    - The IPs of all the attached containers
    - The subnet associated with the network
5. What is the purpose of the `host` network?
6. Name one or two scenarios where the use of the `host` network is appropriate.
7. What is the purpose of the `none` network?

8. In what scenarios should the `none` network be used?
9. Why would we use a reverse proxy such as Traefik together with our containerized application?

Answers

Here are example answers for the questions of this chapter:

1. The three core elements of the Docker CNM are as follows:
   - **Sandbox**: A network namespace for a container where its network stack resides
   - **Endpoint**: An interface that connects a container to a network
   - **Network**: A collection of endpoints that can communicate with each other directly
2. To create a custom Docker `bridge` network called `frontend`, you can use the `docker network create` command with the `--driver` flag set to `bridge` (which is the default driver) and the `--subnet` flag to specify the subnet for the network. Here's an example command:

   ```
   $ docker network create --driver bridge \
       --subnet 172.25.0.0/16 frontend
   ```

   This will create a bridge network named `frontend` with a subnet of `172.25.0.0/16`. You can then use this network when starting containers with the `--network` option:

   ```
   $ docker run --network frontend <docker-image>
   ```

3. To run two `nginx:alpine` containers attached to the `frontend` network that we created earlier, you can use the following `docker run` commands:

   ```
   $ docker run --name nginx1 --network frontend -d nginx:alpine
   $ docker run --name nginx2 --network frontend -d nginx:alpine
   ```

   These commands will start two containers named `nginx1` and `nginx2` with the `nginx:alpine` image and attach them to the `frontend` network. The `-d` flag runs the containers in the background as daemons. You can then access the containers by their container names (`nginx1` and `nginx2`) or their IP addresses within the `frontend` network.
4. Here is the solution:

   A. To get the IPs of all the containers attached to the `frontend` Docker network, you can use the `docker network inspect` command, followed by the network name. Here's an example command:

   ```
   $ docker network inspect frontend --format='{{range
   .Containers}}{{.IPv4Address}} {{end}}'
   ```

   B. This will output the IPv4 addresses of all the containers attached to the frontend network, separated by spaces.

C. To get the subnet associated with the `frontend` network, you can again use the `docker network inspect` command followed by the network name. Here's an example command:

```
$ docker network inspect frontend --format='{{json .IPAM.
Config}}' | jq -r '.[].Subnet'
```

D. This will output the subnet associated with the `frontend` network in CIDR notation (e.g., `172.25.0.0/16`). The `jq` command is used here to parse the output of the `docker network inspect` command and extract the subnet.

5. The Docker `host` network is a networking mode that allows a Docker container to use the host's networking stack instead of creating a separate network namespace. In other words, containers running in `host` network mode can directly access the network interfaces and ports of the Docker host machine.

   The purpose of using the `host` network mode is to improve network performance since it avoids the overhead of containerization and network virtualization. This mode is often used for applications that require low-latency network communication or need to listen on a large number of ports.

   However, using the `host` network mode can also present security risks since it exposes the container's services directly on the Docker host's network interfaces, potentially making them accessible to other containers or hosts on the same network.

6. The Docker `host` network mode is appropriate for scenarios where network performance is critical and where network isolation is not a requirement. For example, see the following:

   - In cases where the containerized application needs to communicate with other services running on the host machine, such as a database or cache service, the use of the `host` network mode can improve performance by eliminating the need for **network address translation** (**NAT**) and routing between containers.
   - In scenarios where the containerized application needs to listen on a large number of ports, the `host` network mode can simplify network configuration and management by allowing the container to use the same network interfaces and IP addresses as the host machine, without the need to manage port mapping between the container and host network namespaces.

7. The purpose of the Docker `none` network is to completely disable networking for a container. When a container is started with the `none` network mode, it does not have any network interfaces or access to the network stack of the host machine. This means that the container cannot communicate with the outside world or any other container.

