Docker : Up and Running

Build and deploy containerized web apps with Docker and Kubernetes

Dr. Gabriel Nicolas Schenker

www.bpbonline.com

First published: 2023

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55517-883

www.bpbonline.com

Dedicated to

My beloved wife ***Veronicah***

About the Author

Dr. Gabriel Nicolas Schenker has more than 30 years of experience as an independent consultant, architect, leader, trainer, mentor, and developer. Currently, Gabriel works as the lead architect in the office of the CTO at iptiQ by SwissRE. Prior to that, he held the position of lead solutions architect at Maison, of lead curriculum developer at Confluent, and of principal curriculum developer at Docker. Gabriel has a Ph.D. in Physics, and he was a Docker Captain. He is a Certified Docker Associate, a certified Apache Kafka Operator, a certified Apache Kafka Developer, and an ASP Insider. When not working, Gabriel enjoys time with his wonderful wife Veronicah and his children.

About the Reviewer

Shilpa Kaul has 15+ years of extensive experience with Enterprise Architecture, Analysis, Design, Development and Cloud Migration. Currently working as Application Modernization Consultant - HCL Google Cloud Ecosystem supporting the enterprise client journey to cloud by analyzing their application portfolios and translating them into actionable entities. Skilled in development of tools/accelerators using Python/data analytics. Experience also includes working on Docker, Kubernetes and cloud orchestration tools like Canonical Juju Framework. AWS Certified Practitioner and Google Cloud Digital Leader certified. Passionate about learning new technologies especially in areas of containerization, DevOps, python and ML. In her free time she likes to spend time listening to music, travelling and reading.

Acknowledgement

I would like to express my deepest gratitude to my wife, Veronicah, who has been my rock and my source of strength throughout this journey. Your unwavering love and support have been invaluable to me, and I could not have accomplished this without you. Your encouragement and belief in me have inspired me to keep pushing forward, even on the most challenging days. Thank you for being my constant source of love and comfort. This book is as much yours as it is mine, and I am forever grateful for your partnership. I love you.

My gratitude also goes to the team at BPB for being supportive throughout this process, which took longer than I initially expected.

Preface

This book introduces the readers to Docker, practicing the readers in writing the Docker commands and how to create Docker files, images, creating containers. While doing so, you get a stronghold on Docker tools like Docker Images, Dockerfiles, and Docker Compose. The book explains the architecture of Docker, designing multi-containers, understanding how to automate the work of containerization, working with other tools such as Kubernetes and Jenkins, securing Docker containers, and more.

This book is dedicated to developers, DevOps, QA automation, and operations engineers interested in Docker containers and Kubernetes. No prior knowledge of containers and container orchestration is needed as the book introduces the concepts from the ground up. It is though highly recommended that the reader has some basic knowledge of coding or scripting in one of the popular languages such as JavaScript, Python, Java, .NET C#, PowerShell, or Bash.

By the end of this book, the reader will become a successful developer or engineer who can convert their complex stacks of applications into a single file and run it with a single command, thereby simplifying the entire process of shipping and deploying enterprise applications without worrying about their infrastructure.

The book is divided into **13 chapters**. They will start from the basics of Docker containers, progressing slowly to more elaborate topics such as building, testing, shipping, and running individual containers and whole containerized applications in Kubernetes on-premise or in the cloud.

Chapter 1: Explaining Containers and their Benefits – informs the reader about what containers are and why they are crucial in modern software development. We will start with a brief history of the evolution of containers and their surrounding ecosystem. We will mention examples and present some case studies on where the use of containers has brought significant benefits to the development, distribution, and or maintenance of applications. We continue the chapter with an overview of where containers are a good fit and where other technologies or approaches may be appropriate instead. We will show how the use of containers and related technologies brought a shift in the demand for skills and resulted in new job roles. We will present some quantitative information that may convince readers, such as

salary info, job stability, and career progression. We conclude the chapter with a short recap of what the reader has learned.

Chapter 2: Setting Up Your Environment - will guide the reader step by step on how to set up a great working environment for the development with and the use of containers. We will provide easy-to-follow and step by step instructions for both platforms, Windows 10 and Mac OS, on how to install and configure Docker Desktop, Docker Compose, a terminal, and the Visual Studio Code code editor.

Chapter 3: Getting Familiar with Containers – familiarizes the reader with how to download a first Docker image from the Docker Hub and run a container from this image. They will get to know the most important characteristics or attributes of a container, such as its name and status. Furthermore, they will learn how to stop and remove containers as well as how to visualize what's really running inside a container. Finally, the reader will list and analyze the log of a running container and run their own processes in the context of a running container. In this chapter, we will mainly use Docker Desktop and the terminal.

Chapter 4: Using Existing Docker Images – teaches the reader how to use existing container images, such as ones for SQL databases locally on their computer. They will learn how an image is made up of layers, and they learn to define and use Docker volumes to persist and share data and how to configure applications running inside a container. Tools used in this chapter are Docker Desktop and the terminal.

Chapter 5: Creating Your Own Docker Image –leads the reader through creating their own Docker images and shows how to tag and publish them to a container registry such as Docker Hub. To author Dockerfiles, we will use VS Code, and for all other operations Docker for the Desktop and the terminal.

Chapter 6: Demystifying Container Networking – introduces 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 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. Lastly, the reader is introduced to network types that can span whole clusters of multiple nodes. We conclude with some tips on how to excel in a job interview when container network-related questions are asked, and we provide a short recap of the key learnings. In this chapter, we will mainly use the terminal and some simple Linux tools

Chapter 7: Managing Complex Apps with Docker Compose – 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. Tools used in this chapter are Docker Desktop, the terminal, and VS Code to author the various Docker Compose files

Chapter 8: Testing and Debugging Containerized Applications – elaborates on how to significantly reduce friction in the development process by enabling hot reloading and debugging of an application running inside a container. The reader will also learn how to write and run automated tests for such an app and how to instrument the app optimally. The reader will get some valuable tips and hints on how to best prepare for an interview as a Docker developer. In this chapter, we mainly use our code editor VS code as well as its integrated terminal.

Chapter 9: Establishing an Automated Build Pipeline – explains why no SW development team can be successful without building up and/or leveraging a fully automated CI/CD pipeline. In this chapter, the reader learns how to build and use the continuous implementation (CI) part of such a pipeline. Later, we will discuss and compare some free and some commercial SaaS offerings and implement a complete CI/CD pipeline using GitHub actions as an example. We then provide some tips on how to prepare for a job interview for DevOps engineer covering mainly the CI part. We conclude the chapter with a recap of the key learnings. The tools used will be the terminal and VS Code.

Chapter 10: Orchestrating Containers– introduces container orchestration in general and Kubernetes in particular. Kubernetes is currently the clear leader in the container orchestration space. We will start with a high-level overview of the architecture of a Kubernetes cluster and then discuss the main objects used in Kubernetes to define and run containerized applications. Since many teams do not want to manage their own cluster, we provide a brief overview of the most popular fully managed Kubernetes offerings in the cloud. Finally, we discuss the position of the container in the new context of so-called serverless computing. Lastly, we will train the reader for a job interview where container orchestration-related questions will be asked. In this chapter, we mainly use Docker Desktop with its integrated Kubernetes, the terminal, and VS Code to author YAML files.

