
642439809

Docker

Up and Running

Build and deploy containerized web apps with Docker and Kubernetes



Dr. Gabriel Nicolas Schenker

bpb

Docker : Up and Running

Build and deploy containerized web apps with Docker and Kubernetes

Dr. Gabriel Nicolas Schenker



www.bpbonline.com

Docker : Up and Running

Build and deploy containerized web apps with Docker and Kubernetes

Dr. Gabriel Nicolas Schenker



www.bpbonline.com

Copyright © 2023 BPB Online

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

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

Copyright © 2023 BPB Online

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

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***

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 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.

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.

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 Dockerfiles, 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

Preface

This book introduces the readers to Docker, practicing the readers in writing the Docker commands and how to create Dockerfiles, 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

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

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.

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.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

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.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

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



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

1. Explaining Containers and their Benefits 1
Introduction 1
Structure 1
Objectives 2
What is a container? 2
Evolution of containers and associated infrastructure 3
Architecture and core capabilities 4
Main use cases 6
Standardize infrastructure 7
Turbocharge a microservice architecture 8
Providing standard deployment model 10
Improved security 10
Reducing friction in the development process 12
Skills, jobs, salaries, and career paths 12
Developers *12*
DevOps engineers *13*
QA automation engineers *13*
Operations engineers *14*
Salaries and career paths *14*
Conclusion 15
Questions 16
Answers *16*
Job interview sample questions 17
Sample answers 18

2. Setting Up Your Environment 21
Introduction 21
Structure 22
Objectives 22

Table of Contents

1. Explaining Containers and their Benefits 1
Introduction 1
Structure 1
Objectives 2
What is a container? 2
Evolution of containers and associated infrastructure 3
Architecture and core capabilities 4
Main use cases 6
Standardize infrastructure 7
Turbocharge a microservice architecture 8
Providing standard deployment model 10
Improved security 10
Reducing friction in the development process 12
Skills, jobs, salaries, and career paths 12
Developers *12*
DevOps engineers *13*
QA automation engineers *13*
Operations engineers *14*
Salaries and career paths *14*
Conclusion 15
Questions 16
Answers *16*
Job interview sample questions 17
Sample answers 18

2. Setting Up Your Environment 21
Introduction 21
Structure 22
Objectives 22

Selecting and preparing a package manager 22
Installing chocolatey on Windows 10 or 11 23
Installing Homebrew on MacOS 25
Installing and using Docker Desktop 26
Installing Docker Desktop on Windows 10 or 11 26
Installing Docker Desktop on MacOS 28
Using Docker Desktop to execute basic container commands 30
Configuring Docker Desktop 32
Selecting a code editor and useful plugins 33
Installing VS Code on Windows 10 34
Installing VS Code on MacOS 34
Installing VS Code extensions 35
Configuring and using a great terminal 36
Installing Windows Terminal 37
Installing ITerm2 Version 3 for MacOS 37
Experimenting online with Docker and Kubernetes 37
Play with Docker 38
Play with Kubernetes 40
Conclusion 41
Questions 41
Answers 42
Job interview sample questions 42
Possible answers to job interview questions 43

3. Getting Familiar with Containers 45
Introduction 45
Structure 46
Objectives 46
Downloading a first container image 46
Running a container 49
Running a container in detach mode 52
Running multiple instances of a container 53

Selecting and preparing a package manager 22
Installing chocolatey on Windows 10 or 11 23
Installing Homebrew on MacOS 25
Installing and using Docker Desktop 26
Installing Docker Desktop on Windows 10 or 11 26
Installing Docker Desktop on MacOS 28
Using Docker Desktop to execute basic container commands 30
Configuring Docker Desktop 32
Selecting a code editor and useful plugins 33
Installing VS Code on Windows 10 34
Installing VS Code on MacOS 34
Installing VS Code extensions 35
Configuring and using a great terminal 36
Installing Windows Terminal 37
Installing ITerm2 Version 3 for MacOS 37
Experimenting online with Docker and Kubernetes 37
Play with Docker 38
Play with Kubernetes 40
Conclusion 41
Questions 41
Answers 42
Job interview sample questions 42
Possible answers to job interview questions 43

3. Getting Familiar with Containers 45
Introduction 45
Structure 46
Objectives 46
Downloading a first container image 46
Running a container 49
Running a container in detach mode 52
Running multiple instances of a container 53

Listing container images and container instances 54
Listing Docker images 54
Listing containers 56
Stopping and removing a container 58
Stopping and removing containers in Docker Desktop and VS Code 61
Listing processes running inside a container 65
Retrieving the log of a container 67
Retrieving the log from the command line 67
Displaying container logs in Docker Desktop 68
Displaying container logs in VS Code 69
Running an additional process inside an already running container 70
Conclusion 72
Questions 73
Answers 75
Job interview sample questions 75
Sample answers 75

4. Using Existing Docker Images 77
Introduction 77
Structure 77
Objectives 78
Running a SQL database locally 78
Running a PostgreSQL database server 78
Working with the database 82
Accessing the database from another container 84
A simple node JS app to access PostgreSQL 85
Running a no-SQL database locally 88
Accessing MongoDB from a different container 91
A simple Python app to access Mongo DB 91
Investigating the image layers model 93
What is a Docker image? 94
The layered filesystem 94

Listing container images and container instances 54
Listing Docker images 54
Listing containers 56
Stopping and removing a container 58
Stopping and removing containers in Docker Desktop and VS Code 61
Listing processes running inside a container 65
Retrieving the log of a container 67
Retrieving the log from the command line 67
Displaying container logs in Docker Desktop 68
Displaying container logs in VS Code 69
Running an additional process inside an already running container 70
Conclusion 72
Questions 73
Answers 75
Job interview sample questions 75
Sample answers 75

4. Using Existing Docker Images 77
Introduction 77
Structure 77
Objectives 78
Running a SQL database locally 78
Running a PostgreSQL database server 78
Working with the database 82
Accessing the database from another container 84
A simple node JS app to access PostgreSQL 85
Running a no-SQL database locally 88
Accessing MongoDB from a different container 91
A simple Python app to access Mongo DB 91
Investigating the image layers model 93
What is a Docker image? 94
The layered filesystem 94

The mutable container layer 96
Docker copy-on-write mechanism 97
Using volumes with containers 98
Creating, inspecting, and deleting Docker volumes 99
Mounting a volume into a container 101
Mounting a host folder into a container 103
Configuring an app running inside a container 105
Conclusion 107
Questions 108
Answers 109
Job interview sample questions 110
Job interview sample answers 110

5. Creating Your Own Docker Images 113
Introduction 113
Structure 113
Objective 114
Creating a custom Docker image 114
Creating an image interactively 114
Authoring a Dockerfile 118
The Docker image build process 119
The FROM keyword 122
The COPY and ADD keywords 123
The RUN keyword 125
The WORKDIR keyword 126
The CMD and ENTRYPOINT keywords 127
The EXPOSE keyword 130
Multi-step Dockerfiles 131
Dockerfile best practices 134
Importing a Docker image 134
Building a Docker image 136
Leveraging the Docker Build Kit 139

The mutable container layer 96
Docker copy-on-write mechanism 97
Using volumes with containers 98
Creating, inspecting, and deleting Docker volumes 99
Mounting a volume into a container 101
Mounting a host folder into a container 103
Configuring an app running inside a container 105
Conclusion 107
Questions 108
Answers 109
Job interview sample questions 110
Job interview sample answers 110

5. Creating Your Own Docker Images 113
Introduction 113
Structure 113
Objective 114
Creating a custom Docker image 114
Creating an image interactively 114
Authoring a Dockerfile 118
The Docker image build process 119
The FROM keyword 122
The COPY and ADD keywords 123
The RUN keyword 125
The WORKDIR keyword 126
The CMD and ENTRYPOINT keywords 127
The EXPOSE keyword 130
Multi-step Dockerfiles 131
Dockerfile best practices 134
Importing a Docker image 134
Building a Docker image 136
Leveraging the Docker Build Kit 139

Tagging Docker images ... 140
Creating an account on Docker Hub ... 141
Publishing Docker images ... 142
Publishing to a personal account ... 142
Publishing to an organization ... 144
Conclusion ... 146
Questions ... 147
Answers ... 147
Job Interview sample questions ... 148
Job interview sample answers ... 149

6. Demystifying Container Networking ... 151
Introduction ... 151
Structure ... 152
Objective ... 152
Explaining the container network model ... 152
Discussing the bridge network ... 155
Inspecting a network ... 156
Creating a new bridge network ... 158
Analyzing the network stack of a container ... 158
DNS name resolution on a bridge network ... 161
Elucidating container ports ... 162
Auto mapping ports ... 163
Explicitly mapping ports ... 166
Creating and running a containerized REST API ... 167
Discussing the none and host networks ... 169
The host network ... 169
The none network ... 171
Conclusion ... 172
Questions ... 173
Answers ... 173
Mastering job interview question ... 174

Tagging Docker images ... 140
Creating an account on Docker Hub ... 141
Publishing Docker images ... 142
Publishing to a personal account ... 142
Publishing to an organization ... 144
Conclusion ... 146
Questions ... 147
Answers ... 147
Job Interview sample questions ... 148
Job interview sample answers ... 149

6. Demystifying Container Networking ... 151
Introduction ... 151
Structure ... 152
Objective ... 152
Explaining the container network model ... 152
Discussing the bridge network ... 155
Inspecting a network ... 156
Creating a new bridge network ... 158
Analyzing the network stack of a container ... 158
DNS name resolution on a bridge network ... 161
Elucidating container ports ... 162
Auto mapping ports ... 163
Explicitly mapping ports ... 166
Creating and running a containerized REST API ... 167
Discussing the none and host networks ... 169
The host network ... 169
The none network ... 171
Conclusion ... 172
Questions ... 173
Answers ... 173
Mastering job interview question ... 174

Sample answers to the job interview questions 175

7. Managing Complex Apps with Docker Compose 177

Introduction 177

Structure 177

Objective 178

Explaining the docker-compose syntax 178

Imperative versus declarative syntax 178

Inspecting a simple Docker compose file 179

Using build instructions 181

Using networks 181

Authoring your own Docker-compose file 183

Preparing a multi-service application 183

The API service in Java 183

The API service in .NET 5.x 190

The Web service 197

Writing your first docker-compose file 200

Building and running all or individual services 204

Preparing the database 204

Building service with Docker compose 208

Pushing images to a registry 208

Running an application 209

Running, restarting, stopping, and removing individual services 211

Logging with Docker compose 212

Advanced usages of Docker compose 212

Defining dependencies between services 213

Scaling a service 215

Defining a project name 217

Using docker compose overrides 218

Conclusion 220

Questions 221

Job interview sample questions 221

Sample answers to the job interview questions 175

7. Managing Complex Apps with Docker Compose 177

Introduction 177

Structure 177

Objective 178

Explaining the docker-compose syntax 178

Imperative versus declarative syntax 178

Inspecting a simple Docker compose file 179

Using build instructions 181

Using networks 181

Authoring your own Docker-compose file 183

Preparing a multi-service application 183

The API service in Java 183

The API service in .NET 5.x 190

The Web service 197

Writing your first docker-compose file 200

Building and running all or individual services 204

Preparing the database 204

Building service with Docker compose 208

Pushing images to a registry 208

Running an application 209

Running, restarting, stopping, and removing individual services 211

Logging with Docker compose 212

Advanced usages of Docker compose 212

Defining dependencies between services 213

Scaling a service 215

Defining a project name 217

Using docker compose overrides 218

Conclusion 220

Questions 221

Job interview sample questions 221

Sample answers 222

8. Testing and Debugging Containerized Applications 223

Introduction 223

Structure 223

Objective 224

Enabling hot reload in a container 224

Hot reloading a .NET application 225

Hot reloading a NodeJS application 226

Hot reloading a ReactJS application 228

Debugging an app running inside a container 232

Debugging a Java application 232

Debugging a .NET application 237

Debugging a NodeJS application 241

Debugging a Python application 244

Writing and running unit and integration tests 247

Instrumenting a containerized app 252

Instrumenting a Java application 253

Application level logging 253

Using Spring Boot actuators 255

Conclusion 256

Questions 257

Answers 257

Job Interview sample questions 258

Sample answers to interview questions 258

9. Establishing an Automated Build Pipeline 261

Introduction 261

Structure 261

Objectives 262

Building a Docker image using GitHub actions 262

Preparing an application and a GitHub repository 262

Introducing GitHub actions and workflows 264

Sample answers 222

8. Testing and Debugging Containerized Applications 223

Introduction 223

Structure 223

Objective 224

Enabling hot reload in a container 224

Hot reloading a .NET application 225

Hot reloading a NodeJS application 226

Hot reloading a ReactJS application 228

Debugging an app running inside a container 232

Debugging a Java application 232

Debugging a .NET application 237

Debugging a NodeJS application 241

Debugging a Python application 244

Writing and running unit and integration tests 247

Instrumenting a containerized app 252

Instrumenting a Java application 253

Application level logging 253

Using Spring Boot actuators 255

Conclusion 256

Questions 257

Answers 257

Job Interview sample questions 258

Sample answers to interview questions 258

9. Establishing an Automated Build Pipeline 261

Introduction 261

Structure 261

Objectives 262

Building a Docker image using GitHub actions 262

Preparing an application and a GitHub repository 262

Introducing GitHub actions and workflows 264

Adding a build Docker image workflow 269
Creating a complete CI pipeline 271
Using Jenkins to setup a build pipeline 276
A complete CI/CD pipeline 277
Creating a CI pipeline 278
A complete CI pipeline in Jenkins 283
Adding continuous delivery 290
Discussing some cloud-based SaaS offering for your CI/CD pipeline 293
Azure DevOps 294
AWS DevOps 295
GitLab 297
Conclusion 300
Questions 300
Answers 300
Job Interview sample questions 301
Answers 302

10. Orchestrating Containers 303
Introduction 303
Structure 303
Objectives 304
Distributed application architecture 304
Patterns and best practices 307
Loosely coupling components 307
Preferring stateless over stateful components 307
Using service discovery 308
Routing calls between components 309
Load balancing traffic 310
Programming defensively 310
Guaranteeing redundancy 311
What is an orchestrator? 312
The responsibilities of an orchestrator 312

Adding a build Docker image workflow 269
Creating a complete CI pipeline 271
Using Jenkins to setup a build pipeline 276
A complete CI/CD pipeline 277
Creating a CI pipeline 278
A complete CI pipeline in Jenkins 283
Adding continuous delivery 290
Discussing some cloud-based SaaS offering for your CI/CD pipeline 293
Azure DevOps 294
AWS DevOps 295
GitLab 297
Conclusion 300
Questions 300
Answers 300
Job Interview sample questions 301
Answers 302

10. Orchestrating Containers 303
Introduction 303
Structure 303
Objectives 304
Distributed application architecture 304
Patterns and best practices 307
Loosely coupling components 307
Preferring stateless over stateful components 307
Using service discovery 308
Routing calls between components 309
Load balancing traffic 310
Programming defensively 310
Guaranteeing redundancy 311
What is an orchestrator? 312
The responsibilities of an orchestrator 312

Reconciling the desired state ... 312
Replicating and global services ... 313
Enabling service discovery ... 314
Providing routing services ... 314
Distributing work by load balancing ... 315
Dealing with fluctuating load by scaling ... 315
Facilitating self-healing ... 316
Enabling zero-downtime deployments ... 317
Providing affinity and location awareness ... 318
Security ... 319
Secure communication and cryptographic node identity ... 319
Secure networks and network policies ... 320
Role base access control (RBAC) ... 320
Secrets ... 320
Content trust ... 321
Reverse uptime ... 321
Facilitating introspection ... 322
Overview of popular orchestrators ... 323
Kubernetes ... 323
Docker Swarm ... 324
Apache Mesos and Marathon ... 325
Amazon AWS ECS ... 326
Microsoft Azure ACS ... 327
Understanding the architecture of Kubernetes ... 327
Master and worker nodes ... 330
Pods ... 331
The pod life cycle ... 333
Working with pods ... 335
ReplicaSets and Deployments ... 341
Working with a ReplicaSet ... 342
Working with a Deployment ... 346
Services ... 350

Reconciling the desired state ... 312
Replicating and global services ... 313
Enabling service discovery ... 314
Providing routing services ... 314
Distributing work by load balancing ... 315
Dealing with fluctuating load by scaling ... 315
Facilitating self-healing ... 316
Enabling zero-downtime deployments ... 317
Providing affinity and location awareness ... 318
Security ... 319
Secure communication and cryptographic node identity ... 319
Secure networks and network policies ... 320
Role base access control (RBAC) ... 320
Secrets ... 320
Content trust ... 321
Reverse uptime ... 321
Facilitating introspection ... 322
Overview of popular orchestrators ... 323
Kubernetes ... 323
Docker Swarm ... 324
Apache Mesos and Marathon ... 325
Amazon AWS ECS ... 326
Microsoft Azure ACS ... 327
Understanding the architecture of Kubernetes ... 327
Master and worker nodes ... 330
Pods ... 331
The pod life cycle ... 333
Working with pods ... 335
ReplicaSets and Deployments ... 341
Working with a ReplicaSet ... 342
Working with a Deployment ... 346
Services ... 350

Context-based routing 354
Leveraging managed Kubernetes offerings in the cloud 355
Amazon AWS Elastic Kubernetes Service 356
Microsoft Azure Kubernetes Service 360
Google GCP 364
Discussing containers in the context of serverless 367
Conclusion 368
Questions 368
Answers 369
Job Interview sample questions 370
Possible answers to interview questions 370

11. Leveraging Docker Logs to Provide Insight into Your Apps 373
Introduction 373
Structure 373
Objectives 374
Monitoring your platform 374
Log entries 374
Centrally collection logs 375
Metrics 376
Dashboards 377
Alerts 378
Run books 379
Collecting container logs 379
Scraping container metrics using Prometheus 383
Monitoring Docker metrics with cAdvisor 383
Monitoring application key using Grafana 388
Defining alerts 393
Finding the root cause of an issue 394
Conclusion 396
Job interview questions 396
Possible answers to the job interview questions 396

Context-based routing 354
Leveraging managed Kubernetes offerings in the cloud 355
Amazon AWS Elastic Kubernetes Service 356
Microsoft Azure Kubernetes Service 360
Google GCP 364
Discussing containers in the context of serverless 367
Conclusion 368
Questions 368
Answers 369
Job Interview sample questions 370
Possible answers to interview questions 370

11. Leveraging Docker Logs to Provide Insight into Your Apps 373
Introduction 373
Structure 373
Objectives 374
Monitoring your platform 374
Log entries 374
Centrally collection logs 375
Metrics 376
Dashboards 377
Alerts 378
Run books 379
Collecting container logs 379
Scraping container metrics using Prometheus 383
Monitoring Docker metrics with cAdvisor 383
Monitoring application key using Grafana 388
Defining alerts 393
Finding the root cause of an issue 394
Conclusion 396
Job interview questions 396
Possible answers to the job interview questions 396

Questions 398
Answers *398*

12. Enabling Zero Downtime Deployments 401
Introduction 401
Structure 401
Objectives 402
Establishing a sound versioning strategy 402
Using rolling updates and rollbacks 404
Enabling blue-green deployments 414
Realizing canary releases 422
Canary releases with Gloo 426
Conclusion 434
References 434
Questions 435
Answers *435*
Job Interview sample questions 436
Possible answers to interview questions 436

13. Securing Containers 439
Introduction 439
Structure 440
Objectives 440
Using Kubernetes secrets 440
Explaining Kubernetes secrets *440*
Declaratively defining secrets *441*
Imperatively defining secrets *443*
Using secrets in a pod 445
Mapping secrets to the filesystem *446*
Secrets and environment variables *448*
Trusting containers 450
Scanning container images for vulnerabilities 451
Firewalling apps running in containers 454

Questions 398
Answers *398*

12. Enabling Zero Downtime Deployments 401
Introduction 401
Structure 401
Objectives 402
Establishing a sound versioning strategy 402
Using rolling updates and rollbacks 404
Enabling blue-green deployments 414
Realizing canary releases 422
Canary releases with Gloo 426
Conclusion 434
References 434
Questions 435
Answers *435*
Job Interview sample questions 436
Possible answers to interview questions 436

13. Securing Containers 439
Introduction 439
Structure 440
Objectives 440
Using Kubernetes secrets 440
Explaining Kubernetes secrets *440*
Declaratively defining secrets *441*
Imperatively defining secrets *443*
Using secrets in a pod 445
Mapping secrets to the filesystem *446*
Secrets and environment variables *448*
Trusting containers 450
Scanning container images for vulnerabilities 451
Firewalling apps running in containers 454

Software-defined networks 454
Port and protocol mappings 455
Ingress and egress rules 455
Docker security best practices 457
Check your images 457
Never run as privileged 457
Only allow read access to a root file system 458
Read-only filesystem on Kubernetes 458
Running a plain Docker container 461
Scan for vulnerabilities and secrets 461
Scanning for vulnerabilities 461
Do not expose the Docker daemon socket 462
Network namespace of the host 462
Conclusion 463
Job interview questions 463
Possible answers to the job interview questions 464
Questions 466
Answers 466

Index **469-476**

Software-defined networks 454
Port and protocol mappings 455
Ingress and egress rules 455
Docker security best practices 457
Check your images 457
Never run as privileged 457
Only allow read access to a root file system 458
Read-only filesystem on Kubernetes 458
Running a plain Docker container 461
Scan for vulnerabilities and secrets 461
Scanning for vulnerabilities 461
Do not expose the Docker daemon socket 462
Network namespace of the host 462
Conclusion 463
Job interview questions 463
Possible answers to the job interview questions 464
Questions 466
Answers 466

Index **469-476**

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

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:

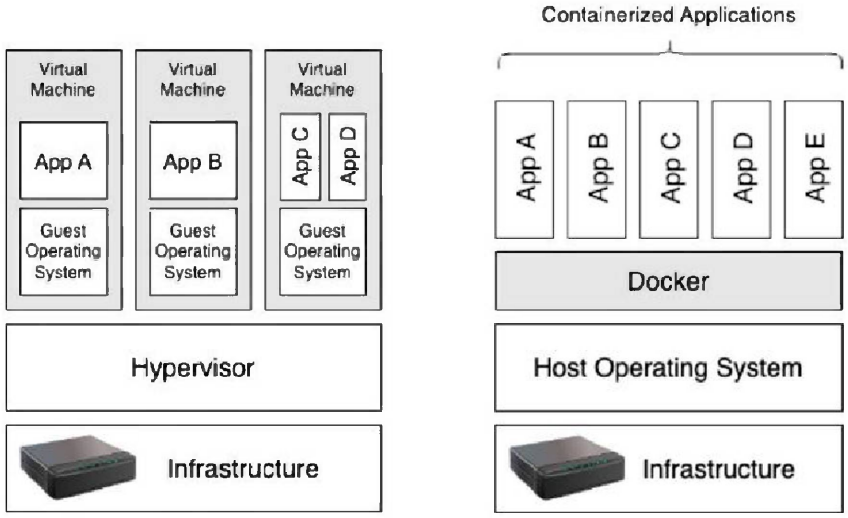


***Figure 1.1:** Virtual machines (VMs) versus Docker containers*

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:

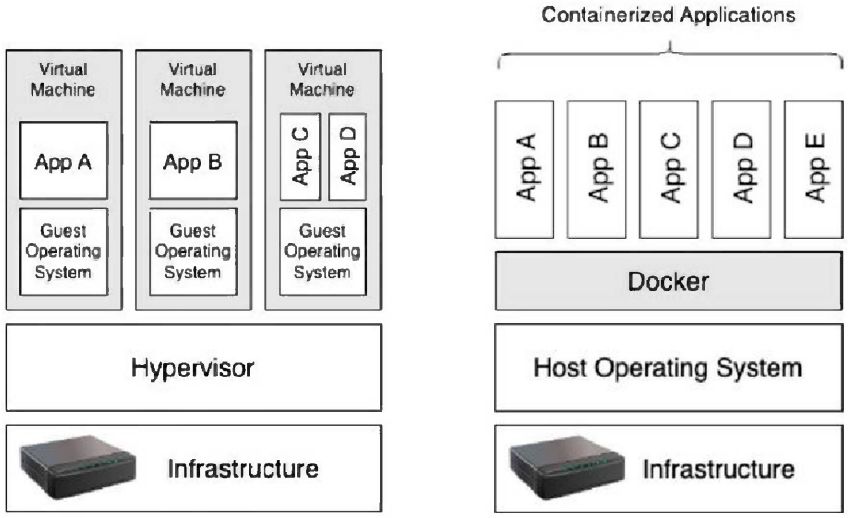


***Figure 1.1:** Virtual machines (VMs) versus Docker containers*

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

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.

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.

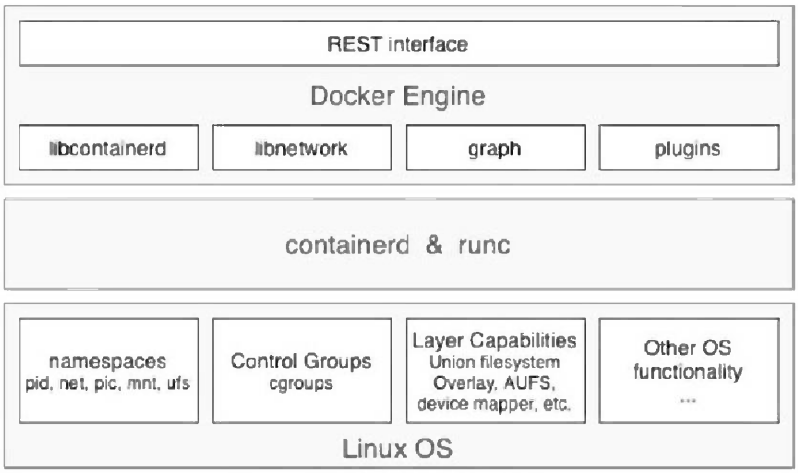


***Figure 1.2**: Docker high-level architecture*

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

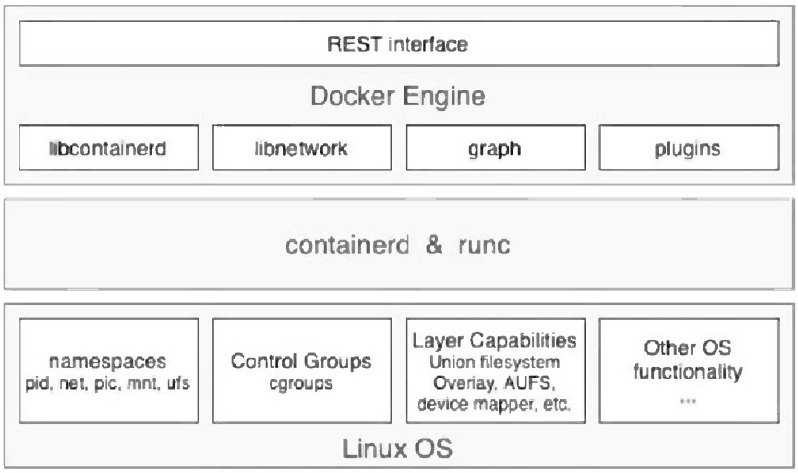


***Figure 1.2**: Docker high-level architecture*

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.

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

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

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.

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

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*.

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.

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

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

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

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

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:

   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

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

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



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.

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.

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.