   The `none` network mode is useful in scenarios where the container does not require network connectivity, such as when running a batch process or a single-use container that performs a specific task and then exits. It can also be used for security purposes, to isolate the container from the network and prevent any potential network-based attacks.

It's important to note that when a container is started with the `none` network mode, it can still access its own filesystem and any volumes that are mounted to it. However, if the container requires network access later on, it must be stopped and restarted with a different network mode.

8. The Docker `none` network mode is useful in scenarios where the container does not require network connectivity, such as the following:

   - Running a batch process or a single-use container that performs a specific task and then exits
   - Running a container that does not need to communicate with other containers or the host machine
   - Running a container that has no need for external network access, such as a container that is used only for testing or debugging
   - Running a container that requires high security and isolation from the network

9. There are several reasons why we might use a reverse proxy such as Traefik together with our containerized application:

   - **Load balancing**: A reverse proxy can distribute incoming traffic across multiple instances of our application running on different containers, ensuring that no single instance becomes overwhelmed with requests.
   - **Routing**: With a reverse proxy, we can route incoming requests to the appropriate container based on the URL or domain name. This allows us to run multiple applications on the same host, each with its own unique domain or URL.
   - **SSL/TLS termination**: A reverse proxy can terminate SSL/TLS connections and handle certificate management, eliminating the need for our application to do this itself. This can simplify our application code and reduce the risk of security vulnerabilities.
   - **Security**: A reverse proxy can act as a buffer between our application and the public internet, providing an additional layer of security. For example, it can block certain types of traffic or filter out malicious requests.
   - **Scalability**: By using a reverse proxy such as Traefik, we can quickly and easily scale our application up or down by adding or removing containers. The reverse proxy can automatically route traffic to the appropriate containers, making it easy to manage our application's infrastructure.

11
Managing Containers with Docker Compose

In the previous chapter, we learned a lot about how container networking works on a single Docker host. We introduced the **Container Network Model** (**CNM**), which forms the basis of all networking between Docker containers, and then we dove deep into different implementations of the CNM, specifically the bridge network. Finally, we introduced Traefik, a reverse proxy to enable sophisticated HTTP application-level routing between containers.

This chapter introduces the concept of an application consisting of multiple services, each running in a container, and how Docker Compose allows us to easily build, run, and scale such an application using a declarative approach.

This chapter covers the following topics:

- Demystifying declarative versus imperative orchestration of containers
- Running a multi-service application
- Building images with Docker Compose
- Running an application with Docker Compose
- Scaling a service
- Building and pushing an application
- Using Docker Compose overrides

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

- Explain, in a few short sentences, the main differences between an imperative and declarative approach for defining and running an application
- Describe, in your own words, the difference between a container and a Docker Compose service

- Author a Docker Compose YAML file for a simple multi-service application
- Build, push, deploy, and tear down a simple multi-service application using Docker Compose
- Use Docker Compose to scale an application service up and down
- Define environment-specific Docker Compose files using overrides

Technical requirements

The code accompanying this chapter can be found at `https://github.com/PacktPublishing/The-Ultimate-Docker-Container-Book/tree/main/sample-solutions/ch11`.

Before we start, let's make sure we have a folder ready for the code you are going to implement in this chapter:

1. Navigate to the folder in which you cloned the previously listed code repository accompanying this book. Normally, this is the `The-Ultimate-Docker-Container-Book` folder in your home folder:

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

2. Create a subfolder called `ch11` and navigate to it:

```
$ mkdir ch11 && cd ch11
```

In the past, you needed to have a separate docker-compose tool installed on your system. This is not the case anymore as the Docker CLI has recently been extended such that it contains all the functionality and more than the docker-compose tool previously offered.

If you are curious, you can find detailed installation instructions for the old `docker-compose` tool here: `https://docs.docker.com/compose/install/`.