Chapter 11: Leveraging Docker Logs to Provide Insight into Your Apps – shows that once your application is running in production, no outage is tolerated. To prevent costly outages or readily fix them if they happen, you need insight

into the inner workings of you app. In this chapter, we will discuss how to collect Docker and application logging information and use it to monitor your application, how to get notified in the event of a critical event, and how to quickly find the root cause in the event of a bug or an outage. We will be using Prometheus to collect logs and Grafana to visualize key metrics.

Chapter 12: Enabling Zero Downtime Deployments - exemplifies how to deploy, update, and scale applications into a Kubernetes cluster using different strategies such as rolling updates or blue-green deployments. We will also explain how zero downtime deployments are achieved to enable disruption-free updates and rollbacks of mission-critical applications. To conclude this chapter, the reader is provided with more tips and hints on how to excel in a job interview for a DevOps position and a discussion of the key learnings.

Chapter 13: Securing Containers - introduce Kubernetes secrets to configure services and protect sensitive data. The reader will learn how to provide trust in container images and how to scan images to proactively identify and eliminate vulnerabilities that bad actors could exploit. Finally, we will discuss the use of software-defined networks, port, and protocol mapping as well as ingress and egress rules to further firewall their precious applications.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/gavpian

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Docker-Up-and-Running**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

Chapter 1
Explaining Containers and their Benefits

Introduction

In this chapter, you will learn what containers are and why they are so important in modern software development. We will start with a brief history of the evolution of containers and their surrounding ecosystem. We will mention examples and present some case studies on where the use of containers has brought significant benefits to the development, distribution, and/or maintenance of applications. We will show how the use of containers and related technologies brought a shift in the demand for skills and resulted in new job roles. We will present some quantitative information that may convince readers, such as salary info, job stability, and career progression. We conclude the chapter with a short recap about what the reader has learned so far.

Structure

In this chapter, we will discuss the following topics:

- What is a container
- Evolution of containers and associated infrastructure
- Architecture and core capabilities
- Main use cases and case studies
- Skills, jobs, salaries, and career paths

Let us start by describing what a container is.

Objectives

In this chapter, the reader will learn what containers are by giving a very familiar analogy and the most important use cases of containers. We will also learn ways in which containers can help us to significantly reduce the friction in the software development life cycle and, on top of that, how containers can make the software supply chain more secure and resilient against cyber threats.

What is a container?

In this section, we want to provide you with a high-level description of what a Docker container is. A container is a process running on an operating system, such as Linux, which is protected by namespaces and CGroups. This is a very simplified picture, but it helps create a mental model in our heads. We will discuss namespaces and CGroups in more detail later in this chapter. Contrary to common belief, a Docker container is not some lightweight **virtual machine** (**VM**), although, in many regard, a VM and a container look and behave similarly. Whilst a VM is a whole virtual computer including a full operating system and all its drivers and interfaces, a container is only containing the so-called user space with its application. A container does never contain the Kernel of an operating system. Rather the container or the set of containers uses the Kernel of their host computer:

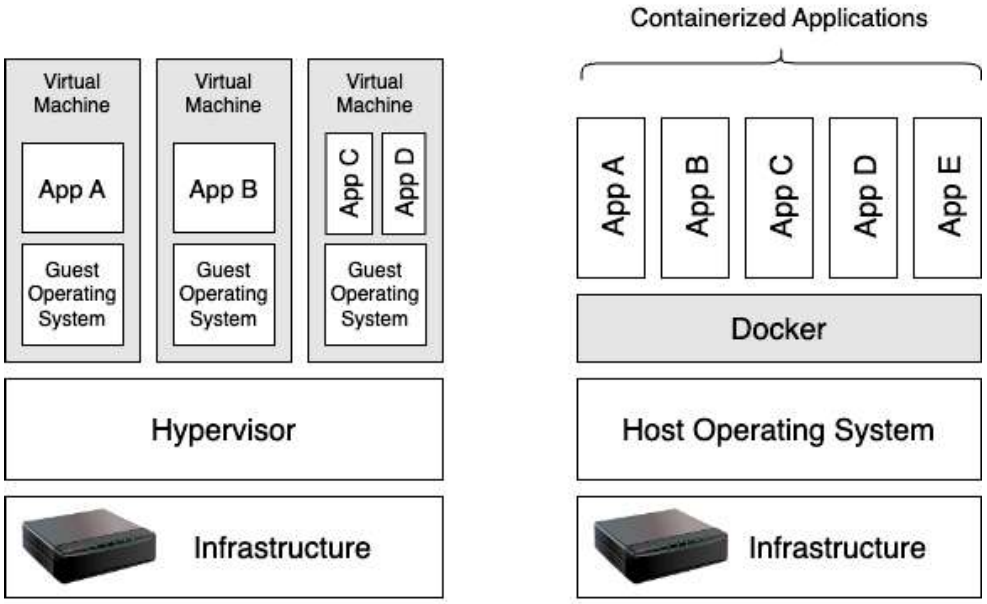

Figure 1.1: *Virtual machines (VMs) versus Docker containers*

Evolution of containers and associated infrastructure

This section provides a short history of the evolution of containers and the associated infrastructure. Docker did not invent containers. Docker made containers popular by making them easy to use. It is similar to what Apple did with the iPhone. The iPhone was not the first smartphone, but Apple brought the idea of smartphones to the masses by making them easy and intuitive to use. Docker did a similar job with containers.

To better understand software containers, let us look around us and come up with a good analogy. In this case, we take physical containers as an example. You know, those big iron boxes that have a well-defined length, width, and height. They are used widely to ship goods around the globe. Certainly, you have seen a container ship transporting merchandise from China to Europe or the USA live or in the media. But traditionally, before those shipping containers existed, people had to transport goods by different means. Prior to the invention of the wheel, goods were transported in sacks or baskets on the shoulders of people or on the back of specially trained animals such as donkeys.

The invention of wheels and later engines made transportation of merchandise a bit more efficient since with a cart, a train, a steamboat, or a truck, we were able to transport much bigger quantities at a time over much longer distances. But at the same time, we started to transport more types of goods. At times the handling of those goods also became more complicated. What did not change through time, though, was the need to unload goods from one means of transportation onto another one. Let us take an example, wheat that is produced in Ukraine and then shipped to Africa, where it is used to bake bread. Farmers in Ukraine grow wheat, and when it is ready to harvest, they transport the grains with their tractors to the next village and unload it into a grain storage tank. Trucks will be loaded with the grain and ship it to the next port, most probably to Odessa. There they unload the wheat into big ships that will then carry massive amounts of grains to the destination port in Africa. From there again, trucks will pick up loads of those grains and transport them to the mill, and so forth. You will agree that during the many unload and load processes that are time-consuming and tedious, a lot can go wrong. Some waste is produced, or part of the goods can be stolen by corrupt workers, to just name a few issues.

The introduction of containers, those big and standardized iron boxes we mentioned previously, revolutionized the shipping industry. Since the dimensions of a shipping container are standardized, the whole ecosystem around it could start standardizing as well. Truck, train, and ship makers could start to produce vehicles that are doing nothing but transport one or many containers from location A to location B. Shippers could start to buy those standardized vehicles and then standardize the process of

shipping around them. Ports and other hubs could start using standardized tools optimized for handling containers such as cranes or forklifts.

Now, having said that, we can easily see that the unloading and loading process has become much simplified. There is no need to open a container anymore during this process. In fact, containers are often sealed to make sure that they have not been opened in transit. With this, there is also less danger that part of the load gets lost or wasted in the process of changing the means of transportation. The shipper, in most cases, does not care what exactly is inside a container and how it has been packaged, to avoid any damage during transportation. For the latter, the producer of the goods is responsible. They best understand how their goods need to be handled to avoid any damage and, as such, are best qualified to fill the container in the first place. Once arrived at the final destination, the recipients of the goods are, in turn, most qualified to unload the container and handle the goods the right way. Shippers, in the meantime, only care about a limited number of things, namely, the place from where to pick up the container, its weight, and the destination to where they must transport it. They may also care about if the container needs to be connected to a power source; for example, if the goods in the container need to be cooled, then those containers will need power for the cooling, yet the shipper is not responsible for the cooling, only for the power needed by the cooler in the container.

This analogy will help give you a clear picture of why the introduction of shipping containers has been such a paradigm change in the industry. So much has improved with this new invention. Software containers represent the exact same paradigm change in the software industry. Prior to having software containers, the **Software Development Life Cycle** (**SDLC**) was similarly complex and non-standardized as the shipping of goods was. But the introduction of software containers streamlined all tools and processes around the building, securing, testing, shipping, and running applications.

Architecture and core capabilities

Now, we are ready for a high-level description of the architecture and the core capabilities of Docker. Let us present a high-level architecture diagram of a system able to run Docker containers. The following illustration shows how a host system on which Docker has been installed looks like. Any computer or server that has the Docker daemon installed is sometimes called a Docker host.

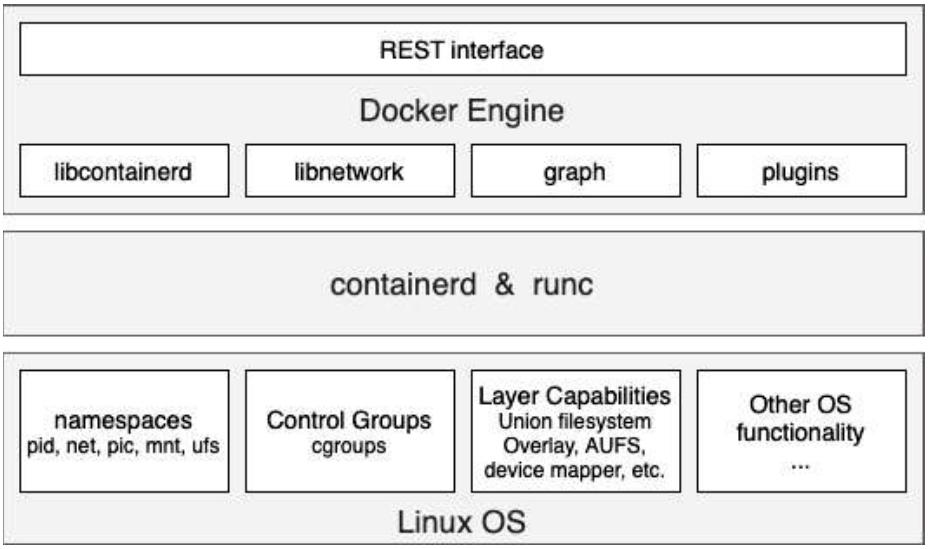

***Figure 1.2**: Docker high-level architecture*

In the preceding diagram, we can see three main elements, all with grey backgrounds:

- At the bottom, we have the Linux operating system
- In the middle, we have the container runtime
- Finally, the Docker engine sits at the top

The enabling factors for containers are low-level Linux—or, more specifically, Unix—features such as namespaces, control groups (cgroups), layer capabilities, and so on. All those are building blocks and are used by the container runtime and the Docker engine. With the help of Linux namespaces, Docker can encapsulate applications that run inside a container. Note that any application is made up of one to several processes. All these processes will run in the same namespace defined by the enclosing container. Examples of namespaces are the process namespace or the network namespace. cgroups can be used by containers to limit how much of the available resources the processes inside the container can use. Resources are CPU time, the amount of memory or disk, the available network bandwidth, and more. By having this throttling mechanism in place, the system administrator can avoid situations in which a single malicious container tries to monopolize all available resources of the host system and, as such, starve all the other applications running in different containers on the same host.

To be able to run containers on a host, normally, we need some low-level libraries. In the case of Docker, these are **`containerd`** and **`runc`**. **`runc`** is a low-level library providing very basic container runtime support. **`containerd`**, in turn, builds on top of **`runc`** and provides more higher-level support for containers and container

images. Originally both have been part of the bigger Docker engine daemon but have later been broken out of this monolithic structure and modularized. Docker did donate `containerd` and `runc` to the **Cloud Native Computing Foundation** (**CNCF**). You may know this organization from other important software it owns, such as Kubernetes.

The container runtime, made up of `runc` and `containerd` is managing the life cycle of containers. It enables the pulling of container images from a central image registry, instantiates a container from that image, and initializes and runs the container. Furthermore, the runtime enables us to stop and restart or remove containers from the system.

The Docker engine augments the container runtime by providing additional features such as some networking libraries, the support for plugins used to extend the engine, and finally, a REST interface through which all container-related operations can be automated. In this book, we will frequently use the **Docker Command Line Interface** (**Docker CLI**). This tool is one of the main consumers of the Docker engine's REST API.

Main use cases

In this section, we want to present a list of the main use cases for Docker containers. In the following, we want to describe some of these use cases in more detail.

Prior to the advent of containers, developers would develop a new applications. Once the application was completed by the definition of the developers, they handed it over to the **Quality Engineering** (**QE**) department. The quality engineers then tested the application according to the specifications. If everything was OK, then QE handed the application over to the **IT operations** (**IT OPS**) department. The IT OPS engineers then tried to install and run the application on servers they had configured before. Ideally, and if they were lucky, the developers had provided some kind of—hopefully—accurate documentation on how to install and configure the application. The situation was not ideal but manageable. It got a bit out of control, though, when in an enterprise, many teams developed many different applications. If those applications needed to be installed and run on the same set of production servers to optimize resource usage, things got quickly quite complex. Each application normally has some dependencies on libraries and frameworks. Unfortunately, not all applications use the same libraries and frameworks, or even worse, they use the same libraries but in different and sometimes incompatible versions. Think of two applications written in Java. One uses Java 8, and the other, a more modern one, uses Java 17. The IT OPS service providers had to really have their act together and be creative in how they could put all those different requirements under a single hat, that is, run the applications on the same servers without breaking anything.