Demystifying declarative versus imperative orchestration of containers

Docker Compose is a tool provided by Docker that is mainly used when you need to run and orchestrate containers running on a single Docker host. This includes, but is not limited to, development, **continuous integration** (**CI**), automated testing, manual QA, or demos. Since very recently, Docker Compose is embedded in the normal Docker CLI.

Docker Compose uses files formatted in YAML as input. By default, Docker Compose expects these files to be called `docker-compose.yml`, but other names are possible. The content of a `docker-compose.yml` file is said to be a declarative way of describing and running a containerized application potentially consisting of more than a single container.

So, what is the meaning of declarative?

First of all, declarative is the antonym of imperative. Well, that doesn't help much. Now that I have introduced another definition, I need to explain both:

- **Imperative**: This is a way in which we can solve problems by specifying the exact procedure that has to be followed by the system.

  If I tell a system, such as the Docker daemon, imperatively how to run an application, then that means that I must describe, step by step, what the system has to do and how it must react if some unexpected situation occurs. I must be very explicit and precise in my instructions. I need to cover all edge cases and how they need to be treated.

- **Declarative**: This is a way in which we can solve problems without requiring the programmer to specify an exact procedure to be followed.

  A declarative approach means that I tell the Docker engine what my desired state for an application is and it has to figure out on its own how to achieve this desired state and how to reconcile it if the system deviates from it.

Docker clearly recommends the declarative approach when dealing with containerized applications. Consequently, the Docker Compose tool uses this approach.

Running a multi-service app

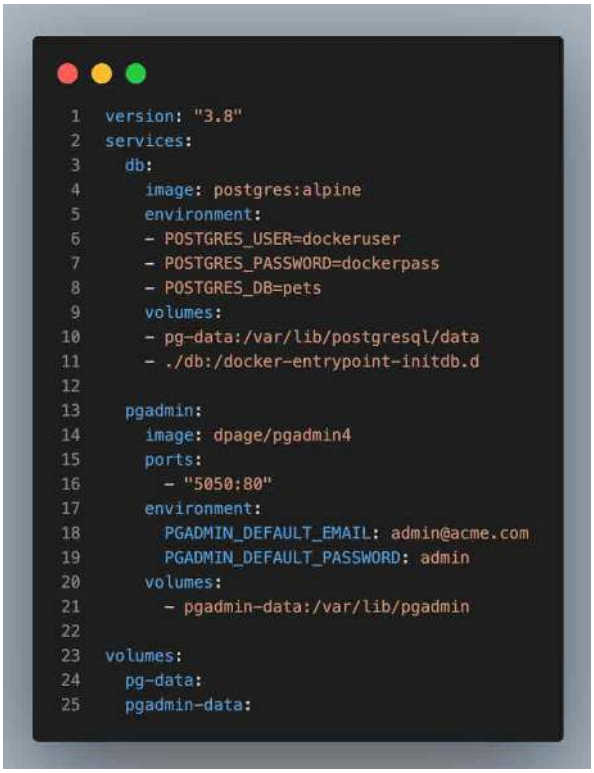
In most cases, applications do not consist of only one monolithic block, but rather of several application services that work together. When using Docker containers, each application service runs in its own container. When we want to run such a multi-service application, we can, of course, start all the participating containers with the well-known `docker container run` command, and we have done this in previous chapters. But this is inefficient at best. With the Docker Compose tool, we are given a way to define the application in a declarative way in a file that uses the YAML format.

Let's create and analyze a simple `docker-compose.yml` file:

1. Inside the chapter's folder (`ch11`), create a subfolder called `step1` and navigate to it:

   ```
   $ mkdir step1 && cd step1
   ```

2. Inside this folder, add a file called `docker-compose.yml` and add the following snippet to it:



```
version: "3.8"
services:
  db:
    image: postgres:alpine
    environment:
    - POSTGRES_USER=dockeruser
    - POSTGRES_PASSWORD=dockerpass
    - POSTGRES_DB=pets
    volumes:
    - pg-data:/var/lib/postgresql/data
    - ./db:/docker-entrypoint-initdb.d

  pgadmin:
    image: dpage/pgadmin4
    ports:
      - "5050:80"
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@acme.com
      PGADMIN_DEFAULT_PASSWORD: admin
    volumes:
      - pgadmin-data:/var/lib/pgadmin

volumes:
  pg-data:
  pgadmin-data:
```