You may understand that in such a situation, upgrading applications introduces major risks. A new release needed careful planning and testing that could easily take months and bind a lot of engineering and management resources. This all meant a lot of friction in the software supply chain. But nowadays, companies are not happy with new releases every few months or so. The requirement is to deploy new features in a short time, ideally daily. Companies that do not follow the path of continuous delivery will struggle or even go out of business. There are enough competitors out there that are waiting to pick up where your company failed to deliver. What can we do?

Standardize infrastructure

The first step was to **Introduce Virtual Machines (VMs)**. This way, one could easily divide the resources of a physical server among several VMs running on this very same machine. Then OPS engineers would deploy one application per VM, and thus, solve the problem quickly. Each VM is like a complete and impermeable sandbox, and thus, everything inside of one VM is shielded from all the other applications running in the other VMs.

Sadly, this solution fell apart quite quickly with the advent of the microservice architecture. Now, a single application suddenly was made up of many pieces, the microservices. In theory, every microservice could be written in a different language using different frameworks and libraries. With that, we are back to square one. Some clever engineers wanted to solve the problem by only packing a single microservice into a VM. But this is a huge waste. VMs are quite heavyweight. They contain a full operating system such as Linux or Windows Server. Now imagine all this for just a single microservice. It looks like we are using a whole container ship to transport a single truckload of wheat.

The solution to this maze came with the introduction of what looks a lot like an extremely lightweight VM, the Docker container. It is not a VM, though, mind you, as it does not contain a whole operating system and also does not provide the same level of isolation as a VM. At the same time, it provides enough isolation for an application or an application service packaged into it. It encapsulates the service and its dependencies in the form of libraries and frameworks. For some, the Docker container looks like the holy grail of modern software development.

Like the introduction of physical containers in logistics—you know, those standardized metal boxes used to ship goods—has revolutionized the shipping industry. Docker containers have similarly helped to revolutionize the IT industry. While the introduction of virtualization in the form of VMs has made the cloud possible in the first place, containers represent the next paradigm shift in the industry. By standardizing how we ship and run the software, we have given a whole new industry of infrastructure and service providers a chance to thrive. Operating systems

have been created that are optimized to host containers. Container orchestration engines such as Kubernetes or Docker Swarm have been created and popularized. Many tools focussing on containers have been built. A whole ecosystem of software is centered around containers. Most prominently, this can be seen by browsing the project page of the CNCF, which you can reach via the following URL: **https://www.cncf.io/projects/**.

Developers package their application services into containers and, on top of that, add supporting libraries and frameworks to the package. Then they hand over those self-contained containers to QA engineers that test them or operation engineers that will run them. For QA engineers and operation engineers alike, a container, in this sense, is just a black box with some well-defined interfaces. This black box is coming in a standard form, though, and thus, the engineers can treat them all equally. This is true for all types of applications shipped in a container. If an operations engineer has a container host at hand that can run a container, then most likely, it can run any other container too. This is not a false promise but true for almost all cases, except some special edge cases.

We can thus look at containers as a way of packaging applications or application services, all their dependencies, and supporting frameworks and libraries in a standardized way. Not surprisingly, Docker came up with the following slogan for containers: "*Build, ship, and run anywhere*."

Turbocharge a microservice architecture

As applications have grown in complexity and the range of features they cover, developers and software architects have been looking for a means to break the monolithic structure of an application apart and modularize it. If too many developers were working on the same monolithic application at the same time, there was a real danger that they did step on each other's feet. Deploying code changes became more and more difficult due to the fact that the work of all developers had to be coordinated.

The solution was to break down the big and complex applications into smaller, less complex pieces that ideally were somewhat loosely coupled with each other. This way, one could assign such a piece or module of software to a team consisting of a small group of people. This team could then work independently from all the other teams and evolve their module. Again, this was possible due to the fact that their module was loosely coupled to the other modules and interacted with them through well-defined interfaces. This was the birth of the so-called micro-service architecture.

In this new architecture, such a software module is now called a microservice. As we mentioned previously, microservices can be developed independently from each other and, as such, also have their own release cycle. A microservice was then compiled and packed into an executable or into, say, a JAR or WAR if you were using

Java as your coding language of choice. A microservice could then be deployed side by side with other microservices onto a single host, a bare metal server, or a **virtual machine (VM)**. A microservice could also be exclusively deployed to a dedicated server if one wanted to run it in isolation for better protection or if the service was resource hungry and needed a lot of CPU or memory when running.

Although, in principle, this architecture looks simple, in practice, there were a few challenges to overcome. Let us assume the case where multiple microservices run on the same host. In this case, we were facing the following potential problems:

- Every microservice running on the same host was running in the same context, so to speak. They all used the same process, user, and network namespaces. They all were fighting for the same resources, such as CPU and memory or IO. A malicious service could easily compromise the other services running in the same context. A service could be very aggressively monopolizing available resources and, at the same time, starving all the other services, resulting in the so-called noisy neighbor problem.
- Let us assume all microservices were written in Java, but not every service used the same version of Java. Thus, the infrastructure team had to make sure to have all required versions of the **Java Runtime Environment (JRE)** installed on the host. Things got even worse if the different teams were using different languages and frameworks to develop their respective service. Just imagine a situation where team A developed service S1 in Java 8, team B developed service S2 in Python 2.7, team C used Node JS 15.x to implement service S3, and so on. In this case, the poor infrastructure team had to make sure all of those runtime environments were installed and maintained on the host. This was a huge burden and often resulted in quite a mess. To make things worse, often, different versions of the same framework were incompatible with each other and could not co-exist on the same host.

Now, let us assume we want to avoid all the preceding problems by placing each microservice onto a dedicated host. But this creates another bunch of problems, as follows:

- Using a dedicated server or VM for each microservice is most often a huge waste. A somewhat reasonable VM can easily cost several dozens of dollars per day, and at the same time, its CPU will most probably remain mostly idle the whole day with just a single microservice residing on it. The problem is that we cannot just provision the cheapest, low-power VM, though, since the VM needs to be able to handle the occasional peek loads of the microservice, which can by far exceed the average load.
- If our application is under load, we may need to scale up certain microservices. Scaling up and down additional VMs when a microservice needs to scale when the load on the application changes significantly can take a long time.

Here, a long time means several minutes, which is not acceptable for high-volume applications.

- Last but not the least, we should also be conscious about our environment. More servers mean more energy consumption. Yet our cloud providers are far from being carbon neutral at the time of this writing.

The solution once again is containers. Microservices can be encapsulated into containers, and multiple containers can be packed on a single VM to share the available resources. Containers can scale out and in near instantly, in a matter of milliseconds or a few seconds at most.

Providing standard deployment model

An additional benefit of using containers is the fact that IT operations engineers can focus on the specifics of their role. They are good at provisioning infrastructure such as networks, clusters, routers, and so on. They are also good at monitoring this infrastructure. The use of containers helps in standardizing infrastructure and processes associated with it. When thinking of servers or cluster nodes, every one of those is almost identical all the time, and it is just another container host. There is no need to install any specific frameworks or libraries on such servers. The only thing required is an OS optimized to host containers such as CoreOS.

Therefore, operations engineers or, more specifically, **system reliability engineers (SREs)**, are not required to be familiar with the internal details of application services running inside containers as those containers are supposed to be self-contained and include everything necessary to successfully run the service on any compatible container host. This way, SRE can look at a container and treat it as if it were a black box. This is not much different from how a shipper today looks at a physical container. They do not care so much about what is inside but just the fact that they need to transport the container from A to B.

We will look into this use case in more detail in *Chapter 10, Container orchestration* and *Chapter 12, Enabling zero-downtime deployments.*

Improved security

It is no secret to us that cyber-attacks are on the rise. Every day we hear in the news of high-profile companies that have been under attack and that highly sensitive data got stolen from their servers. Often important data gets encrypted, and the attackers ask for money, mostly in the form of cryptocurrency, to unlock the data. Sensitive data ranges from personal data such as names, phone numbers, and e-mail addresses to financial data such as credit card numbers or bank account info. Depending on the sector, it can also be sensitive health-related information that gets stolen. Apart

from customer data, the attackers can also steal company secrets. Luckily the use of containers can help us in mitigating the risk of such events. Through the use of Linux low-level functionality—we call them Linux primitives—an application running inside a container is by default more secure than the same application running natively on a server or VM. By using Linux namespaces, different applications are sandboxed from each other. And through the use of Linux cgroups we can limit the amount of resources each containerized application can request and thus avoid scenarios where a malicious application can starve all the other applications running on the same server.

Since container images are immutable, it is straightforward and simple to use special tools that scan all the layers of an image for **Common Vulnerabilities and Exposures** (**CVEs**). By identifying CVEs and eliminating them, we harden our applications and further improve their security.

We can add an additional layer of protection on top of our containers by using content trust. With content trust, the author of a Docker image digitally signs it. The recipient can then verify this cryptographic signature and make sure that the image is indeed originating from the claimed source and that the content of the image has not been tampered with on its way to the destination. This way, we can avoid so-called **man-in-the-middle** (**MITM**) attacks.

Since applications running in a container are running inside their own namespace, they are protected from the outside world. Each container defines a new user and process namespace. If your application defines a user Bob, and another application running in a different application also defines a user Bob, then these two users have nothing in common, as they are living in different namespaces. It is like the real world, where a person named Bob Doe living in Dallas has nothing to do with Bob Doe living in New York. The postal service can easily distinguish those two people via their addresses. In that regard, the City, Street, and house number work as namespaces and can be used to uniquely identify a person whose name alone may not be unique. On Linux, every process has an associated process ID. The fact that a Docker container associates a new process namespace with each container makes it possible and easy to isolate processes from each other, although they are running on the same computer or VM. Namespaces are only one part of the benefits, though. The other ones are cgroups. cgroups allow us to define how many resources a container may consume. Resources are things like the number of CPU cores, amount of RAM, and network bandwidth, to just name the most important ones. By giving us the ability to limit the resource consumption of a container, we can limit the so-called noisy neighbor problem. We will talk more about security and containers in *Chapter 13, Securing a Container*.

Reducing friction in the development process

By using containers during the development of software, developers have the ability to develop and test their applications in a production-like environment right on their own computers. If their application uses a database such as PostgreSQL, they can run this database inside a container right on their local machine. Similarly, they can run other infrastructure or middleware such as document databases like MongoDB, caches like Redis, blob storage such as AWS S3 compatible blob storage, search engines such as ElasticSearch, and many more on their laptop or desktop computer. The only requirement for them is to have enough RAM available. Usually, a minimum of 16 GB of RAM gets you going, but 32 GB is better. With the ever-lower prices for memory, though, this should not be a problem. We have seen many developers using beefier laptops with up to 64 GB of RAM. The ability to run all these containers locally makes for a much-improved developer experience and a much-shortened feedback cycle. Instead of relying on shared servers or cloud resources that may be in an ever-changing state, developers can now have their very own setup that is not polluted by data from other developers or bogged down by the activity of other team members. We talk more about this in *Chapter 8, Testing and debugging containerized applications.*

Everything you have been told so far can, in principle, also be achieved without the use of containers, but with a much bigger effort and resulting maintenance burden. Containers have become the de facto global standard, and with this, it is so much simpler to implement and enforce best practices.

Skills, jobs, salaries, and career paths

In this section, we are going to look into the required skills, new job roles, salaries, and career paths related to Docker containers.

Looking at a company that practices agile software development and invests in continuous delivery, we can see a high demand for roles with the following skills:

Developers

Developers are expected to have a strong understanding of Docker containers and how to use them to reduce friction in their daily development workflow. If a candidate has advanced container skills and is proficient in using containers, building custom container images, and running applications consisting of several containers locally on their machine, he or she has a clear advantage over competing candidates. Finally, if you can credibly show to have a working knowledge of and familiarity with Kubernetes, you are definitely in a good position.

DevOps engineers

Companies are also desperately looking for DevOps engineers that master Docker containers and Kubernetes. If you can prove your expertise in setting up a fully automated CI/CD pipeline, either from scratch or based on infrastructure provided by cloud providers such as Microsoft Azure, Google Cloud, or AWS, then you are in a good position. Intimate familiarity with Kubernetes as the deployment target is a huge advantage.

These days companies develop applications that have to be up and running all the time, 24×7. At the same time, applications need to be updated all the time, mainly when adding new features to them. Those updates should happen at any time while requiring zero downtime. Thus, you should be familiar with concepts such as rolling updates, blue-green deployment, and canary releases. It is an advantage if you have worked with tools such as Helm on Kubernetes. Authoring Dockerfiles or Docker Compose files should be a breeze for you.

You should also have a good understanding of Linux and, more specifically, of Linux namespaces and cgroups that you will leverage to better protect the applications or services running inside a container. Using the possibilities of cgroups you know how to limit the amount of resources a container has access to. With Linux namespaces, you isolate the different services running in the various containers from each other, providing optimal encapsulation and security.

QA automation engineers

Let us talk about testing now. QA automation engineers are in high demand. Relying on manual testing is a thing of yesterday and is not scalable. Thus, manual testing should only be used for exploratory testing. Consequently, you are a specialist in writing automated tests. Following the recommendations of the so-called test pyramid, you write a lot of what we call component tests, where the system under test runs in a container in isolation, and your test code probes this application or service in the container via its public APIs, such as REST API or messages when using a service bus. People also call this black-box testing. The test code will run in a container as well, as will the other services the system under test uses, such as databases and message queuing systems or caches. That said, the industry expects you to not only be familiar with the latest test frameworks but also to be very familiar with Docker containers and the use of multi-container applications. It is a big bonus if you are familiar with Kubernetes and can perform or run your tests on a Kubernetes cluster to simulate a production environment as closely as possible.