Figure 11.1 – Simple Docker Compose file

The lines in the file are explained as follows:

- Line 1: `version` – On this line, we specify the version of the Docker Compose format we want to use. At the time of writing, this is version `3.8`.
- Lines 2–21: `services` – In this section, we specify the services that make up our application in the `services` block. In our sample, we have two application services, and we call them `db` and `pgadmin`.
- Lines 3–11: `db` – The `db` service is using the image name `postgres:alpine`, which is the latest version of the Alpine Linux-based PostgreSQL database:
  - Line 4: `image` – Here, we define which Docker image to use for the service. As mentioned previously, we're using the curated `postgres` image with a tag of `alpine`. Since we're not specifying a version number, it will take the latest stable version of the Alpine-based PostgreSQL image.

- Lines 5–8: `environment` – Here, we are defining the environment variables that will be accessible from within the running PostgreSQL service. In this case, we define the default username, password, and database name.
- Lines 9–11: `volumes` – We are defining two volume mappings.
- Line 10: We are mapping a volume called `pg-data` to the `/var/lib/postgresql/data` container folder. This is where PostgreSQL by default stores the data. This way, the data is persisted into the `pg-data` volume and will survive a restart of the `db` service.
- Line 11: In this case, we are mapping the host folder, `./db`, into a container folder called `/docker-entrypoint-initdb.d`. This is the folder where PostgreSQL expects any initialization files that are run upon the first start of the database. In our case, we'll use it to define a database initialization script called `init-db.sql`.

- Lines 13–21: `pgadmin` – The `pgadmin` service uses a Docker image containing the popular administration tool for PostgreSQL and similar databases called `Pg4Admin`. We are mounting a volume called `pgadmin-data` into the container of the `db` service:
  - Line 14: `image` – This service is using the `dpage/pgadmin4` image. Note we're not defining any tags for the image, so we'll automatically work with the latest version.
  - Lines 15–16: `ports` – Here, we define which container ports we want to map to the host. In this case, we map the default `Pg4Admin` port `80` to the host port `5050`. This way, we can access the admin tool on this latter port from a browser window, as we will see shortly.
  - Lines 17–19: `environment` – Here, we are defining the environment variables that will be accessible from within the running `Pg4Admin` tool container. It is the email and password we will need to log in to the tool.
  - Lines 20–21: `volumes` – We are mapping a Docker volume called `pgadmin-data` to the `/var/lib/pgadmin` folder inside the container. This is the place where the tool stores its data and makes it possible to survive a restart of the tool container.
- Lines 23–25: `volumes` – The volumes used by any of the services must be declared in this section. In our sample, this is the last section of the file. The first time the application is run, volumes called `pg-data` and `pgadmin-data` will be created by Docker and then, in subsequent runs, if the volumes are still there, they will be reused. This could be important if the application, for some reason, crashes and must be restarted. Then, the previous data is still around and ready to be used by the restarted database service.