You should have experience in using popular Docker images that allow you to run a relational or no-SQL database such as PostgreSQL or MongoDB, a message queue such as RabbitMQ or ActiveMQ, blob storage simulators compatible with the AWS

S3 format, distributed cache such as Redis, search engines such as ElasticSearch and more, to support you in more efficiently test your assigned services and applications in isolation and locally or on a build agent in the cloud.

Operations engineers

Finally, let us talk about operations engineers. You should be able to provision and operate container hosts and Kubernetes clusters. You are familiar with tools that enable you to monitor the infrastructure on top of which armies of containers will run. Ideally, you are using **Infrastructure as Code (IaC)** to provision all your Kubernetes clusters, network configurations, load balancers, data volumes, and role-based access control. You know how to configure your Kubernetes clusters so that they can grow and shrink on demand and that they are redundant and span multiple availability zones. You are able to secure your Kubernetes clusters and make them immune to all the cyber-attacks that happen all the time, every day.

But do not panic. Most, yet not all, of the skills you have just read about are covered in this book. Those skills that we do not cover in the book you can easily acquire once you have mastered the content of this book.

Salaries and career paths

According to data from salary comparison websites and job posting sites, the average salary for a Docker container expert can range from $80,000 to $150,000 per year, depending on the location and level of experience. However, this is just a rough estimate and may not accurately reflect the actual salary of a Docker container expert in a specific location or industry.

There are several career paths that a Docker container expert might pursue, depending on their skills, interests, and goals. Some potential career paths for Docker container experts might include:

- **Systems administrator**: A systems administrator is responsible for the installation, configuration, and maintenance of computer systems and servers. They might work with Docker containers to deploy and manage applications in a production environment.
- **DevOps engineer**: A DevOps engineer is responsible for designing and implementing processes and tools to automate the development, testing, and deployment of software. They might work with Docker containers to automate the build, test, and deployment of applications in a **continuous integration and continuous delivery** (**CI/CD**) pipeline.
- **Software developer**: A software developer is responsible for designing, developing, and maintaining software applications. They might work with

Docker containers to package and deploy their applications in a consistent and reproducible manner.

- **Consultant**: A consultant is an expert in a specific field who is hired by organizations to provide advice and guidance on a particular problem or challenge. A Docker container expert might work as a consultant to help organizations adopt and implement Docker container technologies.
- **Cloud computing**: Cloud computing involves delivering computing resources and services over the internet, allowing organizations to access and use computing resources on demand. Docker containers can be used in cloud environments to package and deploy applications in a consistent and reproducible manner, making it easier to scale and manage applications across different cloud environments. A Docker container expert might work in a cloud computing role, such as a cloud solutions architect, cloud engineer, or cloud operations engineer, and be responsible for designing, building, and maintaining cloud infrastructure and applications.
- **Data engineering**: Data engineering involves designing, building, and maintaining systems for collecting, storing, processing, and analyzing data. Docker containers can be used in data engineering to deploy and manage data processing and analytics pipelines in a reproducible and scalable manner. A Docker container expert might work in a data engineering role, such as a data engineer, data pipeline engineer, or data platform engineer, and be responsible for designing and building data infrastructure and pipelines.
- **Cybersecurity**: Cybersecurity involves protecting computer systems and networks from cyber threats and vulnerabilities. Docker containers can be used in cybersecurity to deploy and manage secure applications and environments in a consistent and reproducible manner. A Docker container expert might work in a cybersecurity role, such as a security engineer, security analyst, or security architect, and be responsible for designing and implementing security solutions to protect systems and data.

These are just a few examples of potential career paths for Docker container experts. There are many other job roles and industries that might be relevant for individuals with expertise in Docker containers.

Conclusion

In this chapter, we looked at what containers are by giving a very familiar analogy. We then discussed the most important use cases of containers. We showed ways in which containers can help us to significantly reduce the friction in the software development life cycle and, on top of that, how containers can make the software supply chain more secure and resilient against cyber threats. We also showed how

the use of containers and related technologies brought a shift in the demand for skills and resulted in new job roles.

In the upcoming chapter, we are going to setup our personal computers in the best possible way so that we can work with containers efficiently and effectively. This includes installing the Docker Desktop software that, to this day, remains one of the developers' favorite tools to work with containers. Stay tuned.

Questions

To assess your progress, please try to answer the following questions:

1. Containers and VMs are not the same. Provide three differences.
2. Name the two foundational capabilities of an operating system that enable containers.
3. Name at least three typical use cases for Docker containers
4. Use a few short sentences to explain what the differences between containers and Docker are?
5. Explain in a few short sentences what runC is.
6. Where can containers run? Name three to five different host systems.

Answers

Here are the answers to the preceding questions:

1. A few important differences between VMs and containers are:
    a. VM contains a complete operating system; the container does not
    b. A VM is more secure than a container, as it contains its own OS kernel and is, in general, having better isolation against the outside world than a container
    c. A VM is considered to be long living; a container is supposed to be ephemeral
    d. VMs are heavyweight, whereas containers are (often) lightweight
    e. VMs have a much longer startup time than containers
2. The two capabilities are namespaces and cgroups.
3. Here is a list of typical use cases for Docker containers
    a. Supercharging microservice architecture
    b. Removing friction from the development process

   c. Powering CI/CD pipelines
   d. Providing a standard deployment model for enterprises
   e. Enabling standard tooling and infrastructure (see CNCF)
   f. Lowering the TOC and the time between new releases of legacy applications

4. According to the official page: **https://containerd.io**, containerd is available as a daemon for Linux and Windows. It manages the complete container lifecycle of its host system, from image transfer and storage to container execution and supervision to low-level storage to network attachments and beyond. Contrary to that, Docker is built on top of runC and containerd and provides additional features on top of it, such as a robust CLI, Docker compose, and more, which make the use of containers much easier for developers, DevOps, and operations engineers.
5. According to Docker, the originator of runC, it is a lightweight, universal container runtime. It includes all of the plumbing code used by Docker to interact with system features related to containers. It is designed with the following principles in mind:
   a. Designed for security
   b. Usable at large scale and ready for production
   c. No unwanted dependencies: just the container runtime and nothing else
6. In the meantime, containers can be hosted on a vast number of systems. Here is an incomplete list:
   a. Your laptop, be it a Windows machine, a Mac, or a Linux machine
   b. Server or VM running Linux (various flavors)
   c. Server or VM running Microsoft Windows Server
   d. An IBM mainframe
   e. Supercomputers
   f. A tiny single-board computer, such as Raspberry PI
   g. An edge computer (IoT)

Job interview sample questions

So, you finally got a chance for an onsite interview for a job as a software developer. The job is offered by a cool new startup that develops a highly acclaimed new mobile app for iOS and Android. During the interview, you are confronted with the following questions:

1. Our company is not yet using Docker containers. The management is hesitant and not sure whether to invest in this new technology. Please provide the "elevator pitch" for containers to our CEO.
2. We are an IT shop specialized in providing a cloud-native SaaS solution in the insurance business. The CTO is on edge to start adopting Docker containers in the company. What important use cases would you present him that speak for quick and consequent adoption of containers? Why are these use cases important for our purpose?
3. In our company, there is a lot of confusion about what a VM is versus what a container is. Some vocal engineers claim that they are basically the same. And since we are already using VMs to run our software, we do not have to bother about containers, they say. Please help us understand why this is not true by providing three to four clear differences between VMs and containers.
4. Our company does not yet use containers. If you had a say, where would you start? What would be your first use case for containers? Explain why you would start like this.
5. Our head of security claims that containers are as secure as VMs. Is she right? Justify your answer.

Sample answers

Here are some sample answers to the preceding questions raised during the interview:

1. Containers have many advantages we should and can leverage. Similar to what the introduction of physical containers did to the shipping industry, the introduction of software containers can help our company to standardize on infrastructure and processes in the **Software Development Life Cycle** (**SDLC**). It can help reduce the friction during the development phase by enabling our SW engineers to write, test and debug the software they write in a production-like environment right on their local machines, as it is possible to run middleware and infrastructure such as databases inside containers right on their computers. Furthermore, containers isolate applications and their dependencies inside containers and thus decouple each application from all the others. No more version conflicts on the host machine due to different requirements of different applications running on the same machine, as the dependencies, such as frameworks or libraries, are packaged with the applications inside their respective container. The whole software release cycle will be streamlined and become more robust. We can use containers as artefacts of our versioning strategy, and we power

our whole CI/CD pipelines with containers. Containerized applications are also inherently more secure than applications running natively on the host computer, as containers provide a good layer of isolation to the applications. Many more advantages are available when using containers. But the ones we named are some of the most compelling ones.

2. We would present the CTO with the following use cases:
    a. Standardize deployment model:
        i. When using Kubernetes to run containerized applications, the deployment always looks the same. It is always based on a Docker image, no matter what is running inside the image, e.g., a Java or .NET application or a legacy Cobol application; it's "just a container".
    b. Streamline release cycle: …
        i. Using containers as versioning artefacts instead of files such as JARs
        ii. Application and dependencies form a unit of deployment
        iii. Rollbacks are easy; just redeploy the previous version of a Docker image
    c. Improve security: image scanning, image signing, namespaces, and cgroups.
    d. Better scalability:
        i. Faster to startup and tear-down
        ii. Kubernetes solves many of the hard problems of a highly distributed, mission-critical enterprise application, such as scaling, routing, self-healing, high availability, and more
3. On first look, containers look a bit like VMs. But if we look closer, then there are quite some differences between the two. A VM is like a perfect sandbox and encloses a complete virtual server in it. A VM thus contains a complete operating system such as Linux or Windows Server. Applications that run on a VM are perfectly shielded from the outside. The only connection with the outside world is over the network, which can be easily configured to be saved. On the other hand, containers share the Kernel of the host system with each other. Multiple containers can run on a single container host. A VM can be a container host. It is not practicable, though, to run a VM inside a container. Since containers share the Kernel of the container host, they are much cleaner than a VM. Resource utilization is usually much better when a server of a VM hosts many containers than if an application directly runs on a VM. VMs are meant to be stable and run "forever". Containers are meant to

be ephemeral and come and go at will. This leads to the fact that containers usually start up in a matter of milliseconds, whereas VMs need a few minutes to be up and running.

4. We would start using containers to reduce the friction in the development process. Developers should be able to simulate a production-like environment on their local developer machines. For example, they should be able to run databases such as PostgreSQL or MongoDB locally. Other examples include message queue applications such as RabbitMQ or ActiveMQ, search providers such as ElasticSearch, and more. It is way easier to run those supporting applications in containers than to install them natively on the developers' workstations. With this approach, the company can gain familiarity with containers and their supporting ecosystem and then gradually use containers for more advanced scenarios such as testing and debugging. Using containers in production should be the last step. Only do this once you are very familiar with this new technology, its pros but also its cons.
5. No, the head of security is wrong. A VM is still much more secure than a container. A VM is a completely isolated sandbox containing a full operating system such as Linux or Windows Server. All containers that run on the same container host, on the other hand, share the kernel of the host among each other. It is, therefore, possible that a malicious application can use this fact to exploit the other services running in the other containers on the same host. Thus it is important to really understand the limits of container security when running multiple containers in parallel on the same host.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 2
Setting Up Your Environment

Introduction

In the previous chapter, we have learned what Docker containers are and why they have truly revolutionized the IT landscape. We also got compelling reasons why we absolutely should use containers in our own projects.

Every skilled craftsperson, whether a man or a woman, uses a toolbox that has been carefully chosen, with exactly the correct amount of tools for the work and no extraneous clutter. Similar to software engineers, DevOps or operations engineers require a small number of properly chosen tools in order to be productive. It is far preferable to have a small number of highly specialized instruments than a seemingly endless array of devices that you only have a cursory understanding of.

Many of you will have a Windows 10 or 11 machine to work with, and others will be using a Mac. Maybe you are even using a laptop with a Linux OS installed. To make sure that the instructions and commands presented in this book will be applicable in all those scenarios, it is important that we standardize on a set of tools that are common on all operating systems mentioned. In most cases, the instructions presented in this book apply to Windows, MacOS, and Linux alike. On special occasions where they differ, the book provides alternatives.

Structure

In this chapter, we will discuss the following topics:

- Selecting and preparing a package manager
- Installing and using Docker Desktop
- Selecting a code editor and useful plugins
- Configuring and using a great terminal
- A brief introduction to play with Docker and Kubernetes
- Recap the learnings
- Questions
- Job interview sample questions

Objectives

By the end of this chapter, you will have installed all tools needed to efficiently and effectively work with Docker containers. You will start by installing a package manager onto your laptop, which makes subsequent installs much easier and straightforward. You then will install the probably most important tool of all, Docker Desktop. It is a versatile and powerful tool that you will use to manipulate and run Docker containers locally. This will be followed by installing a powerful terminal that you will be using all the time throughout this book to execute Docker-related commands and more. Finally, you will receive a short introduction to Play with Docker and Play with Kubernetes. Those online sandboxes provide you with an easy and frictionless opportunity to experiment with Docker and Kubernetes if you do not have a laptop at hand that has these tools locally installed.

Let us start by installing the package manager on your laptop.

Selecting and preparing a package manager

Although any of the applications or tools we are going to touch on in this chapter can be installed manually, we prefer to use a package manager for doing the same in a more predictable and repeatable manner.

We will first install a package manager on a Windows 10 or 11 machine and then on MacOS.

Installing chocolatey on Windows 10 or 11

On Windows 10 and 11, our package manager or choice is Chocolatey. It is very popular, and most popular applications, packages, and libraries can be installed with it. More information about this package manager can be found here: **https://chocolatey.org/**.