3. Create a folder called `db` in the `step1` folder and add a file called `init-db.sql` to it with the following content:

```
CREATE TABLE images
(
  imageid serial UNIQUE PRIMARY KEY,
  description character varying(10485760) NOT NULL,
  url character varying(255) NOT NULL
);

ALTER TABLE images
  OWNER TO dockeruser;
ALTER ROLE dockeruser CONNECTION LIMIT -1;

-- add image data
INSERT INTO images (description, url) VALUES('vulture on tree', 'images/vulture.png');
INSERT INTO images (description, url) VALUES('female lions', 'images/3-female-lions.png');
INSERT INTO images (description, url) VALUES('antelopes', 'images/antelopes.png');
INSERT INTO images (description, url) VALUES('birds', 'images/birds.png');
INSERT INTO images (description, url) VALUES('buffalo', 'images/buffalo.png');
INSERT INTO images (description, url) VALUES('cheetah', 'images/cheetah.png');
INSERT INTO images (description, url) VALUES('elephants', 'images/elephants.png');
INSERT INTO images (description, url) VALUES('jackal', 'images/jackal.png');
INSERT INTO images (description, url) VALUES('giraffe', 'images/giraffe.png');
INSERT INTO images (description, url) VALUES('hippos', 'images/hippos.png');
INSERT INTO images (description, url) VALUES('male lion', 'images/male-lion.png');
INSERT INTO images (description, url) VALUES('zebra', 'images/zebra.png');
```

Figure 11.2 – Database initialization script

If you don't want to type in all of the preceding, you can find the file here: `https://github.com/PacktPublishing/The-Ultimate-Docker-Container-Book/blob/main/sample-solutions/ch11/step1/db/init-db.sql`. As you will see later, this file will be used to initialize our database with some initial schema and some data.

4. Next, let's see how we can run the services with the help of Docker Compose. Execute the following command from within your `step1` folder where the `docker-compose.yml` file resides:

```
$ docker compose up
```

Let's analyze the output generated by the preceding command:

- The first few lines are telling us that Docker is pulling the images for the `db` and `pgadmin` services
- The next few lines indicate that Docker is automatically creating a new network called `step1_default`

- Two volumes called `step1_pgadmin-data` and `step1-pg-data`
- Two container instances called `step1-db-1` and `step1-pgadmin-1`

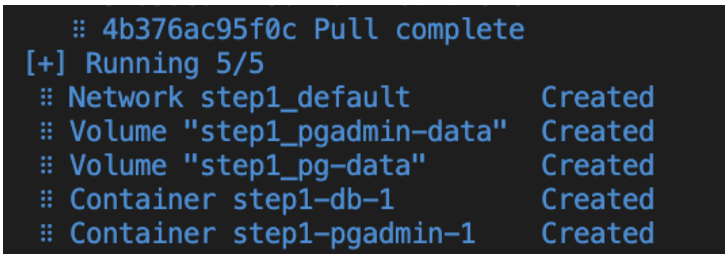


Figure 11.3 – Creating the resources for the Docker Compose application

Note the `step1_` prefix added to all the preceding resources. This is the folder name within which the `docker-compose.yml` exists and from where the app was started, combined with the underscore character.

- Now, let's look at the third part of the output in blue. Here, the database is started up:

```
Attaching to step1-db-1, step1-pgadmin-1
step1-db-1       | The files belonging to this database system will be owned by user "postgres".
step1-db-1       | This user must also own the server process.
step1-db-1       |
step1-db-1       | The database cluster will be initialized with locale "en_US.utf8".
step1-db-1       | The default database encoding has accordingly been set to "UTF8".
step1-db-1       | The default text search configuration will be set to "english".
step1-db-1       |
step1-db-1       | Data page checksums are disabled.
step1-db-1       |
step1-db-1       | fixing permissions on existing directory /var/lib/postgresql/data ... ok
step1-db-1       | creating subdirectories ... ok
step1-db-1       | selecting dynamic shared memory implementation ... posix
step1-db-1       | selecting default max_connections ... 100
step1-db-1       | selecting default shared_buffers ... 128MB
step1-db-1       | selecting default time zone ... UTC
step1-db-1       | creating configuration files ... ok
step1-db-1       | running bootstrap script ... ok
step1-db-1       | sh: locale: not found
step1-db-1       | 2023-03-19 10:20:36.847 UTC [30] WARNING:  no usable system locales were found
step1-db-1       | performing post-bootstrap initialization ... ok
step1-db-1       | initdb: warning: enabling "trust" authentication for local connections
```

Figure 11.4 – Starting up the database