Technically, we could also use other package managers apart from chocolatey. Specifically, the new WinGet package manager from Microsoft looks interesting (**https://docs.microsoft.com/en-us/windows/package-manager/winget/**). Due to the fact that it is still in its infancy and that chocolatey has proven to be very reliable, we stick with our choice.

Before we start, let us have a quick note about command line tools on Windows. On a Windows computer, there are different tools available. The most familiar of those all is probably the command shell. It has been part of the OS since the very beginning. It is a very basic shell.

Over the years and as requirements increased, Microsoft has developed PowerShell. This tool is very powerful and popular among engineers working with and managing Windows. It is also possible to install third-party tools such as Git that provide a Bash shell.

We hence recommend that you either use PowerShell or any other Bash tool while following the samples in this book. Now, let us continue.

To install Chocolatey on Windows 10 or later, go through the following steps:

1. Open PowerShell in Admin mode. To do so, for example, press the windows key and type *powershell* in the search bar. In the window that pops up, click **Run as Administrator**.

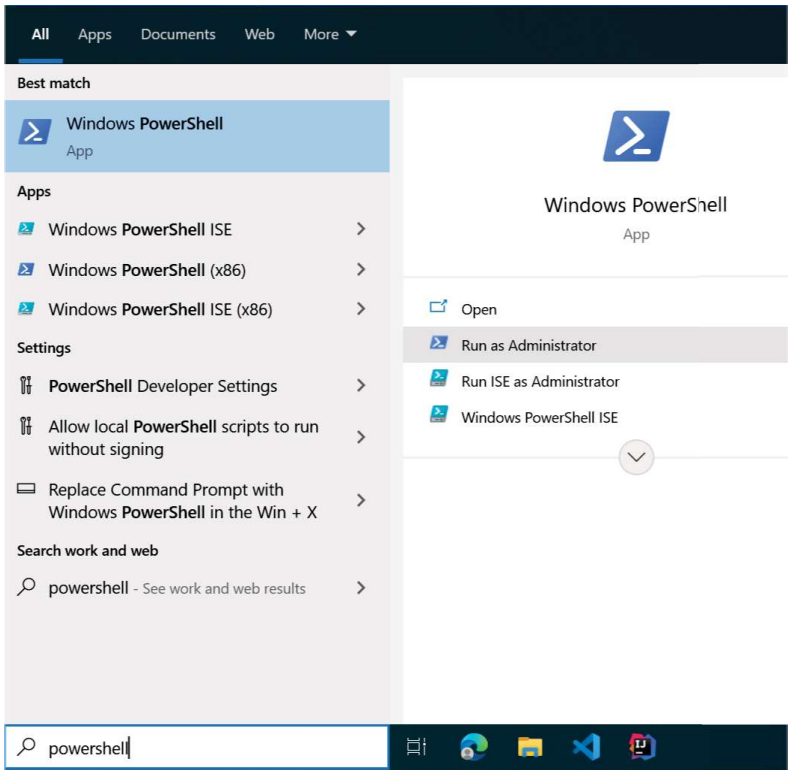

***Figure 2.1**: Starting PowerShell in Admin mode*

Note: It is essential to run PowerShell in elevated mode; otherwise, the installation will fail.

2. Execute the following command inside the Powershell window:

```
Set-ExecutionPolicy Bypass -Scope Process -Force; [System.
Net.ServicePointManager]::SecurityProtocol = [System.Net.
ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-
Object System.Net.WebClient).DownloadString('https://chocolatey.
org/install.ps1'))
```

   This command will download and run the `install.ps1` script from the Chocolatey website. If you do not want to type this horribly long and convoluted command, you can also copy it from here: **https://chocolatey.org/install**.

3. The preceding command will take a few seconds to execute. If you do not see any errors in the output, you are now ready to use chocolatey. To verify this, execute **`choco -v`**

   In the PowerShell console. The version of the package manager should be output. At the time of this writing, the version is **`0.10.15`**.

4. **Exercise**: Try to install a simple tool such as **Notepad++** using Chocolatey. Use the following link to find out what exact command to use: **https://community.chocolatey.org/packages**

   Hint: the command should look similar to this:
   `choco install <app-name>`

   Where **`<app-name>`** is the name of the package to install.

5. **Exercise**: Do the same for Git, which we will be needing later on in the book.

NOTE: You can skip the next section and proceed with the installation of Docker Desktop.

Installing Homebrew on MacOS

On MacOS, the choice of a package manager is easy. The only real contender in this space is Homebrew. Pretty much everybody uses it. More information about this tool can be found here: **https://brew.sh/index**.

To install Homebrew, proceed as follows:

1. Start your default Terminal (type ⌘-*SPACE* to display the Spotlight search box and enter `terminal` in the search box and hit *ENTER*)

2. In the terminal, execute the following command (note that you will have to provide your password for security reasons):

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

3. Now to complete the installation, you have to add brew to your path. Do this with the following commands:

   ```
   echo 'eval "$(/opt/homebrew/bin/brew shellenv)"' >> ~/.zprofile
   eval "$(/opt/homebrew/bin/brew shellenv)"
   ```

4. Verify that your installation has succeeded with this command:

   ```
   brew -version
   ```

   You should see an output similar to this:
   ```
   Homebrew 3.1.7
   ```

```
Homebrew/homebrew-core (git revision b29800c865; last commit
2021-05-14)
Homebrew/homebrew-cask (git revision 21228459c5; last commit
2021-05-14)
```

5. **Exercise**: Try to install a simple tool such as **tree** using Homebrew. Use the following link to find out what exact command to use: **https://formulae.brew.sh**

   Hint: the command should look similar to this:

   ```
   brew install <app-name>
   ```

   Where `<app-name>` is the name of the package to install.

6. **Exercise**: Do the same for Git, which we will be needing later on in the book.

You are now ready to use Homebrew to install additional software on your Mac.

Now that we have prepared our package manager on Windows 10 & 11 or MacOS, we are ready to use it and install all the tools we need to work with Docker containers.

Installing and using Docker Desktop

Docker Desktop is the recommended tool for all engineers working with containers on their local machines. This tool is absolutely free and, according to Stack Overflow (**https://www.docker.com/blog/stack-overflow-survey-reconfirms-developer-love-for-docker/**), one of the most popular and beloved tools. This tool allows you to run Docker containers locally in a specialized VM. The tool is available for Windows, MacOS and Linux. If you want more detailed information about the product, please refer to this page: **https://www.docker.com/get-started**.

Installing Docker Desktop on Windows 10 or 11

Until recently, Docker Desktop could only be installed on Windows 10 Pro because it required the use of Hyper-V, which was not part of the Windows 10 Home edition. Luckily, this has changed with the introduction of the Windows Subsystem for Linux (WSL2) on Windows 10 by Microsoft. Now, Docker Desktop also runs on the Windows 10 Home edition.

TIP: WSL 2 is available on Windows 10 starting from version 1903.

To install Docker Desktop (or short Docker) on your Windows 10 or 11 machine, proceed as follows:

1. Open a PowerShell window in admin mode