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.**

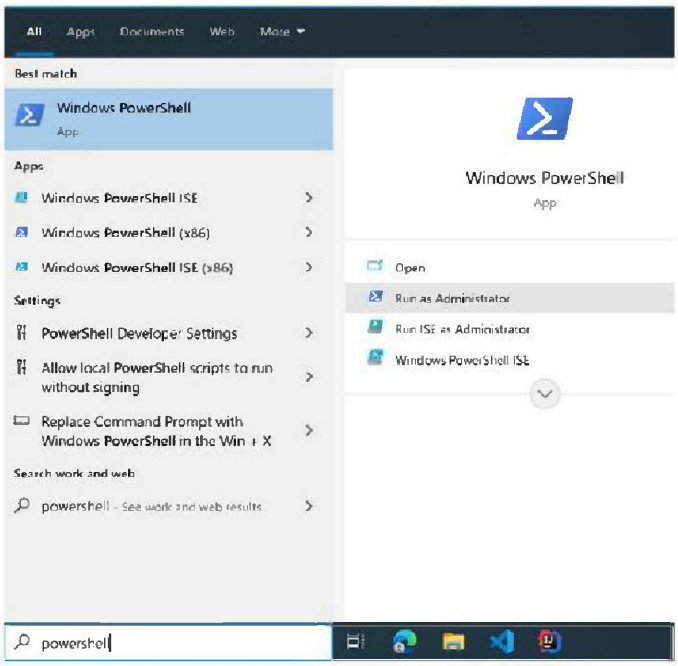


Figure 2.1: Starting PowerShell in Admin mode

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.**

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.**

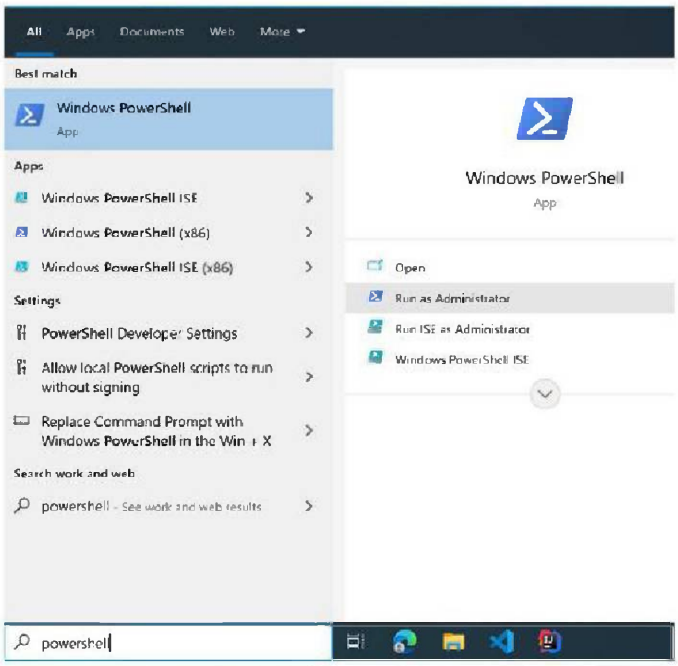


Figure 2.1: Starting PowerShell in Admin mode

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`

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

```
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

2. Install Docker Desktop with Chocolatey using this command:

   ```
   choco install docker-desktop
   ```

   After having successfully installed the application, make sure it is running. If it is not already running, press the Windows key and type **Docker desktop** into the search bar, select the application in the list, and press `Enter`.

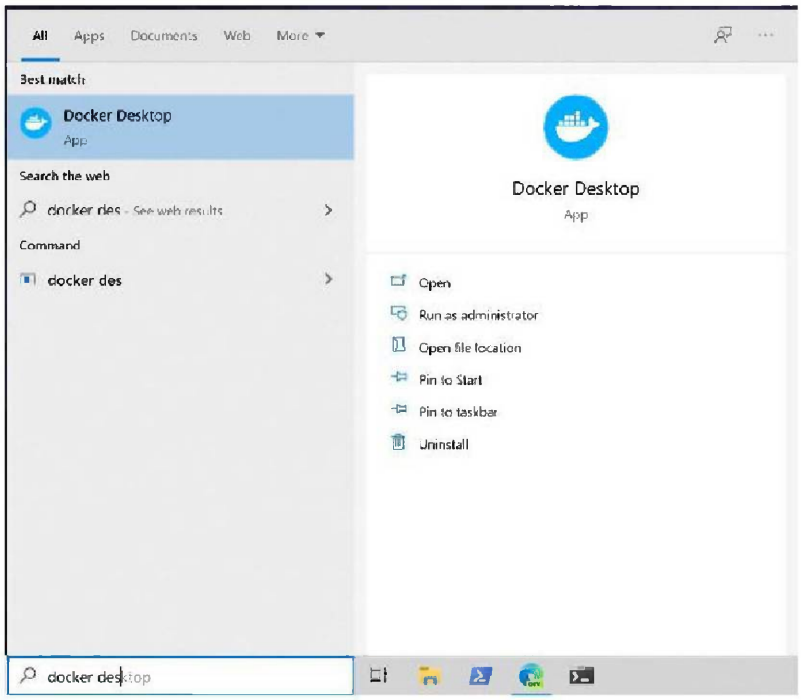


Figure 2.2: Starting Docker desktop on Windows 10

Locate the Whale icon in your system tray, indicating that Docker is now running

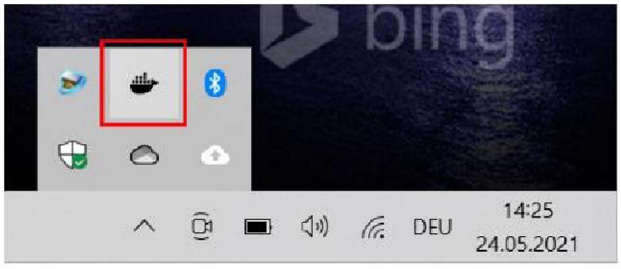


Figure 2.3: Docker icon in the Windows 10 system tray

2. Install Docker Desktop with Chocolatey using this command:

   ```
   choco install docker-desktop
   ```

   After having successfully installed the application, make sure it is running. If it is not already running, press the Windows key and type **Docker desktop** into the search bar, select the application in the list, and press `Enter`.

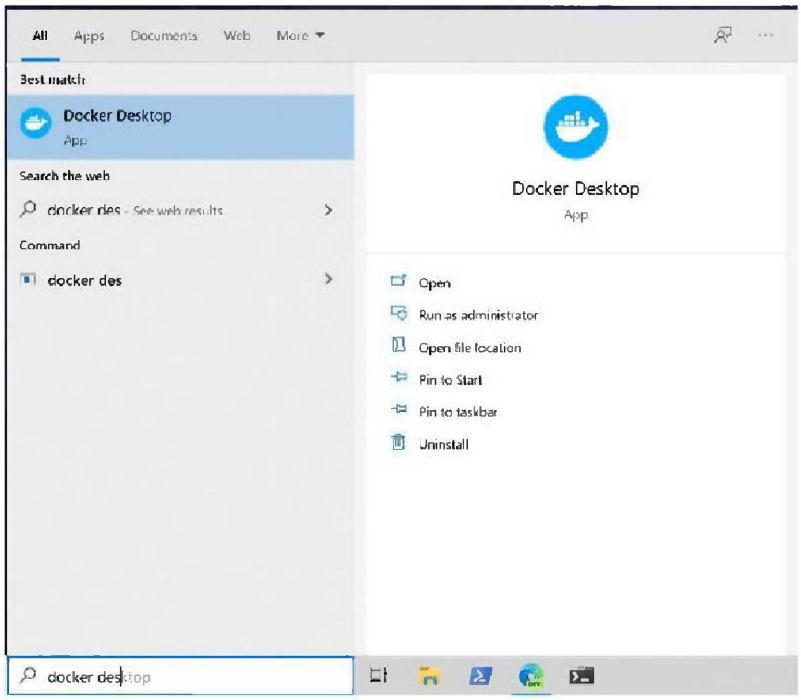


Figure 2.2: Starting Docker desktop on Windows 10

Locate the Whale icon in your system tray, indicating that Docker is now running

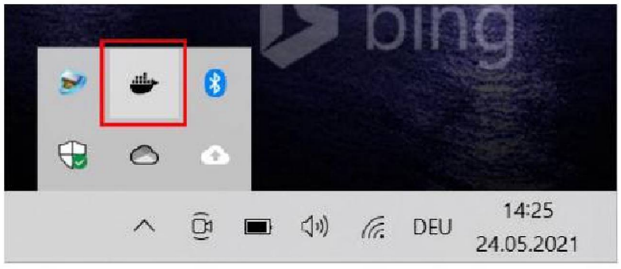


Figure 2.3: Docker icon in the Windows 10 system tray

And click the icon. The Docker Desktop dashboard will be displayed.

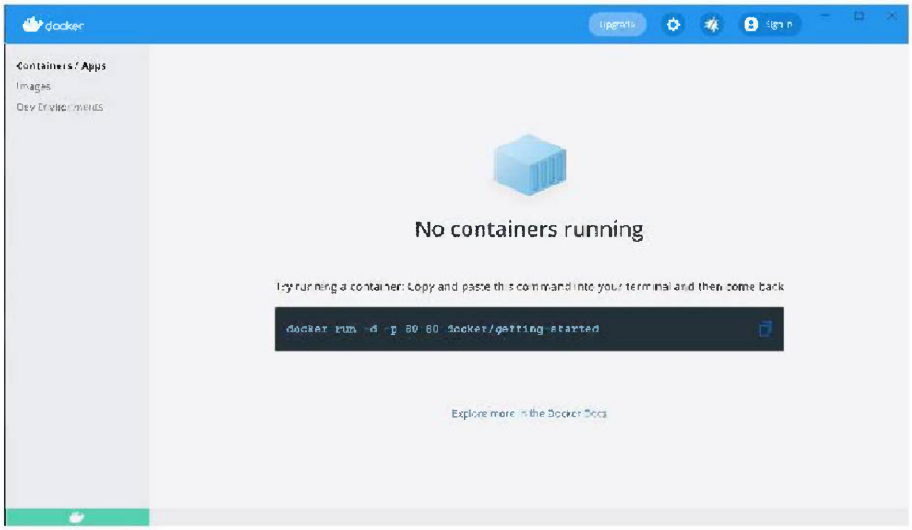


Figure 2.4: Dashboard of the Docker desktop application

3. To finish our installation of Docker, we need to also install the **docker-compose** tool that is not included with Docker Desktop by default. In your terminal, execute the following:

   ```
   choco install docker-compose
   ```

4. After completing the installation test, Docker-compose with the following command:

   ```
   docker-compose --version
   ```

   If everything went well, you should see something along this:

   ```
   gabriel@Solaris ~ % docker-compose --version
   docker-compose version 1.29.2, build unknown
   ```

Figure 2.5: Testing Docker-compose

Now that Docker Desktop is up and running on your Windows 10 or 11 computer, it is time to start exploring the application with a few basic activities. Please proceed to section **First steps with Docker Desktop**.

Installing Docker Desktop on MacOS

To install Docker Desktop (or short Docker) on your MacOS, please follow the following steps:

And click the icon. The Docker Desktop dashboard will be displayed.

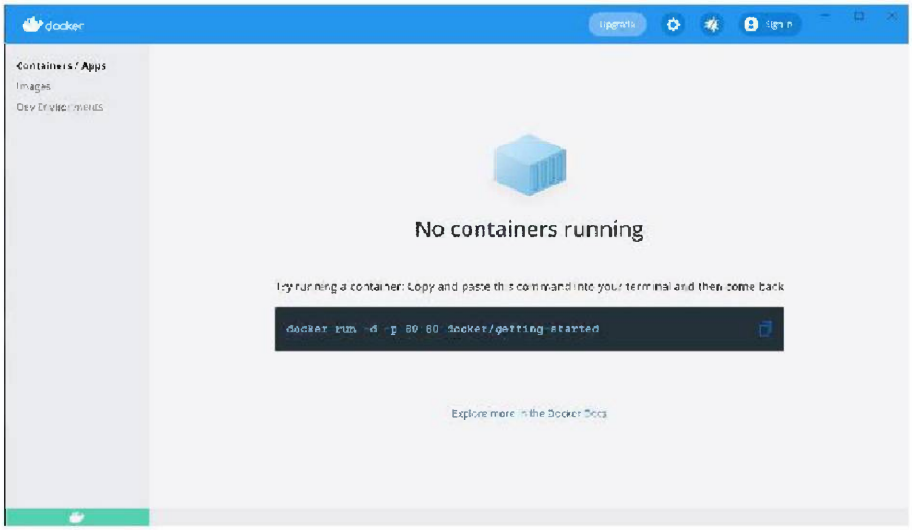


Figure 2.4: Dashboard of the Docker desktop application

3. To finish our installation of Docker, we need to also install the **docker-compose** tool that is not included with Docker Desktop by default. In your terminal, execute the following:

   ```
   choco install docker-compose
   ```

4. After completing the installation test, Docker-compose with the following command:

   ```
   docker-compose --version
   ```

   If everything went well, you should see something along this:

   ```
   gabriel@Solaris ~ % docker-compose --version
   docker-compose version 1.29.2, build unknown
   ```

Figure 2.5: Testing Docker-compose

Now that Docker Desktop is up and running on your Windows 10 or 11 computer, it is time to start exploring the application with a few basic activities. Please proceed to section **First steps with Docker Desktop**.

Installing Docker Desktop on MacOS

To install Docker Desktop (or short Docker) on your MacOS, please follow the following steps:

1. Open your terminal.
2. If you are on a new Mac with the M1 chip, then you need to install Rosetta first with the following command:

   ```
   softwareupdate --install-rosetta
   ```

3. In a new browser, tab navigate to **https://docs.docker.com/docker-for-mac/install/**
4. Select the appropriate CPU; in my case, having a new MacBook Air M1, it is the Apple silicon.
5. Install Docker Desktop following the instructions.
6. Once installed, run Docker, e.g., by using spotlight (press ⌘-*SPACE*).



Figure 2.6: Starting Docker Desktop with Apple spotlight

You should see a whale icon on the upper right of the screen.

1. Click on the whale icon to display the context menu of Docker:

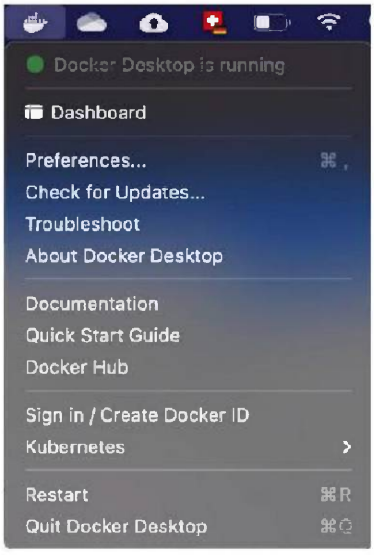


Figure 2.7: Docker Desktop context menu

1. Open your terminal.
2. If you are on a new Mac with the M1 chip, then you need to install Rosetta first with the following command:

   ```
   softwareupdate --install-rosetta
   ```

3. In a new browser, tab navigate to **https://docs.docker.com/docker-for-mac/install/**
4. Select the appropriate CPU; in my case, having a new MacBook Air M1, it is the Apple silicon.
5. Install Docker Desktop following the instructions.
6. Once installed, run Docker, e.g., by using spotlight (press ⌘-*SPACE*).



Figure 2.6: Starting Docker Desktop with Apple spotlight

You should see a whale icon on the upper right of the screen.

1. Click on the whale icon to display the context menu of Docker:

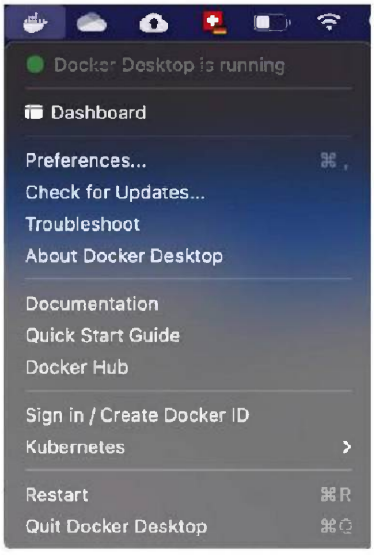


Figure 2.7: Docker Desktop context menu

2. To finish our installation of Docker, we need to also install the Docker-compose tool that is not included with Docker Desktop by default. In your terminal, execute the following:

   ```
   brew install docker-compose
   ```

3. After completing the installation test **docker-compose** with this command:

   ```
   docker-compose --version
   ```

   If everything went well, you should see something along this:

   ```
   gabriel@Solaris ~ % docker-compose --version
   docker-compose version 1.29.2, build unknown
   ```

Figure 2.8: Testing Docker-compose

Now that Docker Desktop is up and running on our computer, it is time to start exploring it with a few basic activities.

Using Docker Desktop to execute basic container commands

Now that we have successfully installed Docker Desktop on either Windows 10 or 11 MacOS, it is time to do a few initial experiments with it and verify that it is working as expected:

1. Open a terminal window.
2. Run the following command in the terminal

   ```
   docker run -d -p 80:80 docker/getting-started
   ```

   This is what you will see:

```
gabriel@Solaris ~ % docker run -d -p 80:80 docker/getting-started
Unable to find image 'docker/getting-started:latest' locally
latest: Pulling from docker/getting-started
060e56d6d07f: Pull complete
2c36e9465b53: Pull complete
cfcc925340bd: Pull complete
5280a9b3efa9: Pull complete
83380a3f4dde: Pull complete
uedf4cu77due: Pull complete
e45379b4b21e: Pull complete
45643c41063c: Pull complete
Digest: sha256:67944b53f8a7d16b3c9900315f9d557ade12c4ee4083cd98068f9fe6d9995808
Status: Downloaded newer image for docker/getting-started:latest
3c13a2c8fcb17c1e903c1384cuc3639lf0uedb63rcdf6eud5e813fue0bb05bu4
gabriel@Solaris ~ %
```

Figure 2.9: Running the Docker hello world

2. To finish our installation of Docker, we need to also install the Docker-compose tool that is not included with Docker Desktop by default. In your terminal, execute the following:

   ```
   brew install docker-compose
   ```

3. After completing the installation test **docker-compose** with this command:

   ```
   docker-compose --version
   ```

   If everything went well, you should see something along this:

   ```
   gabriel@Solaris ~ % docker-compose --version
   docker-compose version 1.29.2, build unknown
   ```

Figure 2.8: Testing Docker-compose

Now that Docker Desktop is up and running on our computer, it is time to start exploring it with a few basic activities.

Using Docker Desktop to execute basic container commands

Now that we have successfully installed Docker Desktop on either Windows 10 or 11 MacOS, it is time to do a few initial experiments with it and verify that it is working as expected:

1. Open a terminal window.
2. Run the following command in the terminal

   ```
   docker run -d -p 80:80 docker/getting-started
   ```

   This is what you will see:

```
gabriel@Solaris ~ % docker run -d -p 80:80 docker/getting-started
Unable to find image 'docker/getting-started:latest' locally
latest: Pulling from docker/getting-started
060e56d6d07f: Pull complete
2c36e9465b53: Pull complete
cfcc925340bd: Pull complete
5280a9b3efa9: Pull complete
83380a3f4dde: Pull complete
uedf4cu77due: Pull complete
e45379b4b21e: Pull complete
45643c41063c: Pull complete
Digest: sha256:67944b53f8a7d16b3c9900315f9d557ade12c4ee4083cd98068f9fe6d9995808
Status: Downloaded newer image for docker/getting-started:latest
3c13a2c8fcb17c1e903c1384cuc3639lf0uedb63rcdf6eud5e813fue0bb05bu4
gabriel@Solaris ~ %
```

Figure 2.9: Running the Docker hello world

Do not worry about the details yet, as we will explain everything in *Chapter 3, Getting familiar with containers.*

3. Make sure the container runs:

```
docker container ls
```

You should see an output similar to this:



Figure 2.10: Listing all running containers

As you can see, Docker lists one container using the image called **`docker/getting-started`** and having the status `Up 3 minutes`, among other information.

4. Open a browser tab and navigate to **http://localhost**

You should see a **Getting started** Web page.

5. Get the version information of the Docker client and server with the command:

```
docker version
```

That is what you should see:

```
gabriel@Solaris ~ % docker version
Client:
 Cloud integration: 1.0.14
 Version:           20.10.6
 API version:       1.41
 Go version:        go1.16.3
 Git commit:        370c289
 Built:             Fri Apr  9 22:46:57 2021
 OS/Arch:           darwin/arm64
 Context:           default
 Experimental:      true

Server: Docker Engine - Community
 Engine:
  Version:          20.10.6
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.13.15
  Git commit:       8728dd2
  Built:            Fri Apr  9 22:44:13 2021
  OS/Arch:          linux/arm64
  Experimental:     false
 containerd:
  Version:          1.4.4
  GitCommit:        05f951a3781f4f2c1911b05e61c160e9c30eaa8e
 runc:
  Version:          1.0.0-rc93
  GitCommit:        12644e614e25b05da6fd08a38ffa0cfe1903fdec
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0
```

Figure 2.11: Displaying the version information of Docker

Do not worry about the details yet, as we will explain everything in *Chapter 3, Getting familiar with containers.*

3. Make sure the container runs:

```
docker container ls
```

You should see an output similar to this:



Figure 2.10: Listing all running containers

As you can see, Docker lists one container using the image called **`docker/getting-started`** and having the status `Up 3 minutes`, among other information.

4. Open a browser tab and navigate to **http://localhost**

You should see a **Getting started** Web page.

5. Get the version information of the Docker client and server with the command:

```
docker version
```

That is what you should see:

```
gabriel@Solaris ~ % docker version
Client:
 Cloud integration: 1.0.14
 Version:           20.10.6
 API version:       1.41
 Go version:        go1.16.3
 Git commit:        370c289
 Built:             Fri Apr  9 22:46:57 2021
 OS/Arch:           darwin/arm64
 Context:           default
 Experimental:      true

Server: Docker Engine - Community
 Engine:
  Version:          20.10.6
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.13.15
  Git commit:       8728dd2
  Built:            Fri Apr  9 22:44:13 2021
  OS/Arch:          linux/arm64
  Experimental:     false
 containerd:
  Version:          1.4.4
  GitCommit:        05f951a3781f4f2c1911b05e61c160e9c30eaa8e
 runc:
  Version:          1.0.0-rc93
  GitCommit:        12644e614e25b05da6fd08a38ffa0cfe1903fdec
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0
```

Figure 2.11: Displaying the version information of Docker

You are now all setup and ready to start working with Docker containers.

In the next section, we will be configuring Docker Desktop.

Configuring Docker Desktop

Web developers, DevOps, and operations engineers these days work with quite powerful personal computers or laptops. At the same time, the applications that we develop are more and more resource hungry. By default, Docker Desktop comes configured with quite conservative settings. We may want to tune those values, specifically the number of CPUs used and the memory consumed.

To alter the configuration of Docker Desktop, proceed as follows:

1. Click on the Docker whale icon and select **Dashboard** on your Mac or just click on the whale icon in the system tray of your Windows 10 or 11 machine to display the dashboard of Docker Desktop.
2. On the dashboard, select the settings (gear) icon, and in the **Preferences** popup, select **Resources**:

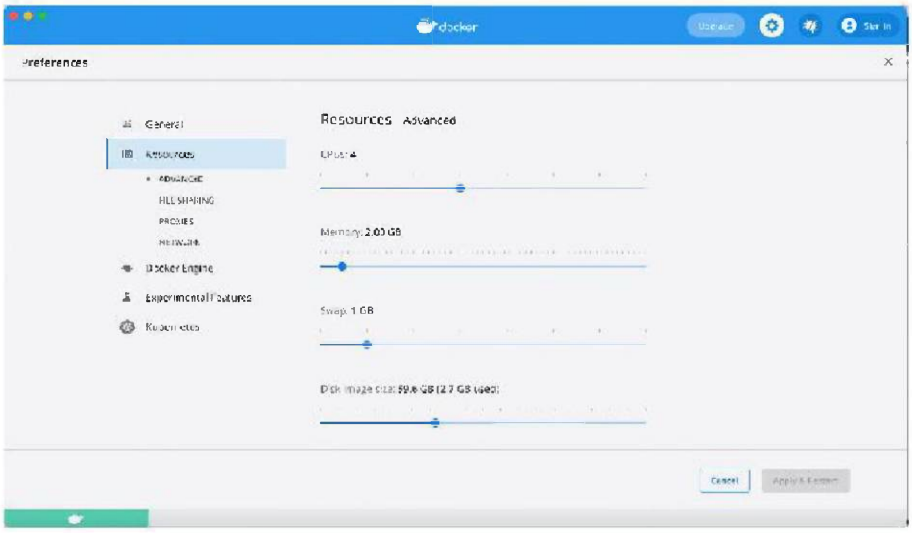


***Figure 2.12**: Resource settings for Docker*

3. Depending on the resources of your working machine, bump up the values for CPU and memory. It is recommended to use 8 GB memory on a machine with 16 GB RAM.

You are now all setup and ready to start working with Docker containers.

In the next section, we will be configuring Docker Desktop.

Configuring Docker Desktop

Web developers, DevOps, and operations engineers these days work with quite powerful personal computers or laptops. At the same time, the applications that we develop are more and more resource hungry. By default, Docker Desktop comes configured with quite conservative settings. We may want to tune those values, specifically the number of CPUs used and the memory consumed.

To alter the configuration of Docker Desktop, proceed as follows:

1. Click on the Docker whale icon and select **Dashboard** on your Mac or just click on the whale icon in the system tray of your Windows 10 or 11 machine to display the dashboard of Docker Desktop.
2. On the dashboard, select the settings (gear) icon, and in the **Preferences** popup, select **Resources**:

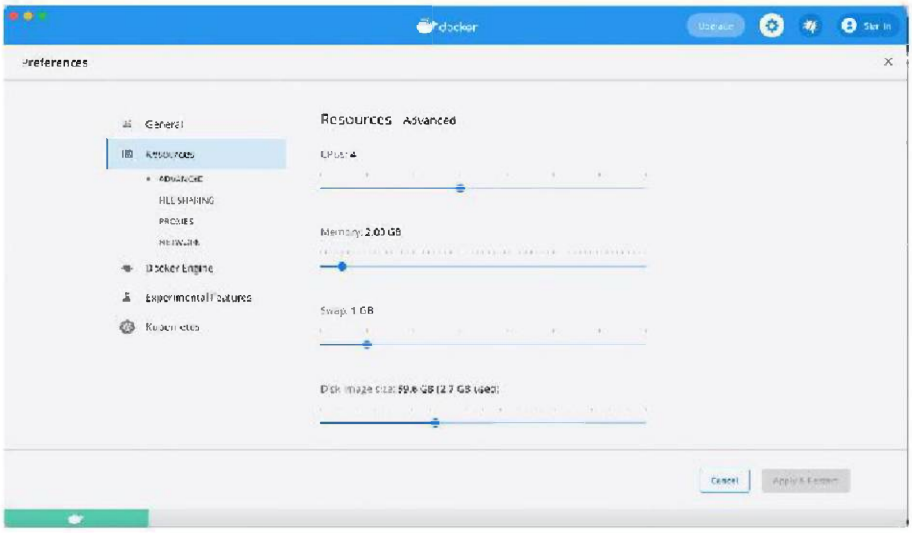


***Figure 2.12**: Resource settings for Docker*

3. Depending on the resources of your working machine, bump up the values for CPU and memory. It is recommended to use 8 GB memory on a machine with 16 GB RAM.

Sooner or later, you will want to work with Kubernetes. Docker Desktop supports running a single node Kubernetes cluster for development purposes. This comes in very handy at the moment you want to test the deployment and operation of your app to and on Kubernetes. By default, Kubernetes support in Docker Desktop is turned off. To change this setting, do the following:

1. Open the Docker Desktop dashboard.
2. On the **Preferences** popup, select the settings (gear) button
3. Select the **Kubernetes** tab and check the **Kubernetes** checkbox

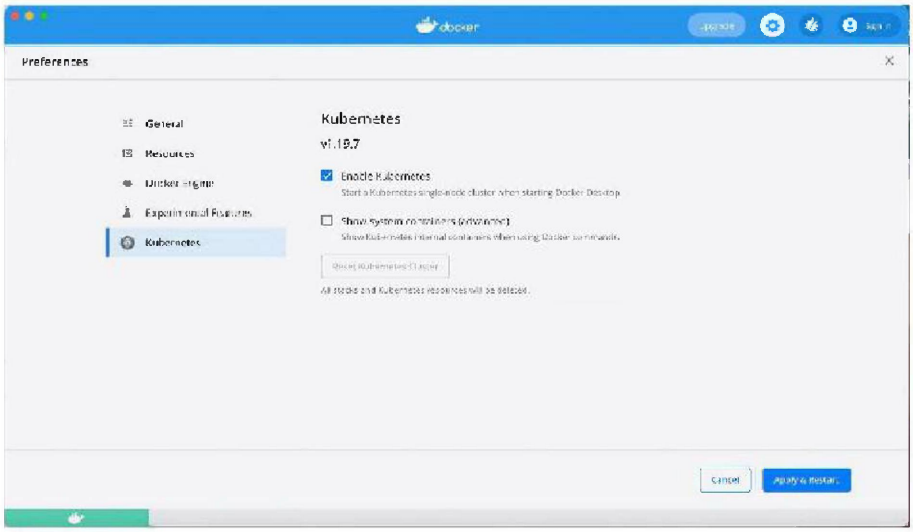


Figure 2.13: Enable Kubernetes on Docker desktop

4. Only check the other checkbox called **Show system container** if you need to debug Kubernetes.
5. Click the **Apply & Restart** button to restart Docker Desktop. This will take a moment since Docker Desktop needs to download a list of container images containing the whole Kubernetes infrastructure and then run it.

Having installed and configured the basic Docker tools, we now need a good code editor. That is what we are going to install and configure next.

Selecting a code editor and useful plugins

Throughout this book, we will author and use many scripts, Dockerfiles, Docker-compose files, and even whole application projects. It is, therefore, wise to use a

Sooner or later, you will want to work with Kubernetes. Docker Desktop supports running a single node Kubernetes cluster for development purposes. This comes in very handy at the moment you want to test the deployment and operation of your app to and on Kubernetes. By default, Kubernetes support in Docker Desktop is turned off. To change this setting, do the following:

1. Open the Docker Desktop dashboard.
2. On the **Preferences** popup, select the settings (gear) button
3. Select the **Kubernetes** tab and check the **Kubernetes** checkbox

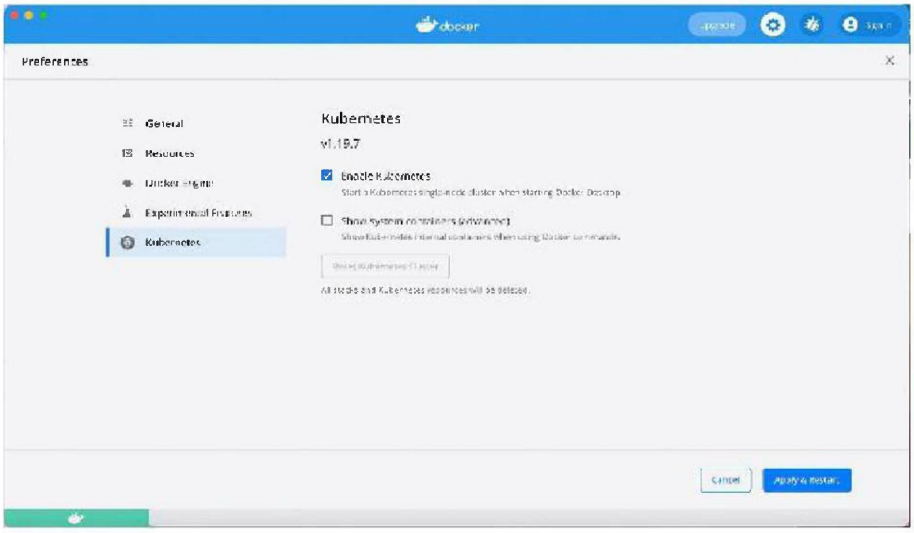


Figure 2.13: Enable Kubernetes on Docker desktop

4. Only check the other checkbox called **Show system container** if you need to debug Kubernetes.
5. Click the **Apply & Restart** button to restart Docker Desktop. This will take a moment since Docker Desktop needs to download a list of container images containing the whole Kubernetes infrastructure and then run it.

Having installed and configured the basic Docker tools, we now need a good code editor. That is what we are going to install and configure next.

Selecting a code editor and useful plugins

Throughout this book, we will author and use many scripts, Dockerfiles, Docker-compose files, and even whole application projects. It is, therefore, wise to use a

powerful yet simple to use code editor. There are, of course, many possibilities, from very basic to extremely powerful and heavy-weight ones. Our selection falls somewhere in the middle. For this book, we have selected Visual Studio Code based on the following reasons. It is:

- Completely free
- Very lightweight
- Available for all major platforms, Windows 10 and 11, MacOS, and Linux
- Infinitely extendable through plugins
- The most beloved editor among developers by a wide margin (**https://insights.stackoverflow.com/survey/2020**)

TIP: You can get more information about Visual Studio Code (also short VS Code) from here: https://code.visualstudio.com.

Installing VS Code on Windows 10

The advantage of having installed a powerful package manager such as Chocolatey makes it very easy for us to install additional software packages. Visual Studio Code is the first example. To install VS Code, please proceed as follows:

1. Open a PowerShell window in Admin mode.
2. In this window, execute the following command:

   ```
   choco install -y vscode
   ```

   This will install the latest version of VS Code on your machine. Depending on your network speed, this may take a minute or two.
3. Once the application is installed, test it by executing
   code .

   In your PowerShell window. Please note the point (.) at the end of the command. VS code should start and have the current folder opened as a working folder.

Once you got your VS Code editor running, you can move on to the section **Installing VS Code Extensions**.

Installing VS Code on MacOS

Since we have installed Homebrew as our package manager, it is now very easy to install any further software on our Mac. To install VS Code, proceed as follows:

1. Open your terminal.

powerful yet simple to use code editor. There are, of course, many possibilities, from very basic to extremely powerful and heavy-weight ones. Our selection falls somewhere in the middle. For this book, we have selected Visual Studio Code based on the following reasons. It is:

- Completely free
- Very lightweight
- Available for all major platforms, Windows 10 and 11, MacOS, and Linux
- Infinitely extendable through plugins
- The most beloved editor among developers by a wide margin (**https://insights.stackoverflow.com/survey/2020**)

TIP: You can get more information about Visual Studio Code (also short VS Code) from here: https://code.visualstudio.com.

Installing VS Code on Windows 10

The advantage of having installed a powerful package manager such as Chocolatey makes it very easy for us to install additional software packages. Visual Studio Code is the first example. To install VS Code, please proceed as follows:

1. Open a PowerShell window in Admin mode.
2. In this window, execute the following command:

   ```
   choco install -y vscode
   ```

   This will install the latest version of VS Code on your machine. Depending on your network speed, this may take a minute or two.
3. Once the application is installed, test it by executing
   code .

   In your PowerShell window. Please note the point (.) at the end of the command. VS code should start and have the current folder opened as a working folder.

Once you got your VS Code editor running, you can move on to the section **Installing VS Code Extensions**.

Installing VS Code on MacOS

Since we have installed Homebrew as our package manager, it is now very easy to install any further software on our Mac. To install VS Code, proceed as follows:

1. Open your terminal.

2. Execute this command

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

3. Test the installation by executing `code .`

   In your terminal. Please note the point (.) at the end of the command. VS Code should start and have the current folder opened as a working folder.

Now that we have installed the editor on our machine let us make it a bit more versatile by adding a few powerful extensions to it.

Installing VS Code extensions

VS Code is so powerful due to the infinite number of extensions you can install for it. Let us install a few that are useful for our purpose. We will start with the extension for Docker. This extension will make working with Docker containers extremely easy and pleasant from within the editor. Proceed as follows:

1. Start VS Code.
2. On the left side of the editor, select the extension sidebar (or hit *SHIFT-CTRL-X* on Windows or ⇧-⌘-X on MacOS)

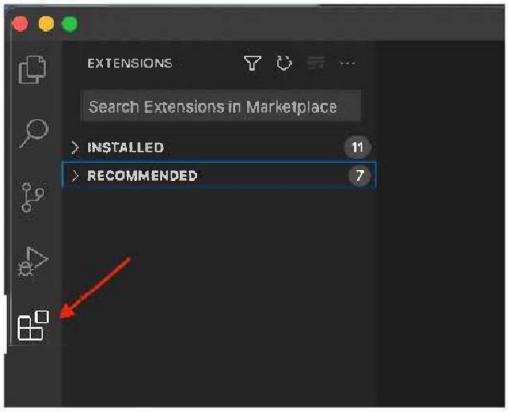


Figure 2.14: Visual studio code extensions sidebar

3. In the search box, enter `docker`.

2. Execute this command

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

3. Test the installation by executing `code .`

   In your terminal. Please note the point (.) at the end of the command. VS Code should start and have the current folder opened as a working folder.

Now that we have installed the editor on our machine let us make it a bit more versatile by adding a few powerful extensions to it.

Installing VS Code extensions

VS Code is so powerful due to the infinite number of extensions you can install for it. Let us install a few that are useful for our purpose. We will start with the extension for Docker. This extension will make working with Docker containers extremely easy and pleasant from within the editor. Proceed as follows:

1. Start VS Code.
2. On the left side of the editor, select the extension sidebar (or hit *SHIFT-CTRL-X* on Windows or ⇧-⌘-X on MacOS)

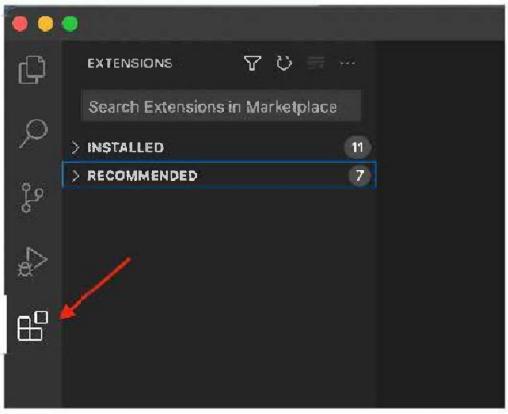


Figure 2.14: Visual studio code extensions sidebar

3. In the search box, enter `docker`.

4. Select the (recommended) extension called **Docker**, that is provided by Microsoft:



Figure 2.15: Docker extension for Visual Studio Code

5. Click **Install** to install the extension.

Let us install another useful extension. This time it will be a set of extensions that are useful when you develop Java applications. Still, from within VS Code, do the following:

1. Open the extension's sidebar with *SHIFT-CTRL-X* (or ⇧-⌘-X).
2. In the search box, enter `Java`.
3. Select the **Java Extension Pack** by Microsoft and click **Install** to install the various plugins that belong to this package.

Feel free to install as many extensions as you find useful for your daily work.

An alternative, more automatable way of installing VS code extensions is from the command line. Use the following command to install an extension:

```
code -install-extension <extension-name>
```

where `<extension-name>` is the name of the extension you are trying to install. This is equally valid on Windows 10 or 11 and on MacOS. For more details, please refer to the excellent online documentation, which you can find here: **https://code.visualstudio.com/docs/editor/extension-marketplace#_command-line-extension-management.**

Now that we have a code powerful editor, let us install and configure a good terminal.

Configuring and using a great terminal

Depending on which platform you are running, we can highly recommend two terminals. On Windows 10 and 11, it is the **Windows Terminal**, and on MacOS, it is iTerm3. Let us start with the former.

4. Select the (recommended) extension called **Docker**, that is provided by Microsoft:



Figure 2.15: Docker extension for Visual Studio Code

5. Click **Install** to install the extension.

Let us install another useful extension. This time it will be a set of extensions that are useful when you develop Java applications. Still, from within VS Code, do the following:

1. Open the extension's sidebar with *SHIFT-CTRL-X* (or ⇧-⌘-X).
2. In the search box, enter `Java`.
3. Select the **Java Extension Pack** by Microsoft and click **Install** to install the various plugins that belong to this package.

Feel free to install as many extensions as you find useful for your daily work.

An alternative, more automatable way of installing VS code extensions is from the command line. Use the following command to install an extension:

```
code -install-extension <extension-name>
```

where `<extension-name>` is the name of the extension you are trying to install. This is equally valid on Windows 10 or 11 and on MacOS. For more details, please refer to the excellent online documentation, which you can find here: **https://code.visualstudio.com/docs/editor/extension-marketplace#_command-line-extension-management.**

Now that we have a code powerful editor, let us install and configure a good terminal.

Configuring and using a great terminal

Depending on which platform you are running, we can highly recommend two terminals. On Windows 10 and 11, it is the **Windows Terminal**, and on MacOS, it is iTerm3. Let us start with the former.

Installing Windows Terminal

On Windows, you have plenty of choices when it comes to terminals. Recently though, Windows Terminal has become very popular and, at this moment, is by far the best option out there. You can swiftly install it by following the mentioned steps:

1. Open a PowerShell window in Admin mode.
2. Execute the following command:

   ```
   choco install microsoft-windows-terminal
   ```

   Note that this package requires at least Windows 10 version 1903/OS build 18362
3. You may want to personalize your terminal with nice fonts supporting ligatures, and a custom prompt. Scott Hanselmann from Microsoft has written a few helpful blog posts about it. Here is a good start: **https://www.hanselman.com/blog/take-your-windows-terminal-and-powershell-to-the-next-level-with-terminal-icons**

Now that we have learned how to install and configure a terminal for Windows 10 or 11 let us look into how to do the same on MacOS.

Installing ITerm2 Version 3 for MacOS

Although MacOS has an integrated terminal, it is highly recommended to use a more advanced terminal such as iTerm2 Version 3, which provides more advanced features like split panes and more. You can find additional information about this tool here: **https://iterm2.com/version3.html**.

1. Open a terminal window.
2. Install iTerm2 using this command:

   ```
   brew install --cask iterm2
   ```
3. Press ⌘-*SHIFT* to bring up the spotlight and enter `term` in the search box, and then press *Enter*. iTerm2 should be started.

The next section provides a quick overview of some cool and useful playgrounds for Docker and Kubernetes.

Experimenting online with Docker and Kubernetes

At times we do want to do some quick experiments with containers or with Kubernetes without having to first install a bunch of software on our computers. We

Installing Windows Terminal

On Windows, you have plenty of choices when it comes to terminals. Recently though, Windows Terminal has become very popular and, at this moment, is by far the best option out there. You can swiftly install it by following the mentioned steps:

1. Open a PowerShell window in Admin mode.
2. Execute the following command:

   ```
   choco install microsoft-windows-terminal
   ```

   Note that this package requires at least Windows 10 version 1903/OS build 18362
3. You may want to personalize your terminal with nice fonts supporting ligatures, and a custom prompt. Scott Hanselmann from Microsoft has written a few helpful blog posts about it. Here is a good start: **https://www.hanselman.com/blog/take-your-windows-terminal-and-powershell-to-the-next-level-with-terminal-icons**

Now that we have learned how to install and configure a terminal for Windows 10 or 11 let us look into how to do the same on MacOS.

Installing ITerm2 Version 3 for MacOS

Although MacOS has an integrated terminal, it is highly recommended to use a more advanced terminal such as iTerm2 Version 3, which provides more advanced features like split panes and more. You can find additional information about this tool here: **https://iterm2.com/version3.html**.

1. Open a terminal window.
2. Install iTerm2 using this command:

   ```
   brew install --cask iterm2
   ```
3. Press ⌘-*SHIFT* to bring up the spotlight and enter `term` in the search box, and then press *Enter*. iTerm2 should be started.

The next section provides a quick overview of some cool and useful playgrounds for Docker and Kubernetes.

Experimenting online with Docker and Kubernetes

At times we do want to do some quick experiments with containers or with Kubernetes without having to first install a bunch of software on our computers. We

would like to have a little playground that we can use with zero effort from our side. This is where Play with Docker and Play with Kubernetes come into play.

Play with Docker

A few years ago, the two Docker captains, Marcos Liljedhal and Jonathan Leibiusky, came up with the idea to provide a playground to all interested people that want to experiment with containers but did not have the time or knowledge to first setup their own environment. The company Docker found this idea compelling and has since then sponsored this project by providing the necessary infrastructure on top of which Play with Docker runs.

Let us explore this playground a bit:

1. Open a new browser tab and navigate to **https://labs.play-with-docker.com**
2. Login using your Docker account.

NOTE: If you do not yet have one, then please signup. It is free, and you will need a Docker account anyways for some of the exercises provided in this book in later chapters.

If you have difficulties to access this site, please make sure you turn off popup blockers for this site!

3. Orient yourself in the presented playground. It should look like this:

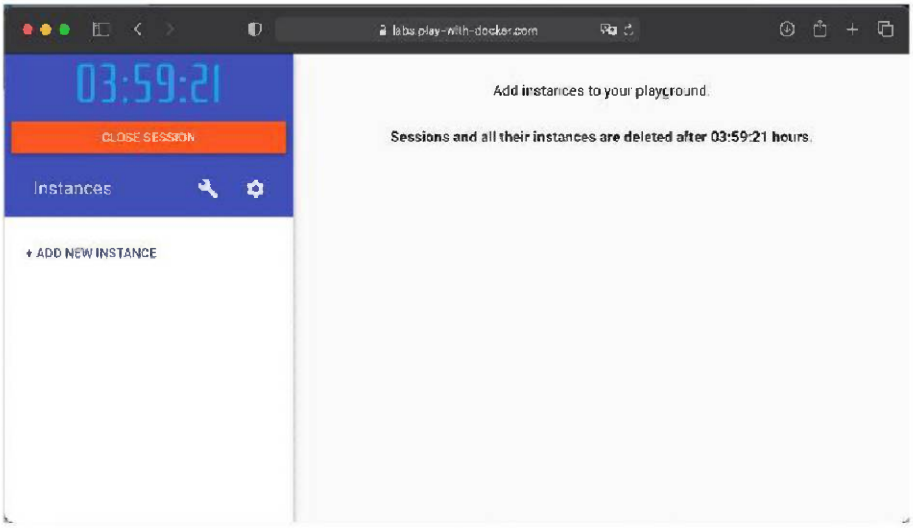


Figure 2.16: Play with Docker

would like to have a little playground that we can use with zero effort from our side. This is where Play with Docker and Play with Kubernetes come into play.

Play with Docker

A few years ago, the two Docker captains, Marcos Liljedhal and Jonathan Leibiusky, came up with the idea to provide a playground to all interested people that want to experiment with containers but did not have the time or knowledge to first setup their own environment. The company Docker found this idea compelling and has since then sponsored this project by providing the necessary infrastructure on top of which Play with Docker runs.

Let us explore this playground a bit:

1. Open a new browser tab and navigate to **https://labs.play-with-docker.com**
2. Login using your Docker account.

NOTE: If you do not yet have one, then please signup. It is free, and you will need a Docker account anyways for some of the exercises provided in this book in later chapters.

If you have difficulties to access this site, please make sure you turn off popup blockers for this site!

3. Orient yourself in the presented playground. It should look like this:

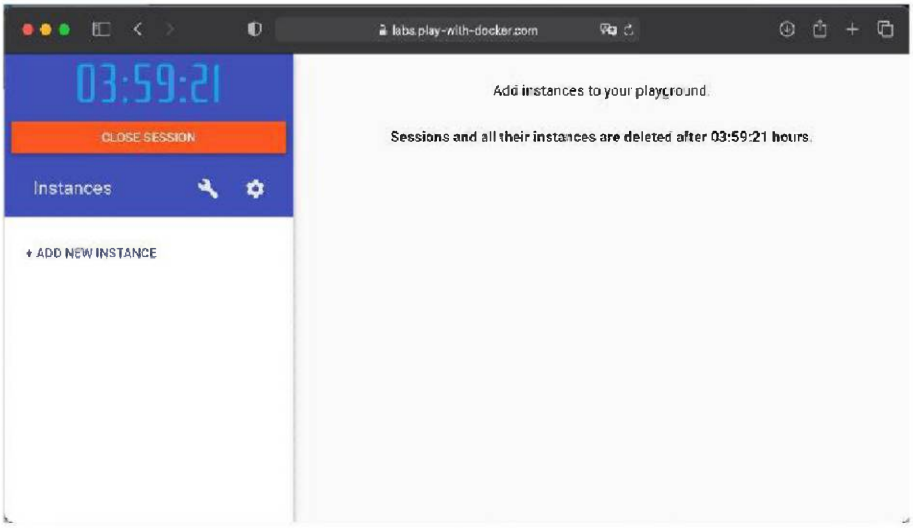


Figure 2.16: Play with Docker

In the upper left corner, you can see a clock that tells you how much time is left in the current session. Using the playground is free, but it is limited to 4 hours max per session. At any time, you can start a new session. Please also notice the **+ADD NEW INSTANCE** link on the left side. With it, you can create a new container host where you can play with containers. You can create up to four hosts.

4. Click on the **+ADD NEW INSTANCE** link to create a new container host. After a short time, a terminal and some additional information are displayed on the right side of the page as shown:



Figure 2.17: Running instance on Play with Docker

5. Inside the terminal, execute the hello world command that we already ran locally on our machine before:

```
docker run -d -p 80:80 docker/getting-started
```

6. Use `curl` to verify that the container is running:

```
curl localhost
```

You should get a bunch of HTML as a response, indicating that the container is indeed up and running, serving the hello world page.

In the upper left corner, you can see a clock that tells you how much time is left in the current session. Using the playground is free, but it is limited to 4 hours max per session. At any time, you can start a new session. Please also notice the **+ADD NEW INSTANCE** link on the left side. With it, you can create a new container host where you can play with containers. You can create up to four hosts.

4. Click on the **+ADD NEW INSTANCE** link to create a new container host. After a short time, a terminal and some additional information are displayed on the right side of the page as shown:



Figure 2.17: Running instance on Play with Docker

5. Inside the terminal, execute the hello world command that we already ran locally on our machine before:

```
docker run -d -p 80:80 docker/getting-started
```

6. Use `curl` to verify that the container is running:

```
curl localhost
```

You should get a bunch of HTML as a response, indicating that the container is indeed up and running, serving the hello world page.

7. Experiment with other Docker commands you have used before, such as
    - Pulling a container image from the Docker hub:
      `docker image pull alpine:latest`
    - Listing all container images available in the local cache:
      `docker image ls`
    - Listing all running containers
      `docker container ls`
    - Retrieving the version information for the client and server used:
      `docker version`
8. When done experimenting with **Play with Docker**, just close your browser window.

Play with Kubernetes

For all those people who want to go a step further than just playing or experimenting with a single container and who want to experiment with Kubernetes, the very same two Docker captains have created another playground called Play with Kubernetes. Here is the link: **https://labs.play-with-k8s.com**, following is the reference figure:



Figure 2.18: Play with Kubernetes landing page

7. Experiment with other Docker commands you have used before, such as
    - Pulling a container image from the Docker hub:
      `docker image pull alpine:latest`
    - Listing all container images available in the local cache:
      `docker image ls`
    - Listing all running containers
      `docker container ls`
    - Retrieving the version information for the client and server used:
      `docker version`
8. When done experimenting with **Play with Docker**, just close your browser window.

Play with Kubernetes

For all those people who want to go a step further than just playing or experimenting with a single container and who want to experiment with Kubernetes, the very same two Docker captains have created another playground called Play with Kubernetes. Here is the link: **https://labs.play-with-k8s.com**, following is the reference figure:

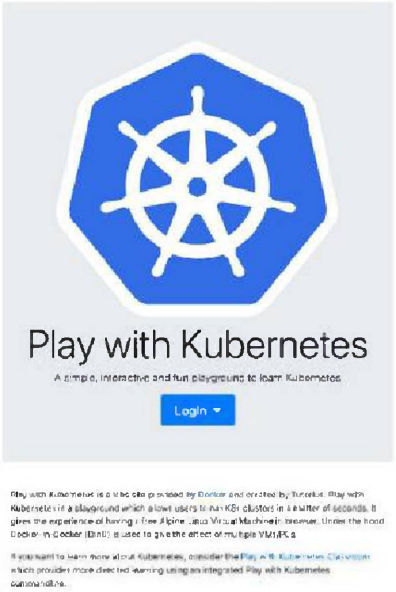


Figure 2.18: Play with Kubernetes landing page

Once again, you have to login. You can either use your Docker or your GitHub account. When you have successfully logged in, click the start button to setup a Kubernetes environment for you. It may take a moment or two; thus, please be patient.

Note: If you have difficulties to access this site, please make sure you turn off popup blockers for this site!

The environment you will see looks pretty much identical to the Docker playground presented in the previous section. At this point, we will not dive into this playground any further as we first need to learn what Kubernetes is and how we can use it. We will learn this in chapter 10, Orchestrating containers. Once you have a basic understanding of how to use Kubernetes and want to experiment with it without having to install anything as a prerequisite on your computer, you may come back to this playground.

Conclusion

In this chapter, we have prepared our working environment by selecting, installing, and configuring a small number of essential tools. We have selected Docker Desktop to run Docker containers locally and Visual Studio Code to author and edit scripts, Dockerfiles, and whole software projects. Finally, we have selected Windows Terminal on Windows 10 or 11 and iTerm 3 on MacOS as our terminal, where we execute container-related commands and scripts. For Windows 10 or 11 users, we have also shown how to enable and configure WSL 2, which provides a native Linux environment running inside Windows.

Last but not least, we gave a quick tour through two amazing online sites where one can explore Docker containers and Kubernetes without having to prepare or install anything.

In the upcoming chapter, we will finally start to get our hands dirty and begin to play with containers. We will start with the most basic example—the hello world of Docker containers—and move on from there.

Questions

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

1. What are package managers used for, and why are developers using them?
2. Name a few reasons why Docker Desktop is the most popular tool for engineers working with Docker containers.
3. Name a few reasons why you need a code editor, such as Visual Studio Code. Why not just use a normal text editor?

Once again, you have to login. You can either use your Docker or your GitHub account. When you have successfully logged in, click the start button to setup a Kubernetes environment for you. It may take a moment or two; thus, please be patient.

Note: If you have difficulties to access this site, please make sure you turn off popup blockers for this site!

The environment you will see looks pretty much identical to the Docker playground presented in the previous section. At this point, we will not dive into this playground any further as we first need to learn what Kubernetes is and how we can use it. We will learn this in chapter 10, Orchestrating containers. Once you have a basic understanding of how to use Kubernetes and want to experiment with it without having to install anything as a prerequisite on your computer, you may come back to this playground.

Conclusion

In this chapter, we have prepared our working environment by selecting, installing, and configuring a small number of essential tools. We have selected Docker Desktop to run Docker containers locally and Visual Studio Code to author and edit scripts, Dockerfiles, and whole software projects. Finally, we have selected Windows Terminal on Windows 10 or 11 and iTerm 3 on MacOS as our terminal, where we execute container-related commands and scripts. For Windows 10 or 11 users, we have also shown how to enable and configure WSL 2, which provides a native Linux environment running inside Windows.

Last but not least, we gave a quick tour through two amazing online sites where one can explore Docker containers and Kubernetes without having to prepare or install anything.

In the upcoming chapter, we will finally start to get our hands dirty and begin to play with containers. We will start with the most basic example—the hello world of Docker containers—and move on from there.

Questions

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

1. What are package managers used for, and why are developers using them?
2. Name a few reasons why Docker Desktop is the most popular tool for engineers working with Docker containers.
3. Name a few reasons why you need a code editor, such as Visual Studio Code. Why not just use a normal text editor?

4. Name a few scenarios where you would use the Play with Docker site. What are two or three of the main reasons to use it?

Answers

1. Package managers such as Chocolatey on Windows and Homebrew on MacOS are used to download applications, packages, and libraries from well-defined repositories and configure and install them on a computer or server. Developers prefer to use package managers over traditional installers since, with their aid, one can automate the whole process where traditional installers often need manual intervention.
2. Docker Desktop has been so popular since it.
    a. Is developed by Docker, the company which invented Docker containers
    b. Provides all the necessary tools to download, run and inspect Docker containers
    c. One can show container logs and execute a shell inside any running container
    d. It integrates well with Docker Hub
    e. It provides support for a single-node Kubernetes or Docker Swarm cluster
3. A normal text editor does not support color coding and IntelliSense. A simple text editor also does not support the definition of tasks such as compiling and debugging. Normal text editors usually are not as extensible as code editors such as VS Code.
4. Some reasons to use Play with Docker are as follows:
    a. You are on a computer without any tools installed and want to quickly demo or experiment with some Docker container features
    b. You want to experiment with a multi-node Docker Swarm cluster
    c. You want to deepen your knowledge about Docker and thus decide to execute some of the workshops provided at **https://training.play-with-docker.com**.

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 start-up that develops a highly acclaimed new mobile app for iOS and Android. During the interview, you are confronted with the following questions:

4. Name a few scenarios where you would use the Play with Docker site. What are two or three of the main reasons to use it?

Answers

1. Package managers such as Chocolatey on Windows and Homebrew on MacOS are used to download applications, packages, and libraries from well-defined repositories and configure and install them on a computer or server. Developers prefer to use package managers over traditional installers since, with their aid, one can automate the whole process where traditional installers often need manual intervention.
2. Docker Desktop has been so popular since it.
    a. Is developed by Docker, the company which invented Docker containers
    b. Provides all the necessary tools to download, run and inspect Docker containers
    c. One can show container logs and execute a shell inside any running container
    d. It integrates well with Docker Hub
    e. It provides support for a single-node Kubernetes or Docker Swarm cluster
3. A normal text editor does not support color coding and IntelliSense. A simple text editor also does not support the definition of tasks such as compiling and debugging. Normal text editors usually are not as extensible as code editors such as VS Code.
4. Some reasons to use Play with Docker are as follows:
    a. You are on a computer without any tools installed and want to quickly demo or experiment with some Docker container features
    b. You want to experiment with a multi-node Docker Swarm cluster
    c. You want to deepen your knowledge about Docker and thus decide to execute some of the workshops provided at **https://training.play-with-docker.com**.

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 start-up that develops a highly acclaimed new mobile app for iOS and Android. During the interview, you are confronted with the following questions:

1. We are running all our backend code in Docker containers. Which tools would you recommend our engineers to use to facilitate their work with containers? And why do you suggest these tools?
2. Our engineers use VS Code. Are you familiar with this tool? If yes, what do you like most about this code editor; how does it improve your workflow as a developer—specifically in regards to working with containers?
3. We are deploying all our backend code to Kubernetes when running in production. Our engineers use Docker Desktop to test everything locally on their computers, including running a container on Kubernetes. What do you have to do to enable this scenario in Docker Desktop?
4. You are starting your new job and have just installed all the necessary tools on your brand-new MacBook Pro. When trying to run our containerized software for the first time locally on your laptop, the containers will not start, and you get some error messages that Docker ran out of space. What are you doing to fix the problem?

Possible answers to job interview questions

1. I would strongly recommend to use Docker Desktop to work locally with Docker containers. It is the most popular tool for developers and DevOps engineers. It makes common tasks with containers easy and even makes it possible to run containers in a single-node Docker Swarm or a single-node Kubernetes cluster. When working with apps consisting of multiple containers, I highly recommend the use of Docker-compose, which was designed by Docker exactly for this purpose. Furthermore, we need a powerful yet lightweight code editor that ideally runs on all platforms, Windows 10 or 11, MacOS, and Linux. Visual Studio Code is such a tool, and it helps that it is the favorite code editor by a large margin, according to StackOverflow.
2. VS Code is a lightweight yet powerful code editor that gains most of its flexibility and power through extensions. A vast ecosystem of such extensions exists. Among them are many extensions that facilitate the work with Docker containers. VS Code also has an integrated terminal, which allows a developer to only have to use one single tool most of the time. One other huge advantage of VS Code is also that it runs on any of the important platforms, Windows 10 or 11, MacOS, and Linux. This is important as most developers work on either a Mac or a Windows machine, yet containers run on Linux.

1. We are running all our backend code in Docker containers. Which tools would you recommend our engineers to use to facilitate their work with containers? And why do you suggest these tools?
2. Our engineers use VS Code. Are you familiar with this tool? If yes, what do you like most about this code editor; how does it improve your workflow as a developer—specifically in regards to working with containers?
3. We are deploying all our backend code to Kubernetes when running in production. Our engineers use Docker Desktop to test everything locally on their computers, including running a container on Kubernetes. What do you have to do to enable this scenario in Docker Desktop?
4. You are starting your new job and have just installed all the necessary tools on your brand-new MacBook Pro. When trying to run our containerized software for the first time locally on your laptop, the containers will not start, and you get some error messages that Docker ran out of space. What are you doing to fix the problem?

Possible answers to job interview questions

1. I would strongly recommend to use Docker Desktop to work locally with Docker containers. It is the most popular tool for developers and DevOps engineers. It makes common tasks with containers easy and even makes it possible to run containers in a single-node Docker Swarm or a single-node Kubernetes cluster. When working with apps consisting of multiple containers, I highly recommend the use of Docker-compose, which was designed by Docker exactly for this purpose. Furthermore, we need a powerful yet lightweight code editor that ideally runs on all platforms, Windows 10 or 11, MacOS, and Linux. Visual Studio Code is such a tool, and it helps that it is the favorite code editor by a large margin, according to StackOverflow.
2. VS Code is a lightweight yet powerful code editor that gains most of its flexibility and power through extensions. A vast ecosystem of such extensions exists. Among them are many extensions that facilitate the work with Docker containers. VS Code also has an integrated terminal, which allows a developer to only have to use one single tool most of the time. One other huge advantage of VS Code is also that it runs on any of the important platforms, Windows 10 or 11, MacOS, and Linux. This is important as most developers work on either a Mac or a Windows machine, yet containers run on Linux.

3. Docker Desktop supports Kubernetes out of the box. It can be configured to run a single-node Kubernetes cluster. Developers can, thus, e.g., test deployments to Kubernetes on their local machine. `kubectl`, the main command line tool when working with Kubernetes, is automatically installed when one installs Docker desktop.

4. I will first make sure that Docker Desktop runs. Then, I will start the Dashboard via the context menu of Docker. On the Dashboard, I click the settings button, and under Preferences—Resources, I adjust the settings for CPU and Memory. Depending on the computer used and the requirements of the application, I will bump up the default values of 4 CPUs and 2 GB RAM. As an example, I would set the memory to 8 GB on a 16 GB computer.

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



3. Docker Desktop supports Kubernetes out of the box. It can be configured to run a single-node Kubernetes cluster. Developers can, thus, e.g., test deployments to Kubernetes on their local machine. `kubectl`, the main command line tool when working with Kubernetes, is automatically installed when one installs Docker desktop.

4. I will first make sure that Docker Desktop runs. Then, I will start the Dashboard via the context menu of Docker. On the Dashboard, I click the settings button, and under Preferences—Resources, I adjust the settings for CPU and Memory. Depending on the computer used and the requirements of the application, I will bump up the default values of 4 CPUs and 2 GB RAM. As an example, I would set the memory to 8 GB on a 16 GB computer.

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 3

Getting Familiar with Containers

Introduction

Now that we have learned about what containers are and why they are so useful to us and the IT sector in general and have setup our working environment, we are ready to start exploring containers and container images.

In this chapter, you will seriously get your hands dirty by learning how to work with containers. You will get to know how to start a container from an existing image and then how to stop and remove that container from the system. You will also learn how to list all containers currently running on your system or lingering in a stopped state in memory. You will further learn how to list all processes running inside a container, how to retrieve the log entries generated by applications running inside the container, and how to run additional processes in an already running container. As you can see, there is quite some work ahead of us. Let us dig in, shall we?

While you are working through this chapter, I highly recommend that you have Docker Desktop running and your terminal started. To achieve an optimal learning effect, try to reproduce each command or technique that you will be presented on your own system.

Chapter 3

Getting Familiar with Containers

Introduction

Now that we have learned about what containers are and why they are so useful to us and the IT sector in general and have setup our working environment, we are ready to start exploring containers and container images.

In this chapter, you will seriously get your hands dirty by learning how to work with containers. You will get to know how to start a container from an existing image and then how to stop and remove that container from the system. You will also learn how to list all containers currently running on your system or lingering in a stopped state in memory. You will further learn how to list all processes running inside a container, how to retrieve the log entries generated by applications running inside the container, and how to run additional processes in an already running container. As you can see, there is quite some work ahead of us. Let us dig in, shall we?

While you are working through this chapter, I highly recommend that you have Docker Desktop running and your terminal started. To achieve an optimal learning effect, try to reproduce each command or technique that you will be presented on your own system.

Structure

In this chapter, we will discuss the following topics:

- Downloading a first container image
- Running a container
- Listing container images and container instances
- Stopping and removing a container
- Listing processes running inside a container
- Retrieving the log of a container
- Running an additional process inside an already running container

We will start by downloading a first container image that we will also use to run containers later on.

Objectives

In this chapter, the reader will learn how to download a Docker image from Docker Hub. You will also learn how to run a container. Different types of containers will be used in this chapter to familiarize ourselves with the run command. Also, you will learn to run a container interactively or in the foreground process as well as in detach mode. The readers will get to know how to list all the Docker images that currently reside in your local image cache, and we will learn how to list containers that run on your system.

Downloading a first container image

To be able to run a container locally, its container image first has to be downloaded to the local image cache of your machine. Once the image is available in the local cache, Docker will use it whenever you want to run a container from this image. This has several advantages. First, starting a new container instance from a cached image is lightning fast, and second, not having to download the image from the internet each time you run a container saves a lot of bandwidth and potentially cost.

Images can be downloaded in different ways, either explicitly using the `Docker pull` command or implicitly using the `docker run` command. Let us start with the first one and discuss it in detail. Let us try to download a Docker image called **alpine**. This image contains the Alpin Linux distro and is one of the many curated standard images on Docker Hub.

Structure

In this chapter, we will discuss the following topics:

- Downloading a first container image
- Running a container
- Listing container images and container instances
- Stopping and removing a container
- Listing processes running inside a container
- Retrieving the log of a container
- Running an additional process inside an already running container

We will start by downloading a first container image that we will also use to run containers later on.

Objectives

In this chapter, the reader will learn how to download a Docker image from Docker Hub. You will also learn how to run a container. Different types of containers will be used in this chapter to familiarize ourselves with the run command. Also, you will learn to run a container interactively or in the foreground process as well as in detach mode. The readers will get to know how to list all the Docker images that currently reside in your local image cache, and we will learn how to list containers that run on your system.

Downloading a first container image

To be able to run a container locally, its container image first has to be downloaded to the local image cache of your machine. Once the image is available in the local cache, Docker will use it whenever you want to run a container from this image. This has several advantages. First, starting a new container instance from a cached image is lightning fast, and second, not having to download the image from the internet each time you run a container saves a lot of bandwidth and potentially cost.

Images can be downloaded in different ways, either explicitly using the `Docker pull` command or implicitly using the `docker run` command. Let us start with the first one and discuss it in detail. Let us try to download a Docker image called **alpine**. This image contains the Alpin Linux distro and is one of the many curated standard images on Docker Hub.

NOTE: Docker Hub is a registry for Docker images managed by Docker. Many images stored on Docker Hub are public and can therefore be downloaded by anybody without needing any credentials. Docker also offers the possibility to store private images that are only available to you if you have the necessary credentials. This is very useful if you want to store confidential images containing proprietary code or applications.

Now, the command to download the said image into your local image cache looks like this:

```
docker image pull alpine
```

This command has multiple parts, which have the following meaning:

- **`docker:`** It is the name of the **command line interface** (CLI) of Docker and is always used to execute any Docker (container) specific command. Thus, remember, any container command always starts with the keyword **`docker`**.
- **`image:`** It denotes the context of the command. In this case, the context is Docker images, meaning the command refers to an image.
- **`pull:`** is the actual command we want to execute. In fact, we want to download (or pull) an image from Docker Hub.
- **`alpine:`** It is the name of the image we want to pull from Docker Hub.

Try to execute the preceding command in your terminal. The first time you do this, you should see an output similar to the following figure:

```
gnsch@sirius ~
$ docker image pull alpine
Using default tag: latest
latest: Pulling from library/alpine
540db60ca938: Pull complete
Digest: sha256:69e70a79f2d41ab5d637de98c1e0b055206ba40a8145e7bddb55ccc04e13cf8f
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
```

Figure 3.1: Pulling the alpine image from Docker Hub

As you will see further down, this output is only looking like this the first time you are trying to download a certain image; that is when the image is not already in the image cache. Now, what do the individual parts of the output tell us?

The first line—**`Using default tag: latest`**—indicates that we are downloading the version of the image that has a tag called **`latest`**. This means that if we omit to specify a tag, Docker will automatically assume the latest. Thus, we could have also used the following command:

```
docker image pull alpine:latest
```

NOTE: Docker Hub is a registry for Docker images managed by Docker. Many images stored on Docker Hub are public and can therefore be downloaded by anybody without needing any credentials. Docker also offers the possibility to store private images that are only available to you if you have the necessary credentials. This is very useful if you want to store confidential images containing proprietary code or applications.

Now, the command to download the said image into your local image cache looks like this:

```
docker image pull alpine
```

This command has multiple parts, which have the following meaning:

- **`docker:`** It is the name of the **command line interface** (CLI) of Docker and is always used to execute any Docker (container) specific command. Thus, remember, any container command always starts with the keyword **`docker`**.
- **`image:`** It denotes the context of the command. In this case, the context is Docker images, meaning the command refers to an image.
- **`pull:`** is the actual command we want to execute. In fact, we want to download (or pull) an image from Docker Hub.
- **`alpine:`** It is the name of the image we want to pull from Docker Hub.

Try to execute the preceding command in your terminal. The first time you do this, you should see an output similar to the following figure:

```
gnsch@sirius ~
$ docker image pull alpine
Using default tag: latest
latest: Pulling from library/alpine
540db60ca938: Pull complete
Digest: sha256:69e70a79f2d41ab5d637de98c1e0b055206ba40a8145e7bddb55ccc04e13cf8f
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
```

Figure 3.1: Pulling the alpine image from Docker Hub

As you will see further down, this output is only looking like this the first time you are trying to download a certain image; that is when the image is not already in the image cache. Now, what do the individual parts of the output tell us?

The first line—**`Using default tag: latest`**—indicates that we are downloading the version of the image that has a tag called **`latest`**. This means that if we omit to specify a tag, Docker will automatically assume the latest. Thus, we could have also used the following command:

```
docker image pull alpine:latest
```

and the result would have been the same. We will talk about image tags in more detail shortly.

The next line—`latest: Pulling from library/alpine`—indicates that the image with version latest is pulled from the source called `library/alpine`. This is the short form of `docker.io/library/alpine:latest`, which you can see on the last line of the preceding output. We will talk about this in more detail a bit later in this chapter.

The line `540db60ca938: Pull complete` tells us that a layer with a hash code 540db60ca938 has been completely downloaded and stored in the local image cache. As we will see, an image consists of one to several layers, each with a unique hash code. In our case, the alpine image is very simple in comparison and consists of a single layer.

Next, the line `Digest: sha256:69e70a79f2d41a`... gives us the cryptographic code of the downloaded image that is used to verify that no tampering has happened with the original image and that we really got what we expected.

Finally, the line `Status: Downloaded newer image for alpine:latest` tells us that a newer version of the requested image has been downloaded from Docker Hub. In our case, there was no image in the local cache; thus, this makes sense.

Now, let us see if we repeat the preceding command again. This time the output should look like the one shown in *figure 3.2*, which is slightly different that the previous output. Can you notice the difference?

```
gnsch@sirius ~
$ docker image pull alpine
Using default tag: latest
latest: Pulling from library/alpine
Digest: sha256:69e70a79f2d41ab5d637de98c1e0b055206ba40a8145e7bddb55ccc04e13cf8f
Status: Image is up to date for alpine:latest
docker.io/library/alpine:latest
```

Figure 3.2: Pulling an image that already exists in the local image cache

Yes, you are right; the Docker CLI has discovered that the image is already present in the local image cache and thus did not download it again. This is indicated by the line `Status: Image is up to date for alpine:latest`. This saves us a lot of time and possibly money, specifically if the image is large.

Now that we have successfully downloaded the first Docker image let us try to use it.

and the result would have been the same. We will talk about image tags in more detail shortly.

The next line—`latest: Pulling from library/alpine`—indicates that the image with version latest is pulled from the source called `library/alpine`. This is the short form of `docker.io/library/alpine:latest`, which you can see on the last line of the preceding output. We will talk about this in more detail a bit later in this chapter.

The line `540db60ca938: Pull complete` tells us that a layer with a hash code 540db60ca938 has been completely downloaded and stored in the local image cache. As we will see, an image consists of one to several layers, each with a unique hash code. In our case, the alpine image is very simple in comparison and consists of a single layer.

Next, the line `Digest: sha256:69e70a79f2d41a`... gives us the cryptographic code of the downloaded image that is used to verify that no tampering has happened with the original image and that we really got what we expected.

Finally, the line `Status: Downloaded newer image for alpine:latest` tells us that a newer version of the requested image has been downloaded from Docker Hub. In our case, there was no image in the local cache; thus, this makes sense.

Now, let us see if we repeat the preceding command again. This time the output should look like the one shown in *figure 3.2*, which is slightly different that the previous output. Can you notice the difference?

```
gnsch@sirius ~
$ docker image pull alpine
Using default tag: latest
latest: Pulling from library/alpine
Digest: sha256:69e70a79f2d41ab5d637de98c1e0b055206ba40a8145e7bddb55ccc04e13cf8f
Status: Image is up to date for alpine:latest
docker.io/library/alpine:latest
```

Figure 3.2: Pulling an image that already exists in the local image cache

Yes, you are right; the Docker CLI has discovered that the image is already present in the local image cache and thus did not download it again. This is indicated by the line `Status: Image is up to date for alpine:latest`. This saves us a lot of time and possibly money, specifically if the image is large.

Now that we have successfully downloaded the first Docker image let us try to use it.

Running a container

In the previous section, we have downloaded a Docker image called alpine, which contains the Alpine Linux distro. We can now use this image and run a container from it. In your terminal, execute the following command:

```
docker container run alpine echo 'Hello World'
```

On your screen, you should see something similar, as shown in *figure 3.3*.



Figure 3.3: Running a first Docker container

Although the result does not look dramatic, there is a lot going on. Let us analyze the preceding command and explain the meaning of each individual part. We have the following:

- **`docker`**: This we already had when we pulled the image; it is the Docker CLI
- **`container`**: This is the context of the command; in this case, we are referring to containers
- **`run`**: This is the actual command that we want to execute—run (a container) in this case
- **`alpine`**: This is the name of the container image from which we want to run a container
- **`echo 'Hello World'`**: Finally, the remains of the preceding run command, we have a (Linux) command that will be executed inside the running container

If we want to run another command, say, a ping command inside a container, we can use the following command:

```
docker container run alpine ping 8.8.8.8 -c 5
```

Running a container

In the previous section, we have downloaded a Docker image called alpine, which contains the Alpine Linux distro. We can now use this image and run a container from it. In your terminal, execute the following command:

```
docker container run alpine echo 'Hello World'
```

On your screen, you should see something similar, as shown in *figure 3.3*.



Figure 3.3: Running a first Docker container

Although the result does not look dramatic, there is a lot going on. Let us analyze the preceding command and explain the meaning of each individual part. We have the following:

- **`docker`**: This we already had when we pulled the image; it is the Docker CLI
- **`container`**: This is the context of the command; in this case, we are referring to containers
- **`run`**: This is the actual command that we want to execute—run (a container) in this case
- **`alpine`**: This is the name of the container image from which we want to run a container
- **`echo 'Hello World'`**: Finally, the remains of the preceding run command, we have a (Linux) command that will be executed inside the running container

If we want to run another command, say, a ping command inside a container, we can use the following command:

```
docker container run alpine ping 8.8.8.8 -c 5
```

This again will start a container from the alpine image and execute the **ping 8.8.8.8 -c5** command inside this container. The result looks like the following shown here:

```
gnsch@sirius ~
$ docker container run alpine ping 8.8.8.8 -c 5
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=37 time=7.303 ms
64 bytes from 8.8.8.8: seq=1 ttl=37 time=9.152 ms
64 bytes from 8.8.8.8: seq=2 ttl=37 time=8.540 ms
64 bytes from 8.8.8.8: seq=3 ttl=37 time=9.064 ms
64 bytes from 8.8.8.8: seq=4 ttl=37 time=8.908 ms

--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 7.303/8.593/9.152 ms
```

Figure. 3.4: Running ping inside an alpine container

As you can see, the DNS server of Google (IP address 8.8.8.8) is pinged five times from within the running container.

You can run any complex command or script inside a container. We will be using this capability many times throughout this book, as you will see.

Now, you may ask yourself what happens with the container once the command, such as the echo command, is finished. This is indeed an important question, and I want to explain this now.

The **echo** or **ping** commands are (mini) applications. To run an application on Linux, a process is spawn. The application is run in the context of this process. Once the application finishes, the process is terminated. Now, when the so-called main process inside a container is terminated, then automatically, the container is also terminated. The main process is just the first process that is started inside a container. Thus, the answer to your question is that the container is terminated automatically when the command is finished.

Now, let us execute this command and see what is going to happen:

```
docker container run centos ping 8.8.8.8 -c5
```

This again will start a container from the alpine image and execute the **ping 8.8.8.8 -c5** command inside this container. The result looks like the following shown here:

```
gnsch@sirius ~
$ docker container run alpine ping 8.8.8.8 -c 5
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=37 time=7.303 ms
64 bytes from 8.8.8.8: seq=1 ttl=37 time=9.152 ms
64 bytes from 8.8.8.8: seq=2 ttl=37 time=8.540 ms
64 bytes from 8.8.8.8: seq=3 ttl=37 time=9.064 ms
64 bytes from 8.8.8.8: seq=4 ttl=37 time=8.908 ms

--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 7.303/8.593/9.152 ms
```

Figure. 3.4: Running ping inside an alpine container

As you can see, the DNS server of Google (IP address 8.8.8.8) is pinged five times from within the running container.

You can run any complex command or script inside a container. We will be using this capability many times throughout this book, as you will see.

Now, you may ask yourself what happens with the container once the command, such as the echo command, is finished. This is indeed an important question, and I want to explain this now.

The **echo** or **ping** commands are (mini) applications. To run an application on Linux, a process is spawn. The application is run in the context of this process. Once the application finishes, the process is terminated. Now, when the so-called main process inside a container is terminated, then automatically, the container is also terminated. The main process is just the first process that is started inside a container. Thus, the answer to your question is that the container is terminated automatically when the command is finished.

Now, let us execute this command and see what is going to happen:

```
docker container run centos ping 8.8.8.8 -c5
```

The result should look like the following shown here:

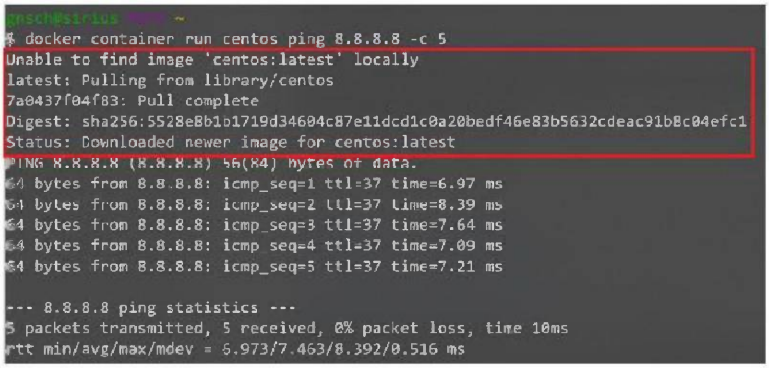


Figure 3.5: Implicitly pulling an image and then running a container from it

Please notice the part marked with the red rectangle. Evidently, the Docker CLI did not find an image called centos in the local image cache, and thus, it automatically downloaded it first and then ran a container from it. Thus, we realize that the Docker run command implicitly pulls an image if it is not already available locally. This comes in very handy in most cases.

Great, now let us run another container that we have prepared specifically for this book. It is the so-called trivia container, and what it does is periodically calling a public API on the Web to download a random trivia. The trivia is then output to STDOUT. The script running inside the container and producing these trivia pauses 5 seconds between individual trivia calls.

Let us run this container with the following command:

```
docker container run gnschenker/trivia:1.0
```

Let the container run for at least 30 seconds or so and observe what is happening. Upon the first run, you should see something similar to the following shown here:

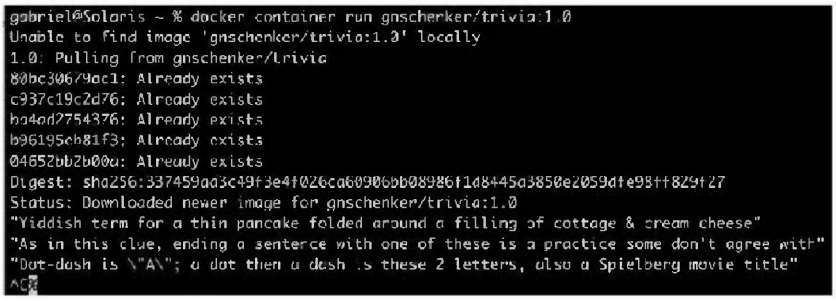


Figure 3.6: Running the Trivia container

The result should look like the following shown here:

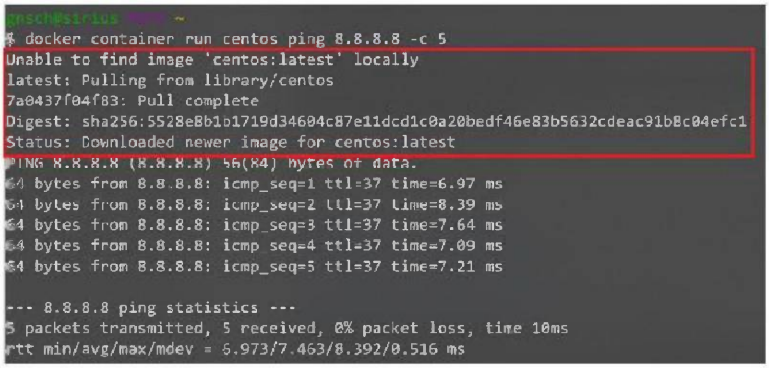


Figure 3.5: Implicitly pulling an image and then running a container from it

Please notice the part marked with the red rectangle. Evidently, the Docker CLI did not find an image called centos in the local image cache, and thus, it automatically downloaded it first and then ran a container from it. Thus, we realize that the Docker run command implicitly pulls an image if it is not already available locally. This comes in very handy in most cases.

Great, now let us run another container that we have prepared specifically for this book. It is the so-called trivia container, and what it does is periodically calling a public API on the Web to download a random trivia. The trivia is then output to STDOUT. The script running inside the container and producing these trivia pauses 5 seconds between individual trivia calls.

Let us run this container with the following command:

```
docker container run gnschenker/trivia:1.0
```

Let the container run for at least 30 seconds or so and observe what is happening. Upon the first run, you should see something similar to the following shown here:

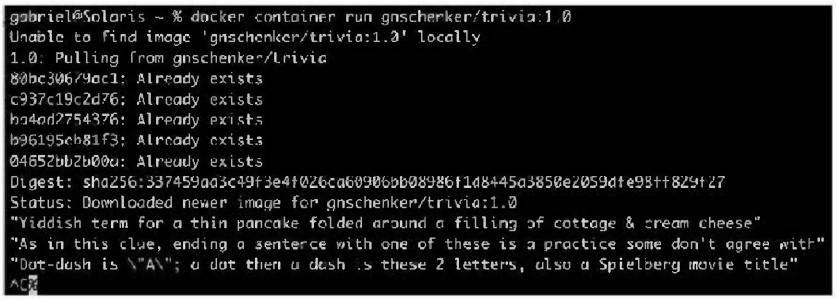


Figure 3.6: Running the Trivia container

Since we did not previously download the image `gnschenker/trivia:1.0` the Docker CLI does this automatically for us, as we can see in the preceding output. The first few lines are logging this. Then, the last three lines of the preceding output show three trivia that my container produced.

You may wonder what is the meaning of the **gnschenker** and the **1.0** parts in the container image name `gnschenker/trivia:1.0`. This is the fully qualified name of a Docker image. And we will explain all the individual parts of this name in the upcoming chapter. For the moment, let me just say this, the first part (`gnschenker`) is the name of my account on Docker Hub, and the second one (**1.0**) is the tag (often also referred to as the version) of the image.

You can terminate the container at any time by pressing *CTRL-c*. You may have to wait a few seconds until the container quits (this is due to the fact that there is a `sleep 5` command executed after every call to the trivia API, and this sleep command cannot be interrupted by the *CTRL-c*).

If you run the trivia container again, then this time it need not be downloaded, and the container starts outputting trivia immediately.

Running a container in detach mode

So far, we ran all our containers interactively or in the foreground, that is, in a blocking way. Most often, we may want to run a container in the background, also called in detach mode. When we do that, Linux spawns a new process and runs the container there, and our foreground process is not blocked. The `docker run` command uses the parameter `--detach` (or short `-d`) for this. Here is how we can run our trivia container in detach mode:

```
docker container run --detach gnschenker/trivia:1.0
```

Since the container runs in a background process, the trivia generated by it will not be displayed in our terminal, as the following figure shows:

```
gabriel@Solaris ~ % docker container run --detach gnschenker/trivia:1.0
48a9f3516506a8ced10bce8094f8a105032cedfda242afa57cf5326644068b3d
gabriel@Solaris ~ %
```

Figure 3.7: *Trivia container started in detach mode*

As you can see, the Docker CLI only outputs a long hash code—corresponding to the ID of the container that is now running in the background—and immediately returns. We are now free to continue working in our terminal window.

This container will now continue to run happily in the background until we either stop and remove it or until we shut down Docker. Let us kill this container, for now, using the preceding ID (`48a9f...`):

Since we did not previously download the image `gnschenker/trivia:1.0` the Docker CLI does this automatically for us, as we can see in the preceding output. The first few lines are logging this. Then, the last three lines of the preceding output show three trivia that my container produced.

You may wonder what is the meaning of the **gnschenker** and the **1.0** parts in the container image name `gnschenker/trivia:1.0`. This is the fully qualified name of a Docker image. And we will explain all the individual parts of this name in the upcoming chapter. For the moment, let me just say this, the first part (`gnschenker`) is the name of my account on Docker Hub, and the second one (**1.0**) is the tag (often also referred to as the version) of the image.

You can terminate the container at any time by pressing *CTRL-c*. You may have to wait a few seconds until the container quits (this is due to the fact that there is a `sleep 5` command executed after every call to the trivia API, and this sleep command cannot be interrupted by the *CTRL-c*).

If you run the trivia container again, then this time it need not be downloaded, and the container starts outputting trivia immediately.

Running a container in detach mode

So far, we ran all our containers interactively or in the foreground, that is, in a blocking way. Most often, we may want to run a container in the background, also called in detach mode. When we do that, Linux spawns a new process and runs the container there, and our foreground process is not blocked. The `docker run` command uses the parameter `--detach` (or short `-d`) for this. Here is how we can run our trivia container in detach mode:

```
docker container run --detach gnschenker/trivia:1.0
```

Since the container runs in a background process, the trivia generated by it will not be displayed in our terminal, as the following figure shows:

```
gabriel@Solaris ~ % docker container run --detach gnschenker/trivia:1.0
48a9f3516506a8ced10bce8094f8a105032cedfda242afa57cf5326644068b3d
gabriel@Solaris ~ %
```

Figure 3.7: *Trivia container started in detach mode*

As you can see, the Docker CLI only outputs a long hash code—corresponding to the ID of the container that is now running in the background—and immediately returns. We are now free to continue working in our terminal window.

This container will now continue to run happily in the background until we either stop and remove it or until we shut down Docker. Let us kill this container, for now, using the preceding ID (`48a9f...`):

```
docker container rm -f 48a9f
```

Do not worry about the details. It will become clear as we discuss the remove command in full detail later in this chapter.

Running multiple instances of a container

In *Chapter 1, Explaining Containers and their Benefits*, we have learned that a container is a self-contained system. It not only contains all the elements of the application running inside the container but also all necessary dependencies of that application, such as libraries and frameworks.

In this regard, the container is a world on its own, and this makes it possible for us to run multiple instances of a Docker image on the same machine without them interfering with each other. Let us try that with the trivia container as follows:

- Run a first trivia container in detach mode and call it trivia-1:

  ```
  docker container run -d --name trivia-1 gnschenker/trivia:1.0
  ```

- Run a second trivia container, also in detach mode and call it trivia-2:

  ```
  docker container run -d --name trivia-2 gnschenker/trivia:1.0
  ```

- Run a third trivia container called trivia-3:

  ```
  docker container run -d --name trivia-3 gnschenker/trivia:1.0
  ```

- List all containers running on your system with the following:

  ```
  docker container ls
  ```

  No worries, we will explain the preceding command in detail in the next section of this chapter.

- Observe an output similar to this one:

```
→  ~ docker container ls
CONTAINER ID   IMAGE                      COMMAND              CREATED          STATUS          PORTS     NAMES
5dd795955430   gnschenker/trivia:1.0      "./entrypoint.sh"    7 seconds ago    Up 6 seconds              trivia-3
39bf1cb3fcec   gnschenker/trivia:1.0      "./entrypoint.sh"    13 seconds ago   Up 12 seconds             trivia-2
8301dedc93d9   gnschenker/trivia:1.0      "./entrypoint.sh"    23 seconds ago   Up 23 seconds             trivia-1
```

Figure 3.8: List of all running containers

- Remove the three trivia containers with the following command:

  ```
  docker container rm -f trivia-1 trivia-2 trivia-3
  ```

  Once again, do not worry if this command is unknown to you. We will explain it in detail in one of the following sections of this chapter.

In the next section, we will learn how we can list the container images and container instances that are present in our system.

```
docker container rm -f 48a9f
```

Do not worry about the details. It will become clear as we discuss the remove command in full detail later in this chapter.

Running multiple instances of a container

In *Chapter 1, Explaining Containers and their Benefits*, we have learned that a container is a self-contained system. It not only contains all the elements of the application running inside the container but also all necessary dependencies of that application, such as libraries and frameworks.

In this regard, the container is a world on its own, and this makes it possible for us to run multiple instances of a Docker image on the same machine without them interfering with each other. Let us try that with the trivia container as follows:

- Run a first trivia container in detach mode and call it trivia-1:

  ```
  docker container run -d --name trivia-1 gnschenker/trivia:1.0
  ```

- Run a second trivia container, also in detach mode and call it trivia-2:

  ```
  docker container run -d --name trivia-2 gnschenker/trivia:1.0
  ```

- Run a third trivia container called trivia-3:

  ```
  docker container run -d --name trivia-3 gnschenker/trivia:1.0
  ```

- List all containers running on your system with the following:

  ```
  docker container ls
  ```

  No worries, we will explain the preceding command in detail in the next section of this chapter.

- Observe an output similar to this one:

```
→  ~ docker container ls
CONTAINER ID   IMAGE                     COMMAND             CREATED          STATUS          PORTS     NAMES
5dd795955430   gnschenker/trivia:1.0     "./entrypoint.sh"   7 seconds ago    Up 6 seconds              trivia-3
39bf1cb3fcec   gnschenker/trivia:1.0     "./entrypoint.sh"   13 seconds ago   Up 12 seconds             trivia-2
8301dedc93d9   gnschenker/trivia:1.0     "./entrypoint.sh"   23 seconds ago   Up 23 seconds             trivia-1
```

Figure 3.8: List of all running containers

- Remove the three trivia containers with the following command:

  ```
  docker container rm -f trivia-1 trivia-2 trivia-3
  ```

  Once again, do not worry if this command is unknown to you. We will explain it in detail in one of the following sections of this chapter.

In the next section, we will learn how we can list the container images and container instances that are present in our system.

Listing container images and container instances

As you start working with Docker, you will need to keep an overview of what Docker images you have currently lingering in your local image cache, and you also will want to know which containers are currently executing on your system or are just hanging there and consuming resources. Let us first talk about the images.

Listing Docker images

In the first section of this chapter, we have learned how to download a Docker image from Docker Hub. Over time we will probably download a plethora of different images. We want to investigate what images and what versions of the images are present; we can use the image list command. In your terminal, execute the following command:

```
docker image ls
```

The output on my machine looks like the figure shown here.

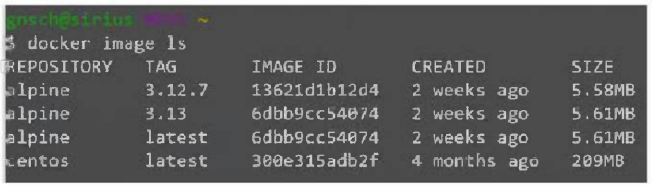


Figure 3.9: Listing all Docker images in the local cache

Please take a moment to analyze the output. There is a lot of important information packed into this screen. The output consists of the five columns Repository, Tag, Image ID, Created, and Size, as follows:

- The repository is the container for all images with the same name. In our sample, we have two repositories **alpine** and **centos**. A repository can contain multiple versions of an image with the same name.
- The versions are distinguished by the Tag. In our sample, we have a single version of the centos image and three versions of the alpine image with tags **latest**, **3.13**, and **3.12.7**.
- The column Created shows you when the owner of the image has created it. Evidently, the centos image is a bit older than the alpine image.

Listing container images and container instances

As you start working with Docker, you will need to keep an overview of what Docker images you have currently lingering in your local image cache, and you also will want to know which containers are currently executing on your system or are just hanging there and consuming resources. Let us first talk about the images.

Listing Docker images

In the first section of this chapter, we have learned how to download a Docker image from Docker Hub. Over time we will probably download a plethora of different images. We want to investigate what images and what versions of the images are present; we can use the image list command. In your terminal, execute the following command:

```
docker image ls
```

The output on my machine looks like the figure shown here.

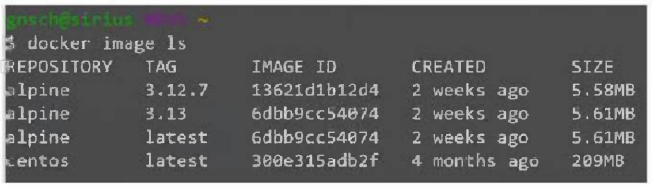


Figure 3.9: Listing all Docker images in the local cache

Please take a moment to analyze the output. There is a lot of important information packed into this screen. The output consists of the five columns Repository, Tag, Image ID, Created, and Size, as follows:

- The repository is the container for all images with the same name. In our sample, we have two repositories **alpine** and **centos**. A repository can contain multiple versions of an image with the same name.
- The versions are distinguished by the Tag. In our sample, we have a single version of the centos image and three versions of the alpine image with tags **latest**, **3.13**, and **3.12.7**.
- The column Created shows you when the owner of the image has created it. Evidently, the centos image is a bit older than the alpine image.

Note: on your machine you may find other images in the list. The preceding output is only a sample.

Finally, the column Size gives you the unpacked size of the corresponding image in the local image cache. It shows that the centos image contains a CentOS Linux distro that is much fatter than the Alpine Linux distro. No wonder the latter is often used when size matters.

Another way to list your images is to use the Dashboard of Docker Desktop. Click on the Docker whale icon in the system tray on Windows or on the right of the menu bar on your Mac to display the context menu, and then select Dashboard. On the left side, select **Images**, and you will be presented with a list similar to the figure shown here.

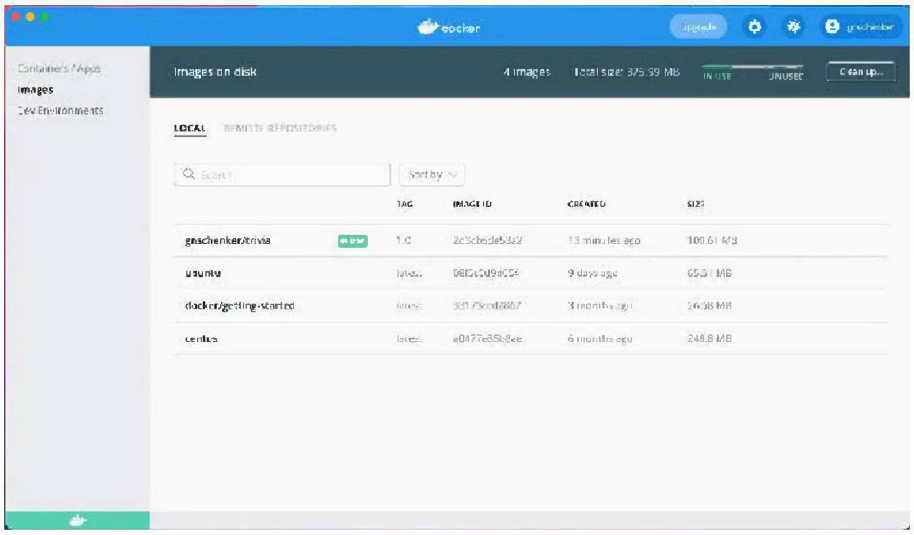


Figure 3.10: List of Docker images in Docker Desktop

In my case, you can see that I currently have four images in my local image cache. Among them is the trivia image we have used earlier in this chapter. Next to the name, we find the tag, image ID, creation date, and size of each image.

If you have many images in your cache, then you will be happy to know that there is a search function available to filter the list of images.

Note: on your machine you may find other images in the list. The preceding output is only a sample.

Finally, the column Size gives you the unpacked size of the corresponding image in the local image cache. It shows that the centos image contains a CentOS Linux distro that is much fatter than the Alpine Linux distro. No wonder the latter is often used when size matters.

Another way to list your images is to use the Dashboard of Docker Desktop. Click on the Docker whale icon in the system tray on Windows or on the right of the menu bar on your Mac to display the context menu, and then select Dashboard. On the left side, select **Images**, and you will be presented with a list similar to the figure shown here.

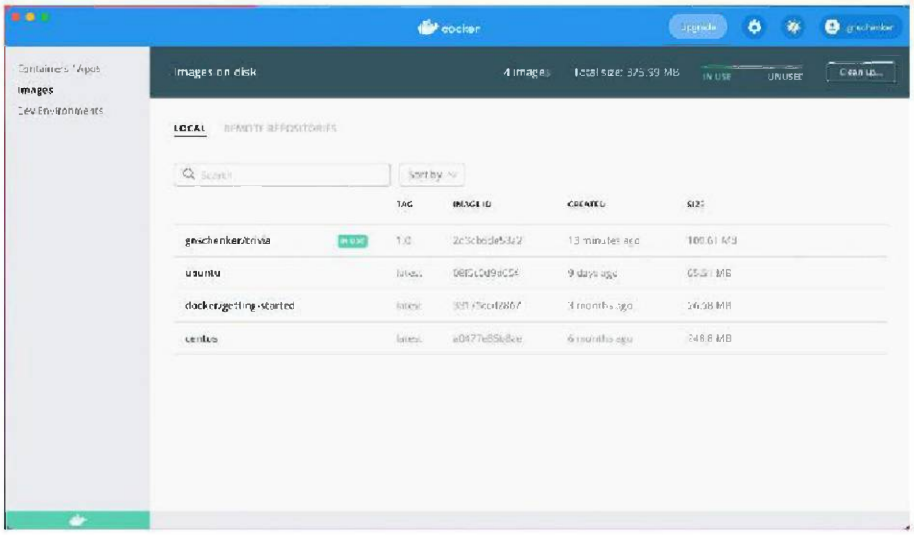


Figure 3.10: List of Docker images in Docker Desktop

In my case, you can see that I currently have four images in my local image cache. Among them is the trivia image we have used earlier in this chapter. Next to the name, we find the tag, image ID, creation date, and size of each image.

If you have many images in your cache, then you will be happy to know that there is a search function available to filter the list of images.

Next, we will use the Docker extension of Visual Studio Code to list images available on our system. Open VS Code and locate the Docker whale icon on the left sidebar and click it. You will see something like the following described in *figure 3.10*.

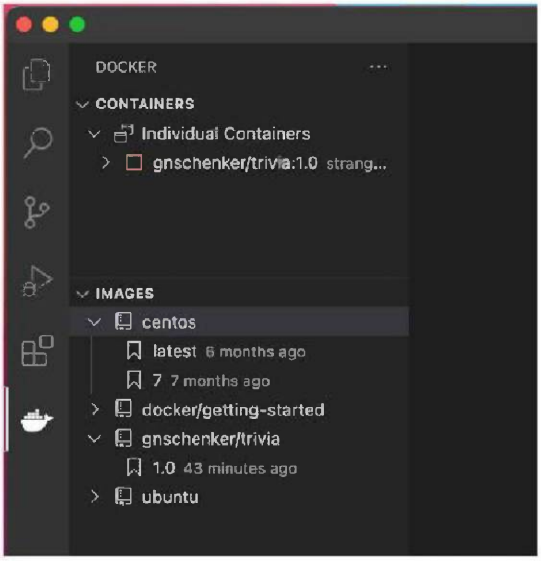


Figure 3.11: Display the list of images in the Docker extension of VS Code

As you can see in the preceding image, there are different sections in the window of the Docker extension. One of them is titled with **IMAGES**. There you see the list of all images that are currently available on your system. The images are grouped by image name. You can, for example, see that of the centos image; I have currently two versions on my system, `latest` and version **7**. Of the trivia container, I have a version with tag **1.0** available.

Listing containers

We have learned how to list images present in the local image cache. Now, we want to do the same for container instances. Execute the following command in your terminal:

```
docker container ls
```

where `ls` is a short form for "list". You should see the following described in *figure 3.11*.

Next, we will use the Docker extension of Visual Studio Code to list images available on our system. Open VS Code and locate the Docker whale icon on the left sidebar and click it. You will see something like the following described in *figure 3.10*.

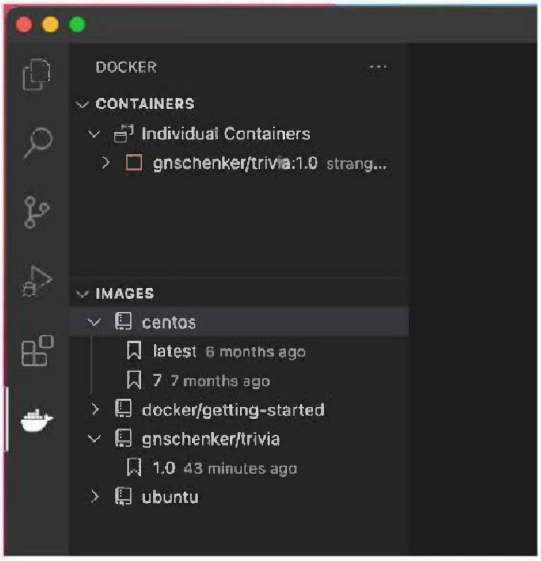


Figure 3.11: Display the list of images in the Docker extension of VS Code

As you can see in the preceding image, there are different sections in the window of the Docker extension. One of them is titled with **IMAGES**. There you see the list of all images that are currently available on your system. The images are grouped by image name. You can, for example, see that of the centos image; I have currently two versions on my system, `latest` and version **7**. Of the trivia container, I have a version with tag **1.0** available.

Listing containers

We have learned how to list images present in the local image cache. Now, we want to do the same for container instances. Execute the following command in your terminal:

```
docker container ls
```

where `ls` is a short form for "list". You should see the following described in *figure 3.11*.

```
gnsch@sirius ~
$ docker container ls
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

***Figure 3.12:** Listing all running containers*

Hmmm, strange, nothing to see here. Why is that so? Let us try a slightly modified version of the preceding command, including the `--all` (or short `-a`) parameter

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

and you should get the following shown here.

```
$ docker container ls --all
CONTAINER ID   IMAGE     COMMAND                  CREATED              STATUS                          PORTS     NAMES
f[illegible]   centos    "ping 8.8.8.8 -c 5"      52 minutes ago       Exited (0) 52 minutes ago                 pedantic_bassi
68752Gac3061   alpine    "ping 8.8.8.8 -c 5"      49 minutes ago       Exited (0) 49 minutes ago                 elated_lehmann
[illegible]    alpine    "echo 'Hello World'"     58 minutes ago       Exited (0) 58 minutes ago                 adoring_shtern
5c[illegible]  alpine    "echo 'Hello World'"     About an hour ago    Exited (0) About an hour ago              amazing_booth
```

***Figure 3.13:** Listing all containers, including the stopped ones*

Ok, this looks better. The difference is that this time when including the `--all` parameter in the command, all containers are listed, not only the currently running ones. Since we have been playing with containers before, we have a bunch of stopped containers.

In the preceding output, we once again have a lot of useful information presented to us. Let us describe the columns in detail as follows:

- **`CONTAINER ID`**: Each running container instance gets a unique ID assigned by Docker automatically. When referring to a specific container, it is often best to use this ID
- **`IMAGE`**: This denotes the image on which the container is based on
- **`COMMAND`**: This shows the first part of the command that was used to run the main process in the container. We can see the echo and ping commands we used in the previous section to demonstrate how to run a container
- **`STATUS`**: It indicates the current status of the container. In our case, it is Exited for all the containers listed. Other possible statuses are **`Created`** or **`Up`**, as shown here.

```
$ docker container ls --all
CONTAINER ID   IMAGE     COMMAND                  CREATED              STATUS                          PORTS     NAMES
[illegible]    alpine    "/bin/sh"                3 seconds ago        Created                                   hopeful_bhaskara
[illegible]    alpine    "ping 8.8.8.8"           14 seconds ago       Up 13 seconds                             funny_germain
[illegible]    centos    "ping 8.8.8.8 -c 5"      41 minutes ago       Exited (0) 41 minutes ago                 pedantic_bassi
[illegible]    alpine    "ping 8.8.8.8 -c 5"      58 minutes ago       Exited (0) 58 minutes ago                 elated_lehmann
[illegible]    alpine    "echo 'Hello World'"     About an hour ago    Exited (0) About an hour ago              adoring_shtern
[illegible]    alpine    "echo 'Hello World'"     About an hour ago    Exited (0) About an hour ago              amazing_booth
```

***Figure 3.14:** Showing containers with different status*

```
gnsch@sirius ~
$ docker container ls
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

***Figure 3.12:** Listing all running containers*

Hmmm, strange, nothing to see here. Why is that so? Let us try a slightly modified version of the preceding command, including the `--all` (or short `-a`) parameter

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

and you should get the following shown here.

```
$ docker container ls --all
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS                      PORTS     NAMES
f35b5f2dbad5   centos    "ping 8.8.8.8 -c 5"      52 minutes ago   Exited (0) 52 minutes ago             pedantic_bassi
68752Gac3661   alpine    "ping 8.8.8.8 -c 5"      49 minutes ago   Exited (0) 49 minutes ago             elated_lehmann
b3a387e2e489   alpine    "echo 'Hello World'"     58 minutes ago   Exited (0) 58 minutes ago             adoring_shtern
5c5f413a8d78   alpine    "echo 'Hello World'"     About an hour ago Exited (0) About an hour ago         amazing_booth
```

***Figure 3.13:** Listing all containers, including the stopped ones*

Ok, this looks better. The difference is that this time when including the `--all` parameter in the command, all containers are listed, not only the currently running ones. Since we have been playing with containers before, we have a bunch of stopped containers.

In the preceding output, we once again have a lot of useful information presented to us. Let us describe the columns in detail as follows:

- `CONTAINER ID`: Each running container instance gets a unique ID assigned by Docker automatically. When referring to a specific container, it is often best to use this ID
- `IMAGE`: This denotes the image on which the container is based on
- `COMMAND`: This shows the first part of the command that was used to run the main process in the container. We can see the echo and ping commands we used in the previous section to demonstrate how to run a container
- `STATUS`: It indicates the current status of the container. In our case, it is Exited for all the containers listed. Other possible statuses are `Created` or `Up`, as shown here.

```
$ docker container ls --all
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS                      PORTS     NAMES
2c62aa5a19cf   alpine    "/bin/sh"                3 seconds ago    Created                               hopeful_bhaskara
a7ad95317425   alpine    "ping 8.8.8.8"           14 seconds ago   Up 13 seconds                         funny_germain
f53b626bbdd3   centos    "ping 8.8.8.8 -c 5"      41 minutes ago   Exited (0) 41 minutes ago             pedantic_bassi
[illegible]    alpine    "ping 8.8.8.8 -c 5"      58 minutes ago   Exited (0) 58 minutes ago             elated_lehmann
b3a387e2e489   alpine    "echo 'Hello World'"     About an hour ago Exited (0) About an hour ago         adoring_shtern
5c5f413a8d78   alpine    "echo 'Hello World'"     About an hour ago Exited (0) About an hour ago         amazing_booth
```

***Figure 3.14:** Showing containers with different status*

- **PORTS**: Lists the ports that have been mapped from the container to the host. This will be explained in detail in *Chapter 6, Demystifying Container Networks*.
- **NAMES**: Lists the name (or names) that have been assigned to the container. If not specified explicitly in the run command, Docker assigns a random name composed from a list of adjectives and names of famous people.

Note you can also use a slightly different command to list your containers. It was the original command in the earlier days of Docker used to list containers:

```
docker ps --all
```

Although this one is shorter, it lacks the clarity of the previous version and is, thus, not recommended.

At times we might want to only list the ID of all containers. In this case, we can use the parameter `-q` to do so; thus:

```
docker container ls -q
```

This preceding command will list the IDs of all containers currently running on the system.

Stopping and removing a container

In the last section, we have noticed that we have a bunch of exited containers dangling in our system. Although these containers are not running anymore, they are still occupying precious resources on our computers. We can easily remove those containers. The command to remove a container looks like the following:

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

where `<container ID>` Is the ID of the container you want to remove.

Remember, you can get access to the ID (and other properties) of the container, for example, by listing the containers on your system with the following code:

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

On the authors' machine, for example, he can run this command to remove one of his existing containers, as shown here.

```
gnsch@sirius MINGW64 ~
$ docker container rm 5d5043a0d578
5d5043a0d578
```

Figure 3.15: Removing a dangling container from the system

- **PORTS**: Lists the ports that have been mapped from the container to the host. This will be explained in detail in *Chapter 6, Demystifying Container Networks*.
- **NAMES**: Lists the name (or names) that have been assigned to the container. If not specified explicitly in the run command, Docker assigns a random name composed from a list of adjectives and names of famous people.

Note you can also use a slightly different command to list your containers. It was the original command in the earlier days of Docker used to list containers:

```
docker ps --all
```

Although this one is shorter, it lacks the clarity of the previous version and is, thus, not recommended.

At times we might want to only list the ID of all containers. In this case, we can use the parameter `-q` to do so; thus:

```
docker container ls -q
```

This preceding command will list the IDs of all containers currently running on the system.

Stopping and removing a container

In the last section, we have noticed that we have a bunch of exited containers dangling in our system. Although these containers are not running anymore, they are still occupying precious resources on our computers. We can easily remove those containers. The command to remove a container looks like the following:

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

where `<container ID>` Is the ID of the container you want to remove.

Remember, you can get access to the ID (and other properties) of the container, for example, by listing the containers on your system with the following code:

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

On the authors' machine, for example, he can run this command to remove one of his existing containers, as shown here.

```
gnsch@sirius MINGW64 ~
$ docker container rm 5d5043a0d578
5d5043a0d578
```

Figure 3.15: Removing a dangling container from the system

Note: I do not need to use the full container ID in the remove command. I could also have just used part of it as in docker container rm 5d5. Docker will automatically remove the correct container as long as the fraction of the ID you are using is unique among all containers on your system.

The Docker CLI simply responds to my command by outputting the ID of the removed container.

It is also possible to remove a container that we identify by its name. The command to do so looks like the following:

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

Before we can remove a container from our system, we need to stop it (if it is not already stopped). The command to do this looks like the following:

```
docker container stop <container ID>
```

or

```
docker container stop <container name>
```

where `<container ID>` is the ID and `<container name>` is the name of the container we want to stop.

Let us try this again and use the trivia container as an example. Execute the following steps:

1. Run the trivia container in detached mode (`--detach` or `-d`):
   ```
   docker container run gnschenker/trivia:1.0 -d --name my-trivia
   ```
2. Stop the container by using the stop command and referencing the container by its name `my-trivia`:
   ```
   docker container stop my-trivia
   ```
3. The stop command may need a few seconds to finish. What Docker does is sending a **SIGINT** signal to the application running inside the container. If the application does not react on this signal, then Docker will forcefully stop the container after a predefined timeout time. By default, this time is 10 seconds.
4. Now, list all containers (including stopped ones) on your system and verify that our my-trivia container is indeed in STATUS `Exited`. Use this command to list the containers:
   ```
   docker container ls --all
   ```

Note: I do not need to use the full container ID in the remove command. I could also have just used part of it as in docker container rm 5d5. Docker will automatically remove the correct container as long as the fraction of the ID you are using is unique among all containers on your system.

The Docker CLI simply responds to my command by outputting the ID of the removed container.

It is also possible to remove a container that we identify by its name. The command to do so looks like the following:

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

Before we can remove a container from our system, we need to stop it (if it is not already stopped). The command to do this looks like the following:

```
docker container stop <container ID>
```

or

```
docker container stop <container name>
```

where `<container ID>` is the ID and `<container name>` is the name of the container we want to stop.

Let us try this again and use the trivia container as an example. Execute the following steps:

1. Run the trivia container in detached mode (`--detach` or `-d`):
   ```
   docker container run gnschenker/trivia:1.0 -d --name my-trivia
   ```
2. Stop the container by using the stop command and referencing the container by its name `my-trivia`:
   ```
   docker container stop my-trivia
   ```
3. The stop command may need a few seconds to finish. What Docker does is sending a **SIGINT** signal to the application running inside the container. If the application does not react on this signal, then Docker will forcefully stop the container after a predefined timeout time. By default, this time is 10 seconds.
4. Now, list all containers (including stopped ones) on your system and verify that our my-trivia container is indeed in STATUS `Exited`. Use this command to list the containers:
   ```
   docker container ls --all
   ```

5. As you can see, the container is still present in the system and, as such, occupying valuable resources. To free those resources, we have to remove the container. Do this with the following command:

   ```
   docker container rm my-trivia
   ```

4. Once again, list all containers and verify that the my-trivia container is not on the list anymore.

Often, we do want to stop and remove a container in one go. We can use the parameter `--force` (or short `-f`) with the remove command. This will forcefully remove a container, even if it is running. In this case, Docker will stop the container first. The command to forcefully remove, say, a container named `my-app` is as follows:

```
docker container rm --force my-app
```

By the way, if you know Unix or Linux well, then you know that one can combine multiple commands by using the result of one command as the input to the other command. This is also possible when using the Docker CLI. Try to find out what the following command does:

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

This is a very handy command, and I personally use it often. I have even defined it as an alias in my `.bash_alias` file:

```
alias drma='docker container rm -f $(docker container ls -aq)'
```

Remember that the Docker CLI allows us to combine parameters, thus:

```
docker container ls -aq
```

is equivalent to

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

Have you found out what the command does? Right, it removes all containers from the system. Here is why:

1. `$(docker container ls -aq)` outputs a list of IDs of all currently running or stopped containers.
2. `docker container rm -f <list of IDs>` forcefully removes these containers, where `<list of IDs>` is the list of IDs generated by the command in Step 1.

You may want to try this out on your system by running a few instances of the trivia container in detach mode and then execute the preceding command. Make sure that all containers have been removed thereafter.

5. As you can see, the container is still present in the system and, as such, occupying valuable resources. To free those resources, we have to remove the container. Do this with the following command:

   ```
   docker container rm my-trivia
   ```

4. Once again, list all containers and verify that the my-trivia container is not on the list anymore.

Often, we do want to stop and remove a container in one go. We can use the parameter `--force` (or short `-f`) with the remove command. This will forcefully remove a container, even if it is running. In this case, Docker will stop the container first. The command to forcefully remove, say, a container named `my-app` is as follows:

```
docker container rm --force my-app
```

By the way, if you know Unix or Linux well, then you know that one can combine multiple commands by using the result of one command as the input to the other command. This is also possible when using the Docker CLI. Try to find out what the following command does:

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

This is a very handy command, and I personally use it often. I have even defined it as an alias in my `.bash_alias` file:

```
alias drma='docker container rm -f $(docker container ls -aq)'
```

Remember that the Docker CLI allows us to combine parameters, thus:

```
docker container ls -aq
```

is equivalent to

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

Have you found out what the command does? Right, it removes all containers from the system. Here is why:

1. `$(docker container ls -aq)` outputs a list of IDs of all currently running or stopped containers.
2. `docker container rm -f <list of IDs>` forcefully removes these containers, where `<list of IDs>` is the list of IDs generated by the command in Step 1.

You may want to try this out on your system by running a few instances of the trivia container in detach mode and then execute the preceding command. Make sure that all containers have been removed thereafter.

Stopping and removing containers in Docker Desktop and VS Code

You probably already noticed it on your own when playing around with Docker Desktop or the Docker extension of VS Code that there is a way to stop and/or remove containers with the help of these tools.

Let us start with Docker Desktop, and from its dashboard run, stop, and remove a container. Proceed as follows:

1. Open the Docker Desktop dashboard
2. Navigate to the **IMAGES** tab and locate the trivia image as shown here:

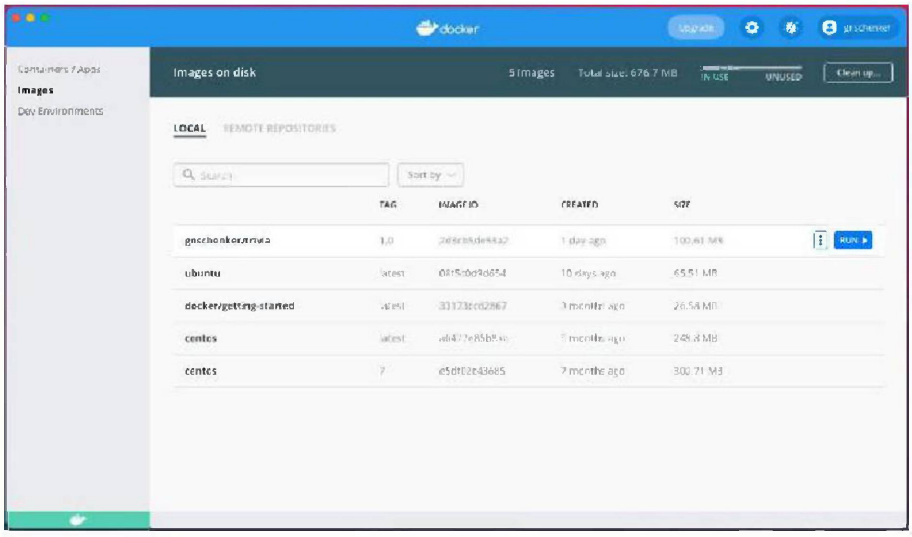


Figure 3.16: List of images in Docker Desktop

Note that a blue **Run** button is displayed on the right side if you hover over one of the images in the list.

Stopping and removing containers in Docker Desktop and VS Code

You probably already noticed it on your own when playing around with Docker Desktop or the Docker extension of VS Code that there is a way to stop and/or remove containers with the help of these tools.

Let us start with Docker Desktop, and from its dashboard run, stop, and remove a container. Proceed as follows:

1. Open the Docker Desktop dashboard
2. Navigate to the **IMAGES** tab and locate the trivia image as shown here:

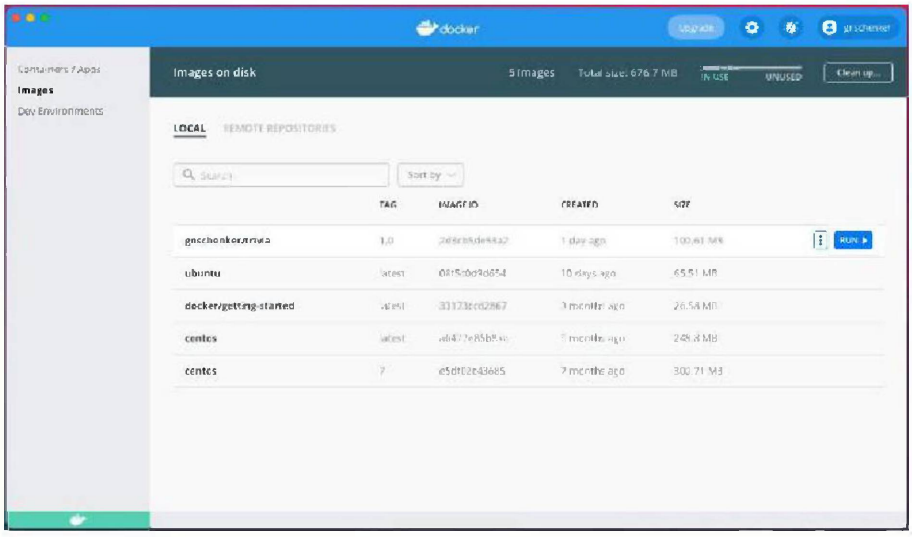


Figure 3.16: List of images in Docker Desktop

Note that a blue **Run** button is displayed on the right side if you hover over one of the images in the list.

3. Click the run button. On the New Container dialog box that will appear, just click **Run**, as shown here:



Figure 3.17: New container dialog in Docker Desktop

4. Notice that the Dashboard switches to the **Containers/Apps** tab and displays the newly started container, as shown here:



Figure 3.18: Docker Desktop displaying the trivia container just started

5. When you hover over the container in the preceding list, then four icons are displayed on the right side of the list. From right, these are DELETE, RESTART, and STOP. Click the **STOP** button and notice how after a short moment, the container icon left to the container name turns from green to grey, indicating that the container has been stopped. In addition, the exit code of the application is displayed—here 137, as shown here:



Figure 3.19: Displaying the container as stopped in Docker Desktop

3. Click the run button. On the New Container dialog box that will appear, just click **Run**, as shown here:



Figure 3.17: New container dialog in Docker Desktop

4. Notice that the Dashboard switches to the **Containers/Apps** tab and displays the newly started container, as shown here:



Figure 3.18: Docker Desktop displaying the trivia container just started

5. When you hover over the container in the preceding list, then four icons are displayed on the right side of the list. From right, these are DELETE, RESTART, and STOP. Click the **STOP** button and notice how after a short moment, the container icon left to the container name turns from green to grey, indicating that the container has been stopped. In addition, the exit code of the application is displayed—here 137, as shown here:



Figure 3.19: Displaying the container as stopped in Docker Desktop

6. Now hover again over the container entry in the list, and this time, click the **DELETE** button. Notice that the container will be removed from the system and will also disappear from this list.
7. **Optional**: Start yet another trivia container, and this time, find out what the RESTART button will do.
8. Use the dashboard of Docker Desktop to stop and remove all containers that are currently running on your system.

Now, we want to see how we can do the same using the Docker extension of VS Code. It is very straightforward. Just follow the following steps:

1. Open VS Code.
2. Click the Docker whale button in the left sidebar of VS Code.
3. Observe that in the window displayed, you have different areas, as shown in *figure 3.20*. The top two are **CONTAINERS** and **IMAGES**:

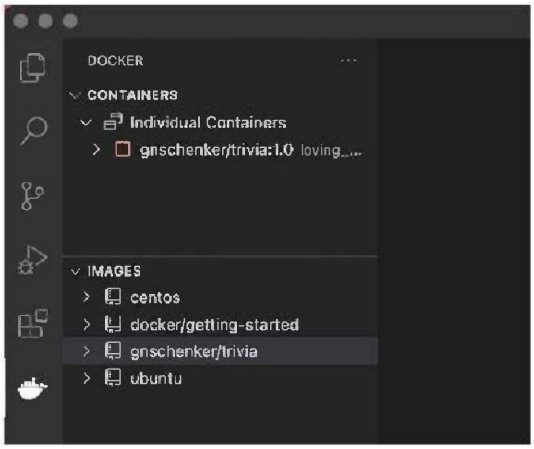


Figure 3.20. The Docker extension in VS Code

6. Now hover again over the container entry in the list, and this time, click the **DELETE** button. Notice that the container will be removed from the system and will also disappear from this list.
7. **Optional**: Start yet another trivia container, and this time, find out what the RESTART button will do.
8. Use the dashboard of Docker Desktop to stop and remove all containers that are currently running on your system.

Now, we want to see how we can do the same using the Docker extension of VS Code. It is very straightforward. Just follow the following steps:

1. Open VS Code.
2. Click the Docker whale button in the left sidebar of VS Code.
3. Observe that in the window displayed, you have different areas, as shown in *figure 3.20*. The top two are **CONTAINERS** and **IMAGES**:

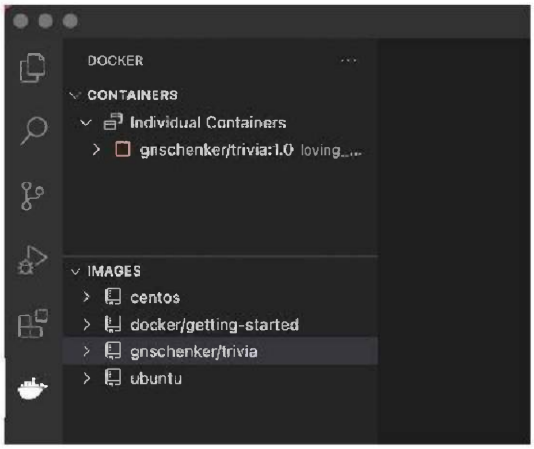


Figure 3.20. The Docker extension in VS Code

4. Locate the **gnschenker/trivia** container entry and open it to display the image with tag 1.0. Right-click on it to display the context menu as shown here:

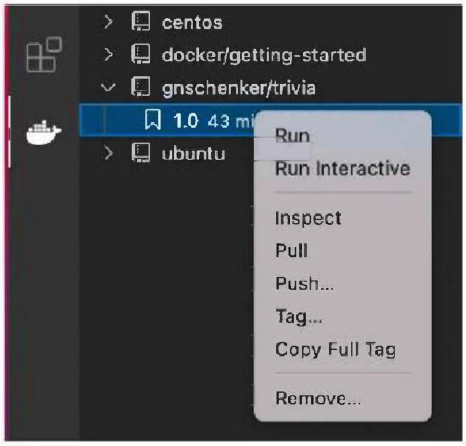


Figure 3.21: Running a container via the VS Code Docker extension

5. Click **Run** and observe the output in the **TERMINAL** window of VS Code as well as the **CONTAINERS** section in the Docker extension, where the new container will appear in the list of containers as shown here:

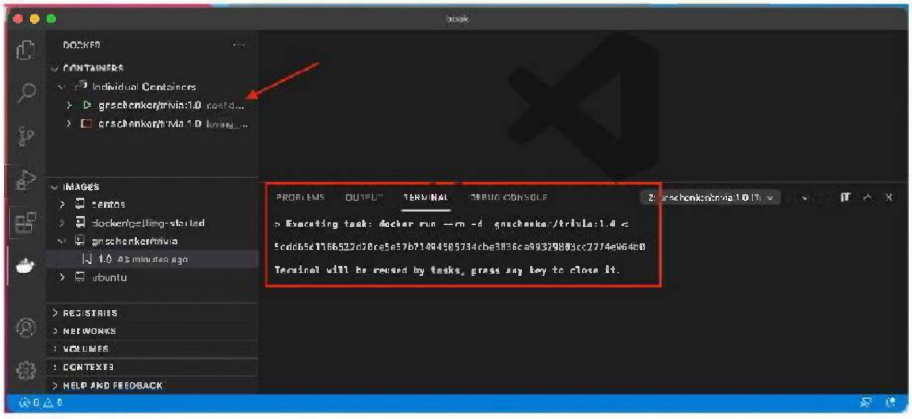


Figure 3.22: Newly started container appears in container list in VS Code

6. Now right-click on the new entry in the list of containers to display the context menu. You should see the familiar **Stop, Restart,** and **Remove** menu items as shown here:

4. Locate the **gnschenker/trivia** container entry and open it to display the image with tag 1.0. Right-click on it to display the context menu as shown here:

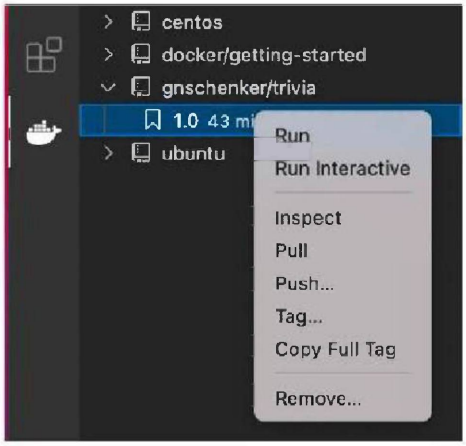


Figure 3.21: Running a container via the VS Code Docker extension

5. Click **Run** and observe the output in the **TERMINAL** window of VS Code as well as the **CONTAINERS** section in the Docker extension, where the new container will appear in the list of containers as shown here:

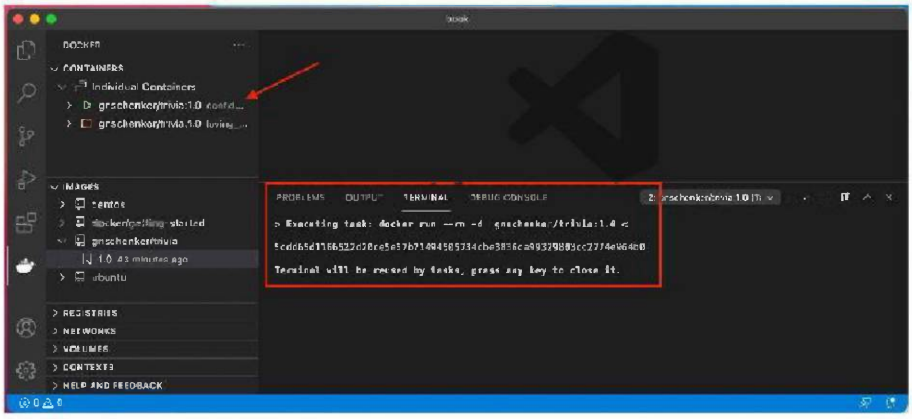


Figure 3.22: Newly started container appears in container list in VS Code

6. Now right-click on the new entry in the list of containers to display the context menu. You should see the familiar **Stop, Restart,** and **Remove** menu items as shown here:

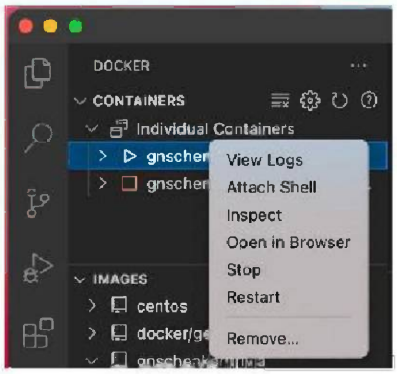


Figure 3.23: Container context menu in the Docker extension of VS Code

7. Select the **Stop** menu item first and observe what is happening.
8. Now, select the **Remove** menu item in the context menu and make sure that the container item disappears.

With this, we are done explaining how to list images or run, list, stop, and remove containers for the moment. But these commands will accompany you throughout the book; thus, make sure you really master them well before you proceed.

The next two sections are especially important when you need to debug your applications or services running in containers. One can never have enough information on what is going on when troubleshooting a system. Two helpful aspects are (a) what is running inside a container and (b) what kind of log output those processes are generating. Let us start with the former.

Listing processes running inside a container

For you, as a developer or for a sys admin, or a DevOps engineer, it can be useful to know what processes are running inside a given container.

1. Start a trivia container in detach mode and call it trivia-1:

   ```
   docker container run --detach --name trivia-1 gnschenker/
   trivia:1.0
   ```

2. Use the top command to display all running processes inside the preceding container

   ```
   docker container top trivia-1
   ```

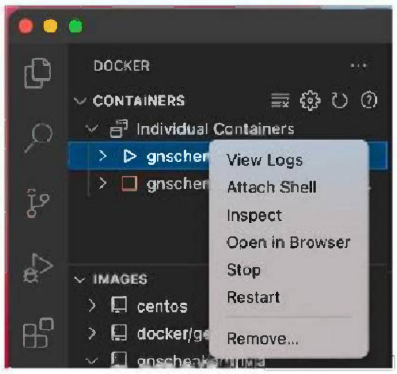


Figure 3.23: Container context menu in the Docker extension of VS Code

7. Select the **Stop** menu item first and observe what is happening.
8. Now, select the **Remove** menu item in the context menu and make sure that the container item disappears.

With this, we are done explaining how to list images or run, list, stop, and remove containers for the moment. But these commands will accompany you throughout the book; thus, make sure you really master them well before you proceed.

The next two sections are especially important when you need to debug your applications or services running in containers. One can never have enough information on what is going on when troubleshooting a system. Two helpful aspects are (a) what is running inside a container and (b) what kind of log output those processes are generating. Let us start with the former.

Listing processes running inside a container

For you, as a developer or for a sys admin, or a DevOps engineer, it can be useful to know what processes are running inside a given container.

1. Start a trivia container in detach mode and call it trivia-1:

   ```
   docker container run --detach --name trivia-1 gnschenker/
   trivia:1.0
   ```

2. Use the top command to display all running processes inside the preceding container

   ```
   docker container top trivia-1
   ```

you should see the following shown here:



Figure 3.24: Running processes inside the trivia container

In the output, we can see the list of processes, here 2 of them. Of each process, we can see information about who launched it (root), its process ID (PID), the uptime, and the command used to launch the process, among others.

3. Start another container, in this case, an Nginx Web server, and call it `web`:

```
docker container run --detach --name web nginx
```

you should see this:

```
gabriel@Solaris ~ % docker container run --detach --name web nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
fcad0c936ea5: Pull complete
286c9ad3753c: Pull complete
1dabc629d1a7: Pull complete
02ea76929254: Pull complete
faaaa8cdf04e: Pull complete
649f0b13b37c: Pull complete
Digest: sha256:6d75c99af15565a301e48297fa2d121e15d80ad526f8369c526324f0f7ccb750
Status: Downloaded newer image for nginx:latest
b212b67322f3fbaa58bcd70d1aad0158c30f72272980d1e2b09389737ce66488
```

Figure 3.25: Running an Nginx container

Since we did never run Nginx before, it is first downloaded and then run, as you can see in the preceding output.

4. Run the same top command to display its processes:

```
docker container top web
```

which will give us something similar to the following shown here:

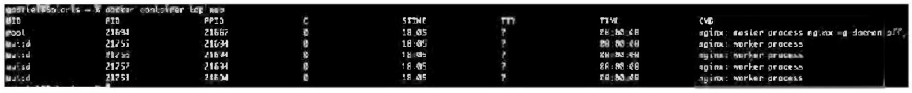


Figure 3.26: Running processes in Nginx

We can identify a master process, the actual Web server, and four worker processes running in this container.

Note: depending on the system you are working on, the number of worker processes may differ.

you should see the following shown here:



Figure 3.24: Running processes inside the trivia container

In the output, we can see the list of processes, here 2 of them. Of each process, we can see information about who launched it (root), its process ID (PID), the uptime, and the command used to launch the process, among others.

3. Start another container, in this case, an Nginx Web server, and call it `web`:

```
docker container run --detach --name web nginx
```

you should see this:

```
gabriel@Solaris ~ % docker container run --detach --name web nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
fcad0c936ea5: Pull complete
286c9ad3753c: Pull complete
1dabc629d1a7: Pull complete
02ea76929254: Pull complete
faaaa8cdf04e: Pull complete
649f0b13b37c: Pull complete
Digest: sha256:6d75c99af15565a301e48297fa2d121e15d80ad526f8369c526324f0f7ccb750
Status: Downloaded newer image for nginx:latest
b212b67322f3fbaa58bcd70d1aad0158c30f72272980d1e2b09389737ce66488
```

Figure 3.25: Running an Nginx container

Since we did never run Nginx before, it is first downloaded and then run, as you can see in the preceding output.

4. Run the same top command to display its processes:

```
docker container top web
```

which will give us something similar to the following shown here:

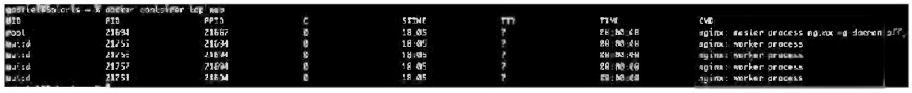


Figure 3.26: Running processes in Nginx

We can identify a master process, the actual Web server, and four worker processes running in this container.

Note: depending on the system you are working on, the number of worker processes may differ.

5. Use the following command to remove both containers from the system when done:

   ```
   docker container rm -f trivia-1 web
   ```

That is it for the processes part. Next, we will learn how to retrieve the logs from a container.

Retrieving the log of a container

Logs normally contain vital information about the runtime behavior of an application. Applications running inside a container are no exception to this. But how can we get access to this information? Well, as always, there exist several ways to do so. We will investigate three of them, first via the command line, second via Docker Desktop, and third via the Docker extension of VS Code. Let us start with the former.

We have used the trivia container quite a bit so far, and we know that it produces output. This output will be part of the container's log. Anything that goes into the Docker container log must be output to either STDOUT or STDERR. Docker will automatically pick those messages up and manage them. Docker will keep this information ready for the user of any other external system that wants to collect it. The log will be stored inside the container in temporary files managed by Docker. Since those temporary files containing the log reside inside the container, it is clear that the logs are lost once the container is removed from the system. You can stop a container, and the logs will still be there since a stopped container is kept in memory. But once the container is removed, the logs are gone.

Retrieving the log from the command line

Let us work with the trivia container. Proceed as follows:

1. Start a trivia container in detach mode and call it trivia-1:

   ```
   docker container run --detach --name trivia-1 gnschenker/
   trivia:1.0
   ```

2. Leave the container running for at least 30 seconds before you proceed.
3. To access the logs of the trivia-1 container, use the following command:

   ```
   docker container logs trivia-1
   ```

   and you should see an output similar to the following figure:

```
gabriel@Solaris ~ % docker container logs trivia-1
"A fitness saying goes, these 8 midriff muscles \"are made in the kitchen\""
"It's Chevy's extended range electric vehicle launched in 2010"
"Traditionally he is the last \"celebrity\" to appear in the parade"
"His biographers Tirechan & Muirchu credit him with converting the Irish to Christianity"
```

Figure 3.27: Retrieving the log of the trivia container

5. Use the following command to remove both containers from the system when done:

   ```
   docker container rm -f trivia-1 web
   ```

That is it for the processes part. Next, we will learn how to retrieve the logs from a container.

Retrieving the log of a container

Logs normally contain vital information about the runtime behavior of an application. Applications running inside a container are no exception to this. But how can we get access to this information? Well, as always, there exist several ways to do so. We will investigate three of them, first via the command line, second via Docker Desktop, and third via the Docker extension of VS Code. Let us start with the former.

We have used the trivia container quite a bit so far, and we know that it produces output. This output will be part of the container's log. Anything that goes into the Docker container log must be output to either STDOUT or STDERR. Docker will automatically pick those messages up and manage them. Docker will keep this information ready for the user of any other external system that wants to collect it. The log will be stored inside the container in temporary files managed by Docker. Since those temporary files containing the log reside inside the container, it is clear that the logs are lost once the container is removed from the system. You can stop a container, and the logs will still be there since a stopped container is kept in memory. But once the container is removed, the logs are gone.

Retrieving the log from the command line

Let us work with the trivia container. Proceed as follows:

1. Start a trivia container in detach mode and call it trivia-1:

   ```
   docker container run --detach --name trivia-1 gnschenker/
   trivia:1.0
   ```

2. Leave the container running for at least 30 seconds before you proceed.
3. To access the logs of the trivia-1 container, use the following command:

   ```
   docker container logs trivia-1
   ```

   and you should see an output similar to the following figure:

```
gabriel@Solaris ~ % docker container logs trivia-1
"A fitness saying goes, these 8 midriff muscles \"are made in the kitchen\""
"It's Chevy's extended range electric vehicle launched in 2010"
"Traditionally he is the last \"celebrity\" to appear in the parade"
"His biographers Tirechan & Muirchu credit him with converting the Irish to Christianity"
```

Figure 3.27: Retrieving the log of the trivia container

Note that the output represents the content of the log as it is at the moment when you run the preceding command. Docker will show the whole log from the beginning by default.

4. Now, if the container has been running for a while and we only want the last few lines of the log output, we can use the `--tail` (or short `-t`) parameter:

   ```
   docker container logs --tail 3 trivia-1
   ```

   This will output the last three lines of the log only.

5. If we want to follow the log, we can use the parameter -f to do so. Note that this will block the current terminal window:

   ```
   docker container logs --tail 10 -f trivia-1
   ```

   this will output the last 10 lines of the log collected so far and continue following the log.

   To stop following the log, just press *CTRL-c*.

6. Stop the trivia-1 container and make sure you can still access the log:

   ```
   docker container stop trivia-1
   docker container logs trivia-1
   ```

7. When done, remove the trivia container from the system:

   ```
   docker container rm trivia-1
   ```

Now that you know how to retrieve the logs from the command line, let us investigate how we do the same with Docker Desktop.

Displaying container logs in Docker Desktop

Docker Desktop is a very powerful tool, as we have seen. One of the features it offers is to display the logs of an arbitrary container on your system. Here are the step-by-step instructions on how to do this.

1. Open the Dashboard of Docker Desktop.
2. Navigate to the **Images** tab and locate the trivia image. Use the **Run** button to run an instance of the trivia image.
3. On the **Containers/Apps** tab, locate the container in the list that you just started and click on it. Docker Desktop will display the log of this container as shown here.

Note that the output represents the content of the log as it is at the moment when you run the preceding command. Docker will show the whole log from the beginning by default.

4. Now, if the container has been running for a while and we only want the last few lines of the log output, we can use the `--tail` (or short `-t`) parameter:

   ```
   docker container logs --tail 3 trivia-1
   ```

   This will output the last three lines of the log only.

5. If we want to follow the log, we can use the parameter -f to do so. Note that this will block the current terminal window:

   ```
   docker container logs --tail 10 -f trivia-1
   ```

   this will output the last 10 lines of the log collected so far and continue following the log.

   To stop following the log, just press *CTRL-c*.

6. Stop the trivia-1 container and make sure you can still access the log:

   ```
   docker container stop trivia-1
   docker container logs trivia-1
   ```

7. When done, remove the trivia container from the system:

   ```
   docker container rm trivia-1
   ```

Now that you know how to retrieve the logs from the command line, let us investigate how we do the same with Docker Desktop.

Displaying container logs in Docker Desktop

Docker Desktop is a very powerful tool, as we have seen. One of the features it offers is to display the logs of an arbitrary container on your system. Here are the step-by-step instructions on how to do this.

1. Open the Dashboard of Docker Desktop.
2. Navigate to the **Images** tab and locate the trivia image. Use the **Run** button to run an instance of the trivia image.
3. On the **Containers/Apps** tab, locate the container in the list that you just started and click on it. Docker Desktop will display the log of this container as shown here.

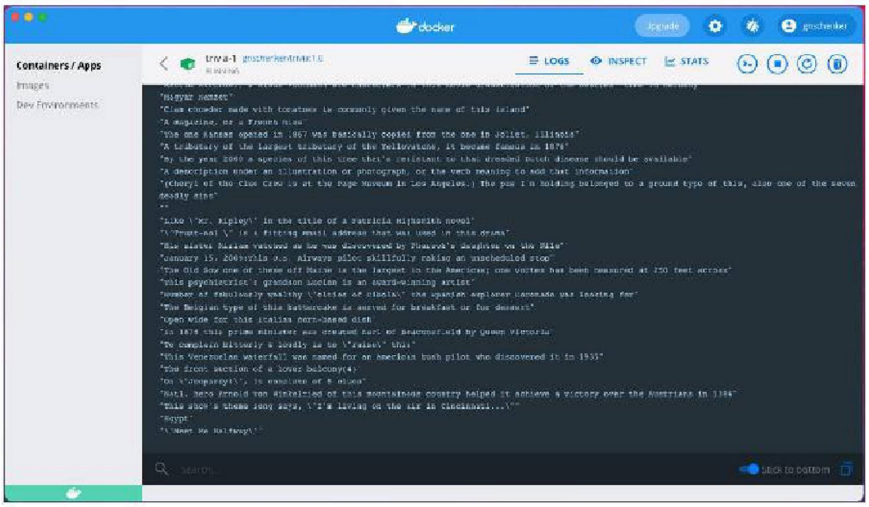


Figure 3.28: Container logs in Docker Desktop

Notice that Docker Desktop follows the log; that is, you always see the fresh entries scrolling by.

4. Use the **STOP** and then the **DELETE** button in the upper right to stop and remove the trivia container.

Finally, we want to know how we can analyze and follow logs in the Docker extension of VS Code.

Displaying container logs in VS Code

The Docker extension of VS Code is similarly powerful as Docker Desktop, and one of its features is that one can inspect and follow the logs of containers residing on the system. Let us do a similar exercise as we did in Docker Desktop.

- Open VS Code and locate and click the Docker whale icon in the left sidebar to display the Docker extension window.
- Under the **IMAGES** section, locate the `gnschenker/trivia:1.0` container and use the context menu on it to run an instance.
- Under the section **CONTAINERS**, locate the newly created and running instance, right-click it to display the context menu, and select **View Logs**
- In the **TERMINAL** output window of VS Code, you should now see the log of the container. The Docker extension follows the log; thus, new entries should scroll by.

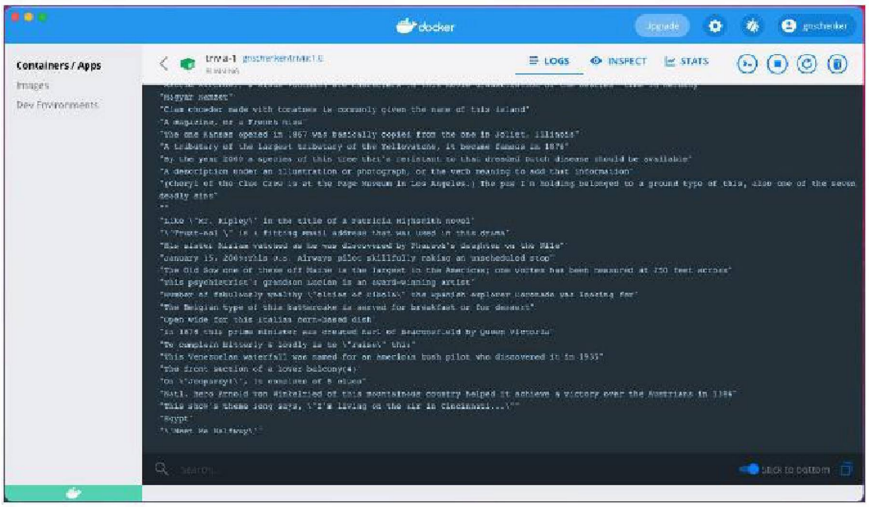


Figure 3.28: Container logs in Docker Desktop

Notice that Docker Desktop follows the log; that is, you always see the fresh entries scrolling by.

4. Use the **STOP** and then the **DELETE** button in the upper right to stop and remove the trivia container.

Finally, we want to know how we can analyze and follow logs in the Docker extension of VS Code.

Displaying container logs in VS Code

The Docker extension of VS Code is similarly powerful as Docker Desktop, and one of its features is that one can inspect and follow the logs of containers residing on the system. Let us do a similar exercise as we did in Docker Desktop.

- Open VS Code and locate and click the Docker whale icon in the left sidebar to display the Docker extension window.
- Under the **IMAGES** section, locate the `gnschenker/trivia:1.0` container and use the context menu on it to run an instance.
- Under the section **CONTAINERS**, locate the newly created and running instance, right-click it to display the context menu, and select **View Logs**
- In the **TERMINAL** output window of VS Code, you should now see the log of the container. The Docker extension follows the log; thus, new entries should scroll by.

Note: the command that the extension uses to follow the log is output in the TERMINAL window at the beginning, right before the log output is listed.

- Right-click the trivia container in the extension window, and in the context menu, select **Stop** and then **Remove** to remove the container from the system.

Now, you know how to get to the logs of any container. This is precious information in case you need to debug an application running inside a container. Together with some helpful Linux tools, such as grep or less, you can quickly find helpful hints on what went wrong or why the application behaved in the way it did, even though the log may be huge.

Next, we will show you how to run any additional process inside an already running container.

Running an additional process inside an already running container

At times we need to run an additional process inside an already running container. This could be to help us debug a misbehaving container. Let us see how we can do this. The first thing we need to know before we can start is the ID or the name of the container of interest. Once we have this command, we need to define which application or process we want to run inside the container. We will use our beloved trivia container to guide you through this step-by-step:

1. Start an instance of the trivia container in detach mode and call it **trivia-1**:

   ```
   docker container run --detach --name trivia-1 gnschenker/trivia:1.0
   ```

2. Run a bash shell interactively inside the trivia-1 container:

   ```
   docker container exec -i -t trivia-1 /bin/bash
   ```

   Notice how in the preceding command, we use the new **exec** command and the two parameters **-i**, which stands for interactive, and **-t**, which stands for a terminal emulator (TTY), meaning that a terminal is connected with this process so the process can interact without terminal window and we will see the output:

3. You should see the following in your terminal:

```
gabriel@Solaris ~ % docker container exec -i -t trivia-1 /bin/bash
root@c4d8d05ee5f1:/#
```

Figure 3.29: Interactive shell running inside the trivia container

Note: the command that the extension uses to follow the log is output in the TERMINAL window at the beginning, right before the log output is listed.

- Right-click the trivia container in the extension window, and in the context menu, select **Stop** and then **Remove** to remove the container from the system.

Now, you know how to get to the logs of any container. This is precious information in case you need to debug an application running inside a container. Together with some helpful Linux tools, such as grep or less, you can quickly find helpful hints on what went wrong or why the application behaved in the way it did, even though the log may be huge.

Next, we will show you how to run any additional process inside an already running container.

Running an additional process inside an already running container

At times we need to run an additional process inside an already running container. This could be to help us debug a misbehaving container. Let us see how we can do this. The first thing we need to know before we can start is the ID or the name of the container of interest. Once we have this command, we need to define which application or process we want to run inside the container. We will use our beloved trivia container to guide you through this step-by-step:

1. Start an instance of the trivia container in detach mode and call it **trivia-1**:

   ```
   docker container run --detach --name trivia-1 gnschenker/trivia:1.0
   ```

2. Run a bash shell interactively inside the trivia-1 container:

   ```
   docker container exec -i -t trivia-1 /bin/bash
   ```

   Notice how in the preceding command, we use the new **exec** command and the two parameters **-i**, which stands for interactive, and **-t**, which stands for a terminal emulator (TTY), meaning that a terminal is connected with this process so the process can interact without terminal window and we will see the output:

3. You should see the following in your terminal:

```
gabriel@Solaris ~ % docker container exec -i -t trivia-1 /bin/bash
root@c4d8d05ee5f1:/#
```

Figure 3.29: Interactive shell running inside the trivia container

Basically, you see a prompt **root@c4d8d05ee5f1:/#**, which means that you are inside the container with hostname **c4d8d05ee5f1** as user **root** in the root folder (/).

4. Execute the Linux command **ls -al** to list the contents of the root folder, and observe that you see a typical Linux file system, as shown in *figure 3.30*. Note that the trivia container is based on an Ubuntu 20.4 distro:

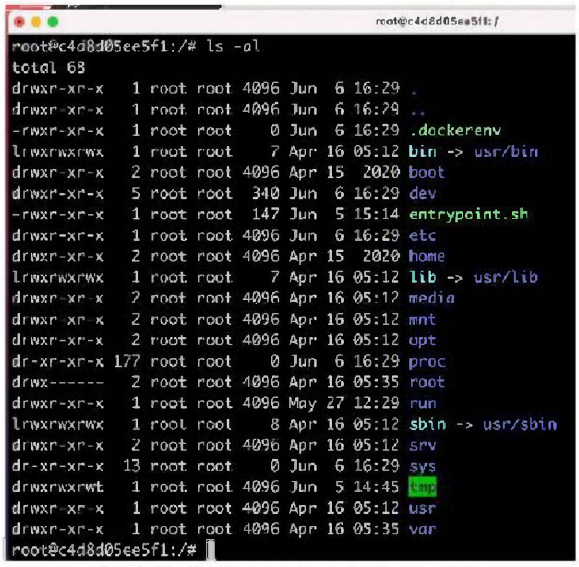


***Figure 3.30**: Contents of the root folder inside the trivia container*

5. Next, execute the Linux command to list all running processes:

   ps

   and observe that you get an output similar to the following shown here:

```
root@c4d8d05ee5f1:/# ps
  PID TTY          TIME CMD
   20 pts/0    00:00:00 bash
  392 pts/0    00:00:00 ps
```

***Figure 3.31**: Listing all processes running inside the trivia container*

   we can see the **bash** shell through which we are interacting with the container and the **ps** command we just ran.

6. **Optional**: Try other Linux commands you know, such as **cd**, **mkdir**, or **find**.

Basically, you see a prompt **root@c4d8d05ee5f1:/#**, which means that you are inside the container with hostname **c4d8d05ee5f1** as user **root** in the root folder (/).

4. Execute the Linux command **ls -al** to list the contents of the root folder, and observe that you see a typical Linux file system, as shown in *figure 3.30*. Note that the trivia container is based on an Ubuntu 20.4 distro:

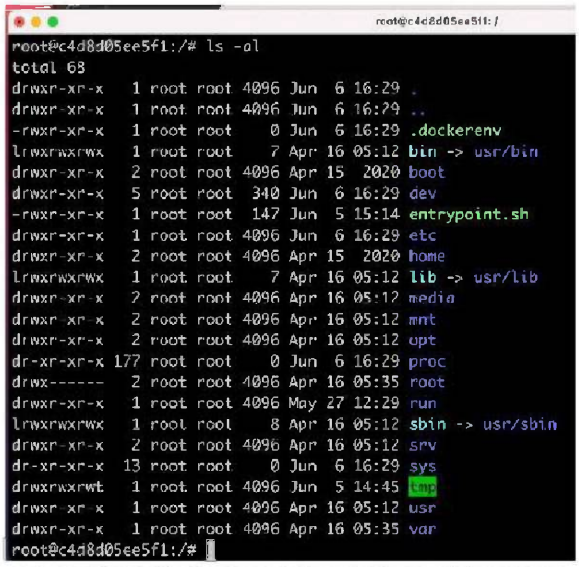


***Figure 3.30**: Contents of the root folder inside the trivia container*

5. Next, execute the Linux command to list all running processes:

   ps

   and observe that you get an output similar to the following shown here:

```
root@c4d8d05ee5f1:/# ps
  PID TTY          TIME CMD
   20 pts/0    00:00:00 bash
  392 pts/0    00:00:00 ps
```

***Figure 3.31**: Listing all processes running inside the trivia container*

   we can see the **bash** shell through which we are interacting with the container and the **ps** command we just ran.

6. **Optional**: Try other Linux commands you know, such as **cd**, **mkdir**, or **find**.

7. To terminate, press *CTRL-d*, which will have you exit the shell.

With that, we are done with this chapter. We have learned a lot. Let us have a look back and recap.

Conclusion

In this chapter, we have learned the basics about Docker containers. The material you have learned so far gives you a very good foundation for your future work with containers. Once again, I recommend you to really make sure you understand every single concept discussed in this chapter before you proceed. This basic knowledge is comparable to what the foundation of a house is. If the foundation is weak or unstable, then the rest of the house will be at risk.

But now, let us recap what we have learned. First, I showed you how to download a Docker image from Docker Hub. Then, you learned how to run a container. We used different types of containers to familiarize ourselves with the run command. You learned to run a container interactively or in the foreground process as well as in detach mode. Next, you learned how to list all the Docker images that currently reside in your local image cache. Then, we explored how to list containers that run on your system. You also learned to list those containers that were stopped yet are still dangling in memory.

Next, we proceeded to how containers can be stopped and removed from the system. We learned not only to do this in the terminal but also via either the dashboard of Docker Desktop or the Docker extension of VS Code.

After this, you learned how to list processes running inside a container and how to view the logs generated by the application running inside a container. Once again, you saw how to do this in the terminal, via Docker Desktop, and via the Docker extension of VS Code. It is always nice to have a choice.

We concluded the chapter by investigating how one can run additional processes inside an already running container. We also showed the use of an interactive bash shell running inside a trivia container.

In the upcoming chapter, you will learn more about how to use existing images for your applications. First, we will show how to run a database locally in a container and access it from another application running in a different container. Then, you will be introduced to Docker volumes that can be used to share data between containers or permanently persist data on media external to a container. Finally, you will be shown how to configure an application running in a container with the help of environment variables and configuration files.

But before you end this chapter, we have prepared a list of questions that may help you assess your learning progress. You will find the correct answer or sample

7. To terminate, press *CTRL-d*, which will have you exit the shell.

With that, we are done with this chapter. We have learned a lot. Let us have a look back and recap.

Conclusion

In this chapter, we have learned the basics about Docker containers. The material you have learned so far gives you a very good foundation for your future work with containers. Once again, I recommend you to really make sure you understand every single concept discussed in this chapter before you proceed. This basic knowledge is comparable to what the foundation of a house is. If the foundation is weak or unstable, then the rest of the house will be at risk.

But now, let us recap what we have learned. First, I showed you how to download a Docker image from Docker Hub. Then, you learned how to run a container. We used different types of containers to familiarize ourselves with the run command. You learned to run a container interactively or in the foreground process as well as in detach mode. Next, you learned how to list all the Docker images that currently reside in your local image cache. Then, we explored how to list containers that run on your system. You also learned to list those containers that were stopped yet are still dangling in memory.

Next, we proceeded to how containers can be stopped and removed from the system. We learned not only to do this in the terminal but also via either the dashboard of Docker Desktop or the Docker extension of VS Code.

After this, you learned how to list processes running inside a container and how to view the logs generated by the application running inside a container. Once again, you saw how to do this in the terminal, via Docker Desktop, and via the Docker extension of VS Code. It is always nice to have a choice.

We concluded the chapter by investigating how one can run additional processes inside an already running container. We also showed the use of an interactive bash shell running inside a trivia container.

In the upcoming chapter, you will learn more about how to use existing images for your applications. First, we will show how to run a database locally in a container and access it from another application running in a different container. Then, you will be introduced to Docker volumes that can be used to share data between containers or permanently persist data on media external to a container. Finally, you will be shown how to configure an application running in a container with the help of environment variables and configuration files.

But before you end this chapter, we have prepared a list of questions that may help you assess your learning progress. You will find the correct answer or sample

answers at the end of the book for each question. If you are not sure about the right answer, do not immediately jump to the solutions but try to go back to the respective section in this book and make sure you got the necessary details.

We also have a list of possible questions that you might be asked during a job interview. For each of the questions, possible answers are given that I would want to hear from a qualified candidate.

Questions

To double-check that you have learned the key concepts of this chapter, we have prepared some questions for you. Try to answer them on your own first, without Googling or looking up the solutions. If you get stuck, we have prepared a sample answer to each question which you can find at the end of this book. Tip: Try to run the commands in your terminal. Make sure that Docker Desktop is running.

1. **Which is the correct command to download the latest version of a Docker image called alpine from the Docker hub into your local image cache? Note that multiple answers may be correct:**
   a. `docker image pull alpine`
   b. `docker pull alpine:latest`
   c. `docker download alpine:latest`
   d. `docker image pull alpine-latest`
   e. `docker run alpine:latest`

2. **How can you list all containers on your local machine, including the stopped ones? Note that multiple answers may be correct:**
   a. `docker container ls --all`
   b. `docker ps`
   c. `docker ps -a`
   d. `docker list all`
   e. `docker containers list --all`

3. **You have a container with the name `my-app` running and want to remove you with the command `docker rm my-app`. What will happen if you execute this command?**
   a. The container is stopped and removed from the system
   b. An error is reported, and the container continues to run
   c. An error is reported, and the container stops
   d. The command is ignored

answers at the end of the book for each question. If you are not sure about the right answer, do not immediately jump to the solutions but try to go back to the respective section in this book and make sure you got the necessary details.

We also have a list of possible questions that you might be asked during a job interview. For each of the questions, possible answers are given that I would want to hear from a qualified candidate.

Questions

To double-check that you have learned the key concepts of this chapter, we have prepared some questions for you. Try to answer them on your own first, without Googling or looking up the solutions. If you get stuck, we have prepared a sample answer to each question which you can find at the end of this book. Tip: Try to run the commands in your terminal. Make sure that Docker Desktop is running.

1. **Which is the correct command to download the latest version of a Docker image called alpine from the Docker hub into your local image cache? Note that multiple answers may be correct:**
   a. `docker image pull alpine`
   b. `docker pull alpine:latest`
   c. `docker download alpine:latest`
   d. `docker image pull alpine-latest`
   e. `docker run alpine:latest`

2. **How can you list all containers on your local machine, including the stopped ones? Note that multiple answers may be correct:**
   a. `docker container ls --all`
   b. `docker ps`
   c. `docker ps -a`
   d. `docker list all`
   e. `docker containers list --all`

3. **You have a container with the name `my-app` running and want to remove you with the command `docker rm my-app`. What will happen if you execute this command?**
   a. The container is stopped and removed from the system
   b. An error is reported, and the container continues to run
   c. An error is reported, and the container stops
   d. The command is ignored

4. **How can you stop and remove a running container called my-webapp from your system? Note that multiple answers may be correct:**
   a. `docker stop my-webapp && docker rm my-webapp`
   b. `docker kill my-webapp`
   c. `docker kill my-webapp && docker rm my-webapp`
   d. `docker my-webapp rm –force`

5. **What is the difference between `docker stop` and `docker kill` commands?**
   a. There is none. They are equivalent
   b. `docker stop` immediately terminates a container
   c. `docker kill` sends a `SIGTERM` signal to the main process of the running container
   d. No answer is correct

6. **How can you get a list of all processes running inside a container called my-app:**
   a. `docker process ls`
   b. `docker container top`
   c. `docker list processes`
   d. `docker container ps`
   e. This is not possible since there is no such command

7. **How can you retrieve the last 10 log entries of a container called my-app and keep following the log?**
   a. `docker logs my-app -n 10 --follow`
   b. `docker my-app logs --tail -f 10`
   c. `docker logs my-app --tail 10 --follow`
   d. `docker log --tail 10 --follow my-app`
   e. It is not possible to do such a thing in one command

8. **How can you execute a ping 8.8.8.8 (where the IP address 8.8.8.8 is the Google DNS) command within the context of an already running container called my-microservice-1?**
   a. `docker run my-microservice-1 ping 8.8.8.8`
   b. `docker my-microservice-1 exec ping 8.8.8.8`

4. **How can you stop and remove a running container called my-webapp from your system? Note that multiple answers may be correct:**
   a. `docker stop my-webapp && docker rm my-webapp`
   b. `docker kill my-webapp`
   c. `docker kill my-webapp && docker rm my-webapp`
   d. `docker my-webapp rm -force`

5. **What is the difference between `docker stop` and `docker kill` commands?**
   a. There is none. They are equivalent
   b. `docker stop` immediately terminates a container
   c. `docker kill` sends a `SIGTERM` signal to the main process of the running container
   d. No answer is correct

6. **How can you get a list of all processes running inside a container called my-app:**
   a. `docker process ls`
   b. `docker container top`
   c. `docker list processes`
   d. `docker container ps`
   e. This is not possible since there is no such command

7. **How can you retrieve the last 10 log entries of a container called my-app and keep following the log?**
   a. `docker logs my-app -n 10 --follow`
   b. `docker my-app logs --tail -f 10`
   c. `docker logs my-app --tail 10 --follow`
   d. `docker log --tail 10 --follow my-app`
   e. It is not possible to do such a thing in one command

8. **How can you execute a ping 8.8.8.8 (where the IP address 8.8.8.8 is the Google DNS) command within the context of an already running container called my-microservice-1?**
   a. `docker run my-microservice-1 ping 8.8.8.8`
   b. `docker my-microservice-1 exec ping 8.8.8.8`

c. `docker container exec my-microservice-1 ping 8.8.8.8`

d. `docker container run my-microservice-1 ping 8.8.8.8`

Answers

1. Answers a, b, and e are correct. Note that the run command implicitly pulls the image if it is not already there
2. Answers a, c, and e are correct
3. Answer b is correct; an error is reported, and the container continues to run
4. Answer a and c are correct; first, stop or kill and then remove the container
5. Answer d is correct. The kill command sends a SIGKILL signal to the main process.
6. Answer b is correct
7. a, c, and d are correct (the short form of --tail is -n)
8. Answer c is correct

Job interview sample questions

Once again, you got a chance for an onsite interview for a job as a software developer. The job is offered by yet another start-up that develops a SaaS solution in the financial sector. During the interview, you are confronted with the following questions:

1. Is it possible to execute an additional process in an already running container? If yes, name 1 or 2 scenarios where that would make sense.
2. Explain what happens if you run a container from a container image that is not yet in the local cache of the computer or server on which the container is supposed to run.
3. An application that is supposed to run inside a container is generating some logs for monitoring and debugging purposes. Where should the application write the log entries to such as Docker can then easily pick them up at provide them to you when you execute the appropriate command in the Docker CLI?

Sample answers

1. Yes, it is possible, using the `docker container exec` command. Possible scenarios are as follows:
    - Running an interactive shell to explore the contents of a container, e.g., during development

c. `docker container exec my-microservice-1 ping 8.8.8.8`

d. `docker container run my-microservice-1 ping 8.8.8.8`

Answers

1. Answers a, b, and e are correct. Note that the run command implicitly pulls the image if it is not already there
2. Answers a, c, and e are correct
3. Answer b is correct; an error is reported, and the container continues to run
4. Answer a and c are correct; first, stop or kill and then remove the container
5. Answer d is correct. The kill command sends a SIGKILL signal to the main process.
6. Answer b is correct
7. a, c, and d are correct (the short form of --tail is -n)
8. Answer c is correct

Job interview sample questions

Once again, you got a chance for an onsite interview for a job as a software developer. The job is offered by yet another start-up that develops a SaaS solution in the financial sector. During the interview, you are confronted with the following questions:

1. Is it possible to execute an additional process in an already running container? If yes, name 1 or 2 scenarios where that would make sense.
2. Explain what happens if you run a container from a container image that is not yet in the local cache of the computer or server on which the container is supposed to run.
3. An application that is supposed to run inside a container is generating some logs for monitoring and debugging purposes. Where should the application write the log entries to such as Docker can then easily pick them up at provide them to you when you execute the appropriate command in the Docker CLI?

Sample answers

1. Yes, it is possible, using the `docker container exec` command. Possible scenarios are as follows:
    - Running an interactive shell to explore the contents of a container, e.g., during development

- Running a Linux network utility such as `ping` or `netcat` (nc) to debug issues with the network and connectivity of an app running inside a container
- Running a command periodically to check the health of the app running inside a container, e.g., a `curl` command if the app is a Web API

2. The Docker CLI analyses the run command you entered and checks if the image you named is already in the local image cache. In this case, the result is "No", and thus, the CLI automatically tries to download the image from the configured container registry, which by default is Docker Hub. If the download to the local image cache is successful, the image is loaded into memory, and a container instance is created and run.
3. Each containerized application should output its logs either to the standard output (`STDOUT`) or standard error output (`STDERR`). From there, Docker can pick them up and store them automatically for you. Later on, one can use the `docker container logs` command to retrieve this log.

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



- Running a Linux network utility such as `ping` or `netcat` (nc) to debug issues with the network and connectivity of an app running inside a container
- Running a command periodically to check the health of the app running inside a container, e.g., a `curl` command if the app is a Web API

2. The Docker CLI analyses the run command you entered and checks if the image you named is already in the local image cache. In this case, the result is "No", and thus, the CLI automatically tries to download the image from the configured container registry, which by default is Docker Hub. If the download to the local image cache is successful, the image is loaded into memory, and a container instance is created and run.
3. Each containerized application should output its logs either to the standard output (`STDOUT`) or standard error output (`STDERR`). From there, Docker can pick them up and store them automatically for you. Later on, one can use the `docker container logs` command to retrieve this log.

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 4
Using Existing Docker Images

Introduction

In the previous chapter, you learned how to work with containers. You learned how to start, stop, and remove containers. You familiarized yourself with commands to retrieve the logs of a container, execute commands inside an already running container, and more. At this time, you should start to feel comfortable playing around with containers.

In this chapter, you will learn how to use existing container images, such as ones for SQL databases, locally on your computer. You will learn how an image is made up of layers, and then 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.

Structure

In this chapter, we will discuss the following topics:

- Running a SQL database locally
- Investigating the image layers model
- Using volumes with containers

CHAPTER 4
Using Existing Docker Images

Introduction

In the previous chapter, you learned how to work with containers. You learned how to start, stop, and remove containers. You familiarized yourself with commands to retrieve the logs of a container, execute commands inside an already running container, and more. At this time, you should start to feel comfortable playing around with containers.

In this chapter, you will learn how to use existing container images, such as ones for SQL databases, locally on your computer. You will learn how an image is made up of layers, and then 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.

Structure

In this chapter, we will discuss the following topics:

- Running a SQL database locally
- Investigating the image layers model
- Using volumes with containers

- Configuring an app running inside a container

Objectives

After reading this chapter, you will be able to run a PostgreSQL, MongoDB, or any similar relational or document database locally in a container. You will be able to explain how a Docker image is layered and the advantages that this brings to the table to an interested layperson. You will be able to write and containerize a simple application written in Node JS or Python that accesses and uses this locally running database. You can create Docker volumes and mount them to a container. You are able to share data between containers using Docker volumes. Furthermore, you are able to mount a local folder into a container, which will be useful when developing, running, and debugging an application inside a container. Finally, you can configure environment variables inside a container either explicitly or by using an env file

Without further ado, let us start.

Running a SQL database locally

One of the most challenging and daunting tasks for a developer has always been to install a database such as Postgres or MongoDB on their local machine. I can remember the days when Oracle was still ruling the database world and when it took me a day to install and configure an Oracle DB on my PC.

This is not necessary anymore. It is super easy to run nearly any of the popular databases in a container on any system, not just your own laptop or desktop. These days enthusiasts and hobbyists run databases even on a Raspberry PI.

Running a PostgreSQL database server

Let us see how we can do that. To start, let us use one of the more popular choices among developers these days, the PostgreSQL database. We can find a well-maintained and curated version of this database on Docker Hub under the following link:

https://hub.docker.com/_/postgres

There we also find a short description on how to use and configure PostgreSQL. But we will use our knowledge from the previous chapter to understand what needs to be in a command to run an instance of the database.

Before we start, we will have to create a Docker network. Do not worry about the details, as we will explain networks in great detail in *Chapter 6, Demystifying container networking*. Use this command to create a network called `my-network`:

```
docker network create my-network
```

- Configuring an app running inside a container

Objectives

After reading this chapter, you will be able to run a PostgreSQL, MongoDB, or any similar relational or document database locally in a container. You will be able to explain how a Docker image is layered and the advantages that this brings to the table to an interested layperson. You will be able to write and containerize a simple application written in Node JS or Python that accesses and uses this locally running database. You can create Docker volumes and mount them to a container. You are able to share data between containers using Docker volumes. Furthermore, you are able to mount a local folder into a container, which will be useful when developing, running, and debugging an application inside a container. Finally, you can configure environment variables inside a container either explicitly or by using an env file

Without further ado, let us start.

Running a SQL database locally

One of the most challenging and daunting tasks for a developer has always been to install a database such as Postgres or MongoDB on their local machine. I can remember the days when Oracle was still ruling the database world and when it took me a day to install and configure an Oracle DB on my PC.

This is not necessary anymore. It is super easy to run nearly any of the popular databases in a container on any system, not just your own laptop or desktop. These days enthusiasts and hobbyists run databases even on a Raspberry PI.

Running a PostgreSQL database server

Let us see how we can do that. To start, let us use one of the more popular choices among developers these days, the PostgreSQL database. We can find a well-maintained and curated version of this database on Docker Hub under the following link:

https://hub.docker.com/_/postgres

There we also find a short description on how to use and configure PostgreSQL. But we will use our knowledge from the previous chapter to understand what needs to be in a command to run an instance of the database.

Before we start, we will have to create a Docker network. Do not worry about the details, as we will explain networks in great detail in *Chapter 6, Demystifying container networking*. Use this command to create a network called `my-network`:

```
docker network create my-network
```

That should look like this in your terminal:

```
gabriel@Solaris ~ % docker network create my-network
b2084785f3763a6701e6e931363007906b40ef13b5076fcba9f6c84583210c8a
```

Figure 4.1: Create a Docker network called my-network

Docker creates the network and simply answers by printing the ID of the newly created network.

Now have a look at the following command:

```
docker container run --network my-network postgres:latest
```

It says that we want to run an instance of the `postgres` Docker image. This will work and will run an instance of the `latest` version of the PostgreSQL Docker image. Since I have not been running this image before on my system, I see the following in my terminal:

```
gabriel@Solaris ~ % docker container run postgres:latest
Unable to find image 'postgres:latest' locally
latest: Pulling from library/postgres
fcad0c936ea5: Already exists
6f8752e7c3ca: Pull complete
55ed0202527f: Pull complete
8f57daa18143: Pull complete
f49772502de9: Pull complete
f5eb0128e6d8: Pull complete
8d16ebd51906: Pull complete
2ecdf6fa5cc9: Pull complete
c12779590507: Pull complete
8a1865a31a87: Pull complete
dfdb3072349f: Pull complete
8e9e5d018281: Pull complete
13f069a90642: Pull complete
Digest: sha256:a8f25ca44e98a4846cad176be8017f8f9c3402Be4eebbf905dd46ccbab77d7a2
Status: Downloaded newer image for postgres:latest
Error: Database is uninitialized and superuser password is not specified.
       You must specify POSTGRES_PASSWORD to a non-empty value for the
       superuser. For example, "-e POSTGRES_PASSWORD=password" on "docker run'.

       You may also use "POSTGRES_HOST_AUTH_METHOD=trust" to allow all
       connections without a password. This is *not* recommended.

       See PostgreSQL documentation about "trust":
       https://www.postgresql.org/docs/current/auth-trust.html
```

Figure 4.2: Running PostgreSQL for the first time

The first part of the output shows us that Docker downloaded the image from the Docker hub and that the image is composed of many layers, all having their own ID, e.g., the first one having `6f8752e7c3ca`. But there is also an error message at the end of the output indicating that we need to pass a master password for the database. The error message even indicates how we can do that. Let us, thus, improve our command such that it looks like the following:

```
docker container run -e POSTGRES_PASSWORD=Top-Secret postgres:latest
```

That should look like this in your terminal:

```
gabriel@Solaris ~ % docker network create my-network
b2084785f3763a6701e6e931363007906b40ef13b5076fcba9f6c84583210c8a
```

Figure 4.1: Create a Docker network called my-network

Docker creates the network and simply answers by printing the ID of the newly created network.

Now have a look at the following command:

```
docker container run --network my-network postgres:latest
```

It says that we want to run an instance of the `postgres` Docker image. This will work and will run an instance of the `latest` version of the PostgreSQL Docker image. Since I have not been running this image before on my system, I see the following in my terminal:

```
gabriel@Solaris ~ % docker container run postgres:latest
Unable to find image 'postgres:latest' locally
latest: Pulling from library/postgres
fcad0c936ea5: Already exists
6f8752e7c3ca: Pull complete
55ed0202527f: Pull complete
8f57daa18143: Pull complete
f49772502de9: Pull complete
f5eb0128e6d8: Pull complete
8d16ebd51906: Pull complete
2ecdf6fa5cc9: Pull complete
c12779590507: Pull complete
8a1865a31a87: Pull complete
dfdb3072349f: Pull complete
8e9e5d018281: Pull complete
13f069a90642: Pull complete
Digest: sha256:a8f25ca44e98a4846cad176be8017f8f9c3402Be4eebbf905dd46ccbab77d7a2
Status: Downloaded newer image for postgres:latest
Error: Database is uninitialized and superuser password is not specified.
       You must specify POSTGRES_PASSWORD to a non-empty value for the
       superuser. For example, "-e POSTGRES_PASSWORD=password" on "docker run'.

       You may also use "POSTGRES_HOST_AUTH_METHOD=trust" to allow all
       connections without a password. This is *not* recommended.

       See PostgreSQL documentation about "trust":
       https://www.postgresql.org/docs/current/auth-trust.html
```

Figure 4.2: Running PostgreSQL for the first time

The first part of the output shows us that Docker downloaded the image from the Docker hub and that the image is composed of many layers, all having their own ID, e.g., the first one having `6f8752e7c3ca`. But there is also an error message at the end of the output indicating that we need to pass a master password for the database. The error message even indicates how we can do that. Let us, thus, improve our command such that it looks like the following:

```
docker container run -e POSTGRES_PASSWORD=Top-Secret postgres:latest
```

If we run the preceding command (first part of), the output will look like this:

```
gabriel@Solaris ~ % docker container run -e POSTGRES_PASSWORD=Top-Secret postgre
s:latest
The files belongirg to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale "en_US.utf8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".

Data page checksums are disabled.

fixing permissions on existing directory /var/lib/postgresql/data ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default max_connections ... 100
selecting default shared buffers ... 128MB
```

Figure 4.3: Running PostgreSQL and defining the root password

This time there was no need to download the image again from the Docker hub; thus, Docker just took the one from the cache. The container started, and with it, the database application that was packaged inside the container. The preceding (shortened) output stems from the database application's initialization process. Once the database is fully initialized, the output stops, and the database waits for any incoming connection.

But by the way, we have started the database, and it is now blocking our terminal. We ran the container and, with it, the database in the foreground process. This may be interesting if you want to debug the usage of your database because you can directly watch the output of it in the terminal as it happens. But normally, you would probably want to run the database in detach mode, that is, in the background. Let us do this. But first, we have to stop the current version by pressing *CTRL-c* in the terminal:

```
^C2021-06-20 13:34:08.949 UTC [1] LOG:  received fast shutdown request
2021-06-20 13:34:08.951 UTC [1] LOG:  aborting any active transactions
2021-06-20 13:34:08.955 UTC [1] LOG:  background worker "logical replication lau
ncher" (PID 73) exited with exit code 1
2021-06-20 13:34:08.960 UTC [68] LOG:  shutting down
2021-06-20 13:34:08.978 UTC [1] LOG:  database system is shut down
gabriel@Solaris ~ %
```

Figure 4.4: Terminating the database with CTRL-c

As you can see previously, the database immediately reacted to the *CTRL-c* command and caused a fast shutdown. Once the database application has ended, the container also stops, and we are free to use our terminal window again.

Now let us run the database again, but this time in detach mode:

If we run the preceding command (first part of), the output will look like this:

```
gabriel@Solaris ~ % docker container run -e POSTGRES_PASSWORD=Top-Secret postgre
s:latest
The files belongirg to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale "en_US.utf8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".

Data page checksums are disabled.

fixing permissions on existing directory /var/lib/postgresql/data ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
```

Figure 4.3: Running PostgreSQL and defining the root password

This time there was no need to download the image again from the Docker hub; thus, Docker just took the one from the cache. The container started, and with it, the database application that was packaged inside the container. The preceding (shortened) output stems from the database application's initialization process. Once the database is fully initialized, the output stops, and the database waits for any incoming connection.

But by the way, we have started the database, and it is now blocking our terminal. We ran the container and, with it, the database in the foreground process. This may be interesting if you want to debug the usage of your database because you can directly watch the output of it in the terminal as it happens. But normally, you would probably want to run the database in detach mode, that is, in the background. Let us do this. But first, we have to stop the current version by pressing *CTRL-c* in the terminal:

```
^C2021-06-20 13:34:08.949 UTC [1] LOG:  received fast shutdown request
2021-06-20 13:34:08.951 UTC [1] LOG:  aborting any active transactions
2021-06-20 13:34:08.955 UTC [1] LOG:  background worker "logical replication lau
ncher" (PID 73) exited with exit code 1
2021-06-20 13:34:08.960 UTC [68] LOG:  shutting down
2021-06-20 13:34:08.978 UTC [1] LOG:  database system is shut down
gabriel@Solaris ~ %
```

Figure 4.4: Terminating the database with CTRL-c

As you can see previously, the database immediately reacted to the *CTRL-c* command and caused a fast shutdown. Once the database application has ended, the container also stops, and we are free to use our terminal window again.

Now let us run the database again, but this time in detach mode:

```
docker container run \
    -e POSTGRES_PASSWORD=Top-Secret \
    --detach \
    postgres:latest
```

When running the preceding command, the container will launch in a background process, and the terminal is ready for further input, as shown here:

```
gabriel@Solaris ~ % docker container run \
> -e POSTGRES_PASSWORD=Top-Secret \
> --detach \
> postgres:latest
dd26521ef672be4383f23f316c979631a4a6adac295bfcbb7efb9c2ebe52ce09
gabriel@Solaris ~ %
```

Figure 4.5: Running PostgreSQL in detach mode

Our database is now running in the background, and the container got an ID of `dd2652...`

Note that I have used the backslash (\) as a line continuation symbol to be able to spread one logical command across several lines.

Attention: All Windows users working with PowerShell need to replace the backslash with a backtick (`) such that the command looks like the following:

```
docker container run `
    -e POSTGRES_PASSWORD=Top-Secret `
    --detach `
    postgres:latest
```

Throughout this book, we will use this technique often to improve the readability of the commands. Thus, please remember that a backslash is the line continuation symbol in Linux/Unix or Bash shell on Windows, whereas the backtick is the same in PowerShell.

In the preceding command, we have not specified a name for the container, though, and Docker will assign a random one to this instance. We can use the ID of the container (`dd2652...`) to refer to it, but often it is more convenient to assign a specific name to the container. Specifically, when working with databases, this makes sense as we can then access it via a friendly name rather than with a cryptic ID. Let us once more adjust our run command and name the container `my-postgres`; furthermore, we attach it to the network `my-network`:

```
docker container run \
    -e POSTGRES_PASSWORD=Top-Secret \
    --detach \
    --name my-postgres \
    --network my-network \
    postgres:latest
```

```
docker container run \
    -e POSTGRES_PASSWORD=Top-Secret \
    --detach \
    postgres:latest
```

When running the preceding command, the container will launch in a background process, and the terminal is ready for further input, as shown here:

```
gabriel@Solaris ~ % docker container run \
> -e POSTGRES_PASSWORD=Top-Secret \
> --detach \
> postgres:latest
dd26521ef672be4383f23f316c979631a4a6adac295bfcbb7efb9c2ebe52ce09
gabriel@Solaris ~ %
```

Figure 4.5: Running PostgreSQL in detach mode

Our database is now running in the background, and the container got an ID of `dd2652...`

Note that I have used the backslash (\) as a line continuation symbol to be able to spread one logical command across several lines.

Attention: All Windows users working with PowerShell need to replace the backslash with a backtick (`) such that the command looks like the following:

```
docker container run `
    -e POSTGRES_PASSWORD=Top-Secret `
    --detach `
    postgres:latest
```

Throughout this book, we will use this technique often to improve the readability of the commands. Thus, please remember that a backslash is the line continuation symbol in Linux/Unix or Bash shell on Windows, whereas the backtick is the same in PowerShell.

In the preceding command, we have not specified a name for the container, though, and Docker will assign a random one to this instance. We can use the ID of the container (`dd2652...`) to refer to it, but often it is more convenient to assign a specific name to the container. Specifically, when working with databases, this makes sense as we can then access it via a friendly name rather than with a cryptic ID. Let us once more adjust our run command and name the container `my-postgres`; furthermore, we attach it to the network `my-network`:

```
docker container run \
    -e POSTGRES_PASSWORD=Top-Secret \
    --detach \
    --name my-postgres \
    --network my-network \
    postgres:latest
```

As we will see later, we have to run the PostgreSQL container attached to a network. This is necessary so that other containerized applications running on the same network can access the database. We do this using the `--network` parameter. Do not worry about the details of container networks yet, as we will explain everything in *Chapter 6, Demystifying container networking*.

We have run a few instances of PostgreSQL so far. If we list our containers, we may see something like the following:



Figure 4.6: Listing all containers

Note how in my list command, I used the `-a` parameter to list all containers, even the stopped ones. In the preceding output, we can see two stopped and two running instances. The first in the list is the one I started last, and you can see that it has the name `my-postgres`. The other instances have random names that were assigned by Docker and are composed of an adjective and the name of a famous scientist.

We should also realize that the preceding list of running containers indicates to us that we can have multiple instances of a PostgreSQL database server running on our laptop or desktop. We can do so without needing to worry about any potential conflicts between the database servers. The fact that they run inside a container shields them perfectly from each other.

***Task:** Stop and remove all container instances other than the one called* `my-postgres`*. Do this in whatever way you prefer, through the command line, the Docker Desktop dashboard, or the VS Code Docker extension.*

Working with the database

Just running a database server is fun and interesting, but beyond that, what can we do with it? The first thing that comes to mind is using the PostgreSQL **command line interface (CLI)** with which we can create databases, run **data definition language (DDL)**, and **data resource management (DRM)** type of statements against the database. It turns out that our Postgres container contains not only the database server application but also the binaries of the CLI. Remember how we learned to run a process inside an already running container? Yes, right; finally, this makes sense, and we will use exactly this technique to run the Postgres CLI inside our `my-postgres` container. Execute the following command to do so:

```
docker container exec -it my-postgres psql -U postgres
```

As we will see later, we have to run the PostgreSQL container attached to a network. This is necessary so that other containerized applications running on the same network can access the database. We do this using the `--network` parameter. Do not worry about the details of container networks yet, as we will explain everything in *Chapter 6, Demystifying container networking*.

We have run a few instances of PostgreSQL so far. If we list our containers, we may see something like the following:



Figure 4.6: Listing all containers

Note how in my list command, I used the `-a` parameter to list all containers, even the stopped ones. In the preceding output, we can see two stopped and two running instances. The first in the list is the one I started last, and you can see that it has the name `my-postgres`. The other instances have random names that were assigned by Docker and are composed of an adjective and the name of a famous scientist.

We should also realize that the preceding list of running containers indicates to us that we can have multiple instances of a PostgreSQL database server running on our laptop or desktop. We can do so without needing to worry about any potential conflicts between the database servers. The fact that they run inside a container shields them perfectly from each other.

***Task:** Stop and remove all container instances other than the one called* `my-postgres`*. Do this in whatever way you prefer, through the command line, the Docker Desktop dashboard, or the VS Code Docker extension.*

Working with the database

Just running a database server is fun and interesting, but beyond that, what can we do with it? The first thing that comes to mind is using the PostgreSQL **command line interface (CLI)** with which we can create databases, run **data definition language (DDL)**, and **data resource management (DRM)** type of statements against the database. It turns out that our Postgres container contains not only the database server application but also the binaries of the CLI. Remember how we learned to run a process inside an already running container? Yes, right; finally, this makes sense, and we will use exactly this technique to run the Postgres CLI inside our `my-postgres` container. Execute the following command to do so:

```
docker container exec -it my-postgres psql -U postgres
```

And you should find yourself inside the container in a running CLI as a user **postgres**, as shown here:

```
gabriel@Solaris ~ % docker container exec -it my-postgres psql -U postgres
psql (13.3 (Debian 13.3-1.pgdg100+1))
Type "help" for help.

postgres=#
```

Figure 4.7: Running the Posigres CLI inside the Postgres database container

We can now use valid Postgres commands to, e.g., create a database **demo** and, within **demo**, a table **users**:

```
postgres=# create database demo;
CREATE DATABASE
postgres=# \connect demo;
You are now connected to database "demo" as user "postgres".
demo=# create table users(
demo(# id int not null,
demo(# name varchar(59) unique not null
demo(# );
CREATE TABLE
demo=# select * from users;
 id | name
----+------
(0 rows)

demo=#
```

Figure 4.8: Using the psql CLI to create a DB and a table

In the preceding output, you can see how we have used this sequence of commands to achieve this result:

```
create database demo;

\connect demo;

create table users (
    id int not null,
    name varchar(50) unique not null
);
select * from users;
```

Note that each command in **psql** is terminated with a semicolon (**;**) and that PostgreSQL does not distinguish between keywords such as "**create**" being written in lower or upper case.

You can also try to insert some data into the table users and list the content of the table:

And you should find yourself inside the container in a running CLI as a user **postgres**, as shown here:

```
gabriel@Solaris ~ % docker container exec -it my-postgres psql -U postgres
psql (13.3 (Debian 13.3-1.pgdg100+1))
Type "help" for help.

postgres=#
```

Figure 4.7: Running the Posigres CLI inside the Postgres database container

We can now use valid Postgres commands to, e.g., create a database **demo** and, within **demo**, a table **users**:

```
postgres=# create database demo;
CREATE DATABASE
postgres=# \connect demo;
You are now connected to database "demo" as user "postgres".
demo=# create table users(
demo(# id int not null,
demo(# name varchar(59) unique not null
demo(# );
CREATE TABLE
demo=# select * from users;
 id | name
----+------
(0 rows)

demo=#
```

Figure 4.8: Using the psql CLI to create a DB and a table

In the preceding output, you can see how we have used this sequence of commands to achieve this result:

```
create database demo;

\connect demo;

create table users (
    id int not null,
    name varchar(50) unique not null
);
select * from users;
```

Note that each command in **psql** is terminated with a semicolon (**;**) and that PostgreSQL does not distinguish between keywords such as "**create**" being written in lower or upper case.

You can also try to insert some data into the table users and list the content of the table:

```
insert into users(id, name) values(1, 'Ann Hunter');
insert into users(id, name) values(2, 'Suzanne Baker');
insert into users(id, name) values(3, 'Will Smith');
select * from users;
```

Try it out.

To end your session with *psql* just press *CTRL-d* and you will be back in your terminal session from where you started *psql* in the first place.

Accessing the database from another container

So, now you know how to run a PostgreSQL database server on your local computer and how to use the *psql* CLI inside that container to manage the databases, schemas, and data of the database server. But usually, that is not the main usage scenario of a database. Normally, you will want to write an application that runs in a different container (or directly on your host) that accesses the database. Let us now investigate how that is possible. We will start by using the *psql* CLI running inside a different container and show how we can access the database running in the `my-postgres` container.

Run the following command:

```
docker container run \
    --network my-network \
    -e PGPASSWORD=Top-Secret \
    --rm \
    -it \
    postgres:latest psql -h my-postgres -U postgres
```

That is, we run a different instance of the `postgres` image and attach it to the same network (`my-network`) as the database server, so they can communicate with each other. As we will see, only applications that are running in containers that are attached to the same network can ever communicate with each other.

Task: Google what the parameters `--rm` *and* `-it` *are used for in the preceding Docker command.*

Instead of letting the container use the default start command—which would start a PostgreSQL server instance—we pass it a different command to execute, namely:

```
psql -h my-postgres -U postgres
```

This command starts `psql` and connects to a host called `my-postgres` (our Postgres server) with user `postgres`. It turns out that we also need to provide a password to the CLI so it can login into the remote server. We do this with the `-e PGPASSWORD=Top-Secret` parameter, which defines an environment variable called `PGPASSWORD` with

```
insert into users(id, name) values(1, 'Ann Hunter');
insert into users(id, name) values(2, 'Suzanne Baker');
insert into users(id, name) values(3, 'Will Smith');
select * from users;
```

Try it out.

To end your session with *psql* just press *CTRL-d* and you will be back in your terminal session from where you started *psql* in the first place.

Accessing the database from another container

So, now you know how to run a PostgreSQL database server on your local computer and how to use the *psql* CLI inside that container to manage the databases, schemas, and data of the database server. But usually, that is not the main usage scenario of a database. Normally, you will want to write an application that runs in a different container (or directly on your host) that accesses the database. Let us now investigate how that is possible. We will start by using the *psql* CLI running inside a different container and show how we can access the database running in the `my-postgres` container.

Run the following command:

```
docker container run \
    --network my-network \
    -e PGPASSWORD=Top-Secret \
    --rm \
    -it \
    postgres:latest psql -h my-postgres -U postgres
```

That is, we run a different instance of the `postgres` image and attach it to the same network (`my-network`) as the database server, so they can communicate with each other. As we will see, only applications that are running in containers that are attached to the same network can ever communicate with each other.

Task: Google what the parameters `--rm` *and* `-it` *are used for in the preceding Docker command.*

Instead of letting the container use the default start command—which would start a PostgreSQL server instance—we pass it a different command to execute, namely:

```
psql -h my-postgres -U postgres
```

This command starts `psql` and connects to a host called `my-postgres` (our Postgres server) with user `postgres`. It turns out that we also need to provide a password to the CLI so it can login into the remote server. We do this with the `-e PGPASSWORD=Top-Secret` parameter, which defines an environment variable called `PGPASSWORD` with

the value `Top-Secret` inside the container. The `psql` CLI will then pick this value up when it connects to the remote server `my-postgres`.

If you run the preceding command, you should see this in your terminal:

```
gabriel@Solaris ~ % docker container run \
   --network my-network \
   --rm \
   -e PGPASSWORD=Top-Secret \
   -it postgres:latest psql -h my-postgres -U postgres
psql (13.3 (Debian 13.3-1.pgdg100+1))
Type "help" for help.

postgres=#
```

Figure 4.9: Running the psql CLI in a container and connecting to a remote Postgres server

You can now use `\connect demo` to connect to the demo database and then use the `select * from users;` command to list all records that you added previously to this database:

```
postgres=# \connect demo;
You are now connected to database "demo" as user "postgres".
demo=# select * from users;
 id |     name
----+--------------
  1 | Ann Hunter
  2 | Suzanne Baker
  3 | Will Smith
(3 rows)

demo-#
```

Figure 4.10: Using psql to work with our remote DB

When you are done experimenting with the `psql` CLI, you can exit it and, at the same time, stop the container by pressing *CTRL-d*.

A simple node JS app to access PostgreSQL

Let us now create a very simple Node JS application that we will be running from within a container attached to the `my-network` network and accessing the PostgreSQL database server.

Proceed as follows:

1. In your `src` folder, create a new folder called `ch04/node-client` and navigate to it:

   ```
   mkdir ch04/node-client && cd ch04/node-client
   ```

the value `Top-Secret` inside the container. The `psql` CLI will then pick this value up when it connects to the remote server `my-postgres`.

If you run the preceding command, you should see this in your terminal:

```
gabriel@Solaris ~ % docker container run \
   --network my-network \
   --rm \
   -e PGPASSWORD=Top-Secret \
   -it postgres:latest psql -h my-postgres -U postgres
psql (13.3 (Debian 13.3-1.pgdg100+1))
Type "help" for help.

postgres=#
```

Figure 4.9: Running the psql CLI in a container and connecting to a remote Postgres server

You can now use `\connect demo` to connect to the demo database and then use the `select * from users;` command to list all records that you added previously to this database:

```
postgres=# \connect demo;
You are now connected to database "demo" as user "postgres".
demo=# select * from users;
 id |     name
----+--------------
  1 | Ann Hunter
  2 | Suzanne Baker
  3 | Will Smith
(3 rows)

demo-#
```

Figure 4.10: Using psql to work with our remote DB

When you are done experimenting with the `psql` CLI, you can exit it and, at the same time, stop the container by pressing *CTRL-d*.

A simple node JS app to access PostgreSQL

Let us now create a very simple Node JS application that we will be running from within a container attached to the `my-network` network and accessing the PostgreSQL database server.

Proceed as follows:

1. In your `src` folder, create a new folder called `ch04/node-client` and navigate to it:

   ```
   mkdir ch04/node-client && cd ch04/node-client
   ```

2. Open VS Code from within this folder:

   ```
   code .
   ```

3. To this folder, add a file **server.js** and add the following code:



Figure 4.11: Node JS code to retrieve users from PostgreSQL.

4. Add a file called **packages.json** with this content:



Figure 4.12: Package.json for the Node JS client

2. Open VS Code from within this folder:

   ```
   code .
   ```

3. To this folder, add a file **server.js** and add the following code:



Figure 4.11: Node JS code to retrieve users from PostgreSQL.

4. Add a file called **packages.json** with this content:



Figure 4.12: Package.json for the Node JS client

5. Add a file called **Dockerfile** and add this content:

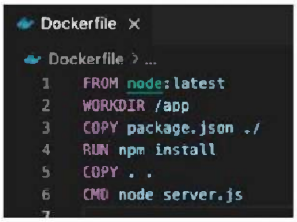


Figure 4.13: Dockerfile for the Node JS client

6. In the terminal, from within the folder **node-client**, run:

```
docker image build -t node-client .
```

Note the point (**.**) at the end of the preceding line!

The output of the preceding command might look similar to this:

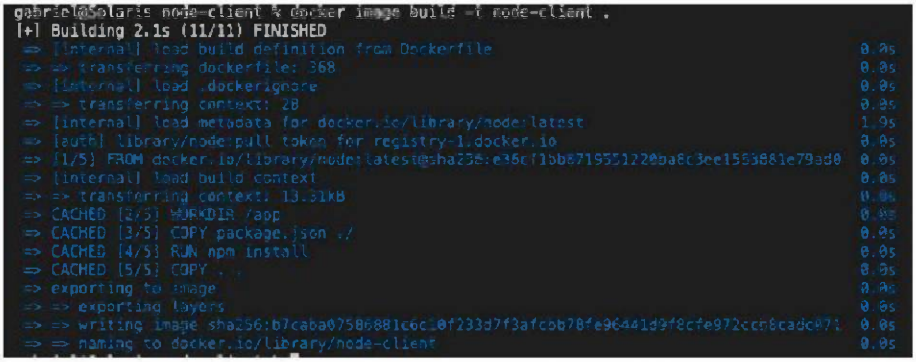


Figure 4.14: Building Docker image for the Node JS client

Note, the first time you run the command, it may take a while since Docker needs to download the **node:latest** base image and then all node modules during the build of the image. In subsequent builds, Docker is intelligent enough to use cached instances of the base image and the build layers.

1. Once the image is successfully built, run the Node JS client with this command:

```
docker container run \
    --network my-network \
    --rm \
    node-client
```

5. Add a file called **Dockerfile** and add this content:

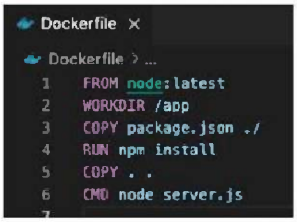


Figure 4.13: Dockerfile for the Node JS client

6. In the terminal, from within the folder **node-client**, run:

```
docker image build -t node-client .
```

Note the point (**.**) at the end of the preceding line!

The output of the preceding command might look similar to this:

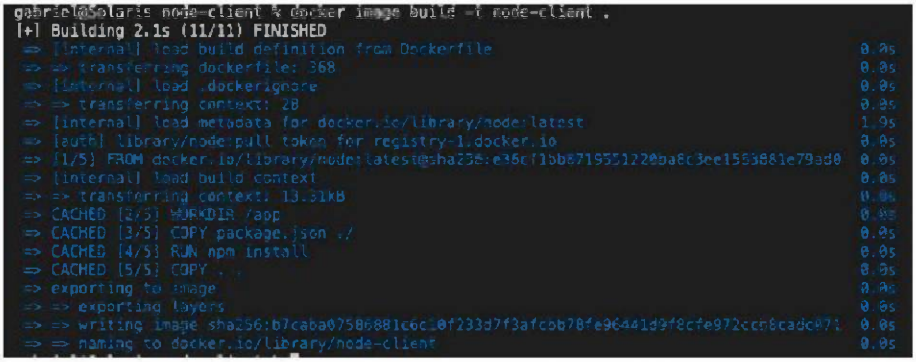


Figure 4.14: Building Docker image for the Node JS client

Note, the first time you run the command, it may take a while since Docker needs to download the **node:latest** base image and then all node modules during the build of the image. In subsequent builds, Docker is intelligent enough to use cached instances of the base image and the build layers.

1. Once the image is successfully built, run the Node JS client with this command:

```
docker container run \
    --network my-network \
    --rm \
    node-client
```

The output in your terminal should look similar to this:

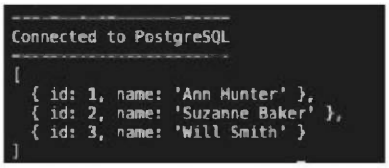


Figure 4.15: Node JS client accessing our PostgreSQL database server

This was a very quick and easy introduction on how to write a Node JS client for PostgreSQL that runs in a container. Do not worry if there are certain details that are not fully transparent yet, as everything will be discussed in detail in subsequent chapters of this book. For the moment, it is important that you get a working client for our PostgreSQL database server, which runs locally on your computer.

Before we continue, let us stop and remove the PostgreSQL database server from our system with the following:

```
docker container rm --force my-postgres
```

Now that we have learned how to run and use a SQL database running inside a Docker container on our local computer and how to access it from other containers on the same Docker network, we want to explore the same for a popular no-SQL database. We select MongoDB to demonstrate this.

Running a no-SQL database locally

MongoDB is a very popular, performant, and highly scalable document database. We can find a curated Docker image of this database on Docker Hub at the following link:

https://hub.docker.com/_/mongo

Let us run an instance of Mongo called `my-mongo` attached to the network `my-network` running in detach mode with the following command:

```
docker container run \
    --name my-mongo \
    --network my-network \
    --detach \
    mongo:latest
```

Since this is the first time you run MongoDB, Docker has to download the image first, which can take a moment. When the container is up and running, you should see something like this in your terminal:

The output in your terminal should look similar to this:

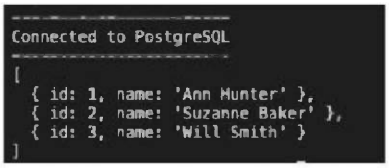


Figure 4.15: Node JS client accessing our PostgreSQL database server

This was a very quick and easy introduction on how to write a Node JS client for PostgreSQL that runs in a container. Do not worry if there are certain details that are not fully transparent yet, as everything will be discussed in detail in subsequent chapters of this book. For the moment, it is important that you get a working client for our PostgreSQL database server, which runs locally on your computer.

Before we continue, let us stop and remove the PostgreSQL database server from our system with the following:

```
docker container rm --force my-postgres
```

Now that we have learned how to run and use a SQL database running inside a Docker container on our local computer and how to access it from other containers on the same Docker network, we want to explore the same for a popular no-SQL database. We select MongoDB to demonstrate this.

Running a no-SQL database locally

MongoDB is a very popular, performant, and highly scalable document database. We can find a curated Docker image of this database on Docker Hub at the following link:

https://hub.docker.com/_/mongo

Let us run an instance of Mongo called `my-mongo` attached to the network `my-network` running in detach mode with the following command:

```
docker container run \
    --name my-mongo \
    --network my-network \
    --detach \
    mongo:latest
```

Since this is the first time you run MongoDB, Docker has to download the image first, which can take a moment. When the container is up and running, you should see something like this in your terminal:

```
gabriel@solaris ~ % docker container run \
    --name my-mongo \
    --network my-network \
    --detach \
    mongo:latest
Unable to find image 'mongo:latest' locally
latest: Pulling from library/mongo
55c604a74c4b: Pull complete
d9ae63d6c154: Pull complete
42ff824326e4: Pull complete
56b6fdecd48a: Pull complete
c801f51a799a: Pull complete
c67ddedacab6: Pull complete
278c286d75c5: Pull complete
fbbf23426b69: Pull complete
9c5a46949e24: Pull complete
fe6d41500023: Pull complete
Digest: sha256:be0b7f6b4e83c39d598696d18e5264368bb2cd8230dcb1e2546d025edSa3dab9
Status: Downloaded newer image for mongo:latest
5814c94dbb8697e497688c49a25b06a9bc0a812cf9f3a2eaf649383ce03673cc
```

Figure 4.16: Running a MongoDB server instance

Now, let us try to use the MongoDB shell called **mongo** to create a database and a collection and fill it with some data. We try to first execute the CLI directly inside our **my-mongo** container, as we did with the PostgreSQL database. Use the following command to execute the CLI:

```
docker container exec -it my-mongo mongo
```

You should see this in your terminal:

```
gabriel@solaris ~ % docker container exec -it my-mongo mongo
MongoDB shell version v4.4.6
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("c41001f7-595a-4d1d-bd4d-ac5588a528f7") }
MongoDB server version: 4.4.6
---
The server generated these startup warnings when booting:
        2021-06-20T15:49:56.168+00:00: Using the XFS filesystem is strongly recommended with the WiredTiger stor
hub.mongodb.org/core/prodnotes-filesystem
        2021-06-20T15:49:56.192+00:00: Access control is not enabled for the database. Read and write access to
unrestricted
---
---
        Enable MongoDB's free cloud-based monitoring service, which will then receive and display
        metrics about your deployment (disk utilization, CPU, operation statistics, etc).

        The monitoring data will be available on a MongoDB website with a unique URL accessible to you
        and anyone you share the URL with. MongoDB may use this information to make product
        improvements and to suggest MongoDB products and deployment options to you.

        To enable free monitoring, run the following command: db.enableFreeMonitoring()
        To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
```

Figure 4.17: Using the MongoDB shell

As you can see, we are greeted by the shell with a welcome text and are now ready to execute some commands against our MongoDB instance.

```
gabriel@solaris ~ % docker container run \
    --name my-mongo \
    --network my-network \
    --detach \
    mongo:latest
Unable to find image 'mongo:latest' locally
latest: Pulling from library/mongo
55c604a74c4b: Pull complete
d9ae63d6c154: Pull complete
42ff824326e4: Pull complete
56b6fdecd48a: Pull complete
c801f51a799a: Pull complete
c67ddedacab6: Pull complete
278c286d75c5: Pull complete
fbbf23426b69: Pull complete
9c5a46949e24: Pull complete
fe6d41500023: Pull complete
Digest: sha256:be0b7f6b4e83c39d598696d18e5264368bb2cd8230dcb1e2546d025edSa3dab9
Status: Downloaded newer image for mongo:latest
5814c94dbb8697e497688c49a25b06a9bc0a812cf9f3a2eaf649383ce03673cc
```

Figure 4.16: Running a MongoDB server instance

Now, let us try to use the MongoDB shell called **mongo** to create a database and a collection and fill it with some data. We try to first execute the CLI directly inside our **my-mongo** container, as we did with the PostgreSQL database. Use the following command to execute the CLI:

```
docker container exec -it my-mongo mongo
```

You should see this in your terminal:

```
gabriel@solaris ~ % docker container exec -it my-mongo mongo
MongoDB shell version v4.4.6
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("c41001f7-595a-4d1d-bd4d-ac5588a528f7") }
MongoDB server version: 4.4.6
---
The server generated these startup warnings when booting:
        2021-06-20T15:49:56.168+00:00: Using the XFS filesystem is strongly recommended with the WiredTiger stor
hub.mongodb.org/core/prodnotes-filesystem
        2021-06-20T15:49:56.192+00:00: Access control is not enabled for the database. Read and write access to
unrestricted
---
---
        Enable MongoDB's free cloud-based monitoring service, which will then receive and display
        metrics about your deployment (disk utilization, CPU, operation statistics, etc).

        The monitoring data will be available on a MongoDB website with a unique URL accessible to you
        and anyone you share the URL with. MongoDB may use this information to make product
        improvements and to suggest MongoDB products and deployment options to you.

        To enable free monitoring, run the following command: db.enableFreeMonitoring()
        To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
```

Figure 4.17: Using the MongoDB shell

As you can see, we are greeted by the shell with a welcome text and are now ready to execute some commands against our MongoDB instance.

Use the following sequence of commands to create a database **demo**, connect to it and then create a collection **users** and fill it with some data:

1. Use **show dbs** to display the list of databases currently defined on your MongoDB server
2. Use the command **use demo** to implicitly create a database **demo** and connect to it
3. Use the command **db.users.insert({id:1, name: "Ann Hunter"})** to insert a document into the collection **users** in the DB **demo**
4. Add a few additional records to the **users** collection
5. Use **show collections** to get a list of all collections in the active database
6. Use **db.users.find()** (or more generically **db.<collection-name>.find()**, where **<collection-name>** is the name of the collection whose content you want to refer to) to output the list of documents stored in this collection.

You should see something along the line of:

```
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
> use demo
switched to db demo
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
> db.users.insert({id:1, name: "Ann Hunter"})
WriteResult({ "nInserted" : 1 })
> db.users.insert({id:2, name: "Suzanne Wellington"})
WriteResult({ "nInserted" : 1 })
> db.users.insert({id:3, name: "John Doe"})
WriteResult({ "nInserted" : 1 })
> show collections
users
> db.users.find()
{ "_id" : ObjectId("60cf67d6ebbf4d63c8199120"), "id" : 1, "name" : "Ann Hunter" }
{ "_id" : ObjectId("60cf67eaebbf4d63c8199121"), "id" : 2, "name" : "Suzanne Wellington" }
{ "_id" : ObjectId("60cf67fcebbf4d63c8199122"), "id" : 3, "name" : "John Doe" }
```

Figure 4.18: Working with the Mongo shell

When done playing around with MongoDB, use *CTRL-d* to exit the mongo shell and return to the console.

Next, we are going to show how you can access the MongoDB server from within a different container running on the same network.

Use the following sequence of commands to create a database **demo**, connect to it and then create a collection **users** and fill it with some data:

1. Use **show dbs** to display the list of databases currently defined on your MongoDB server
2. Use the command **use demo** to implicitly create a database **demo** and connect to it
3. Use the command **db.users.insert({id:1, name: "Ann Hunter"})** to insert a document into the collection **users** in the DB **demo**
4. Add a few additional records to the **users** collection
5. Use **show collections** to get a list of all collections in the active database
6. Use **db.users.find()** (or more generically **db.<collection-name>.find()**, where **<collection-name>** is the name of the collection whose content you want to refer to) to output the list of documents stored in this collection.

You should see something along the line of:

```
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
> use demo
switched to db demo
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
> db.users.insert({id:1, name: "Ann Hunter"})
WriteResult({ "nInserted" : 1 })
> db.users.insert({id:2, name: "Suzanne Wellington"})
WriteResult({ "nInserted" : 1 })
> db.users.insert({id:3, name: "John Doe"})
WriteResult({ "nInserted" : 1 })
> show collections
users
> db.users.find()
{ "_id" : ObjectId("60cf67d6ebbf4d63c8199120"), "id" : 1, "name" : "Ann Hunter" }
{ "_id" : ObjectId("60cf67eaebbf4d63c8199121"), "id" : 2, "name" : "Suzanne Wellington" }
{ "_id" : ObjectId("60cf67fcebbf4d63c8199122"), "id" : 3, "name" : "John Doe" }
```

Figure 4.18: Working with the Mongo shell

When done playing around with MongoDB, use *CTRL-d* to exit the mongo shell and return to the console.

Next, we are going to show how you can access the MongoDB server from within a different container running on the same network.

Accessing MongoDB from a different container

To access the MongoDB server from a different container, we can use another instance of the `mongo:latest` image and run it attached to the same network as the `my-mongo` container. Then, we use the CLI from within that container to access the now remote DB. Use the following command to start the CLI:

```
docker run -it \
    --network my-network \
    --rm \
    mongo:latest mongo --host my-mongo
```

Once we are inside the shell, we can use the same commands as in the previous section to show the list of databases and collections as well as the content of a given collection. We can, of course, also add data to a collection or modify and delete existing data:

```
> show dbs
admin   0.000GB
config  0.000GB
demo    0.000GB
local   0.000GB
> use demo
switched to db demo
> show collections
users
> db.users.find()
{ "_id" : ObjectId("60cf67d6ebbf4d63c8199120"), "id" : 1, "name" : "Ann Hunter" }
{ "_id" : ObjectId("60cf67eaebbf4d63c8199121"), "id" : 2, "name" : "Suzanne Wellington" }
{ "_id" : ObjectId("60cf67fcebbf4d63c8199122"), "id" : 3, "name" : "John Doe" }
>
```

Figure 4.19: *Using the mongo shell from within another container to access the MongoDB server*

As you can see in the preceding output, we have been able to access the data and play with it the same way as when we were using the mongo shell from within the database server container.

Use *CTRL-d* to exit the shell and return to your terminal session.

A simple Python app to access Mongo DB

Let us now create a very simple Python application that we will be running from within a container attached to the `my-network` network and accessing the MongoDB server.

Proceed as follows:

1. Open a terminal window and navigate to your `src` folder
2. From within the `src` folder, create a new folder called `ch04/python-client` and navigate to it:

Accessing MongoDB from a different container

To access the MongoDB server from a different container, we can use another instance of the `mongo:latest` image and run it attached to the same network as the `my-mongo` container. Then, we use the CLI from within that container to access the now remote DB. Use the following command to start the CLI:

```
docker run -it \
    --network my-network \
    --rm \
    mongo:latest mongo --host my-mongo
```

Once we are inside the shell, we can use the same commands as in the previous section to show the list of databases and collections as well as the content of a given collection. We can, of course, also add data to a collection or modify and delete existing data:

```
> show dbs
admin   0.000GB
config  0.000GB
demo    0.000GB
local   0.000GB
> use demo
switched to db demo
> show collections
users
> db.users.find()
{ "_id" : ObjectId("60cf67d6ebbf4d63c8199120"), "id" : 1, "name" : "Ann Hunter" }
{ "_id" : ObjectId("60cf67eaebbf4d63c8199121"), "id" : 2, "name" : "Suzanne Wellington" }
{ "_id" : ObjectId("60cf67fcebbf4d63c8199122"), "id" : 3, "name" : "John Doe" }
>
```

Figure 4.19: *Using the mongo shell from within another container to access the MongoDB server*

As you can see in the preceding output, we have been able to access the data and play with it the same way as when we were using the mongo shell from within the database server container.

Use *CTRL-d* to exit the shell and return to your terminal session.

A simple Python app to access Mongo DB

Let us now create a very simple Python application that we will be running from within a container attached to the `my-network` network and accessing the MongoDB server.

Proceed as follows:

1. Open a terminal window and navigate to your `src` folder
2. From within the `src` folder, create a new folder called `ch04/python-client` and navigate to it:

```
mkdir ch04/python-client && cd ch04/python-client
```

3. Open VS Code from within this folder:

```
code .
```

4. To this folder, add a file `server.py` and add the following code:



Figure 4.20: Python code for the MongoDB client

5. Add a file called `requirements.txt` with this content:

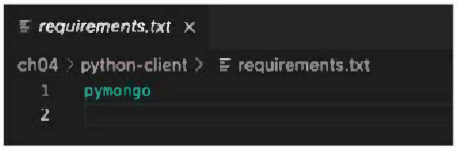


Figure 4.21: Dependencies for the Python client

Yes, that is correct; there is only one dependency to `pymongo` needed for this simple Python client.

6. Add a file called `Dockerfile` and add this content:

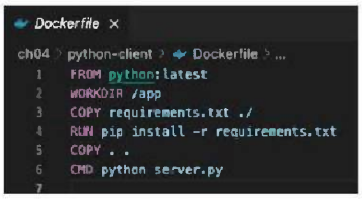


Figure 4.22: Dockerfile for the Python client

```
mkdir ch04/python-client && cd ch04/python-client
```

3. Open VS Code from within this folder:

```
code .
```

4. To this folder, add a file `server.py` and add the following code:



Figure 4.20: Python code for the MongoDB client

5. Add a file called `requirements.txt` with this content:

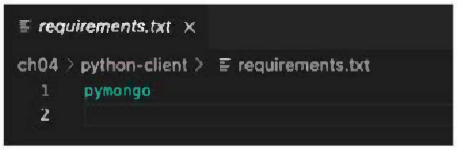


Figure 4.21: Dependencies for the Python client

Yes, that is correct; there is only one dependency to `pymongo` needed for this simple Python client.

6. Add a file called `Dockerfile` and add this content:

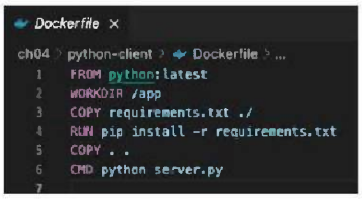


Figure 4.22: Dockerfile for the Python client

7. In the terminal, from within the folder Python-client, run:

   ```
   docker image build -t python-client .
   ```

   Note the point (.) at the end of the previous line!

8. Run the Python client with the following:

   ```
   docker container run --network my-network --rm python-client
   ```

   The output in your terminal should look similar to this:

```
gabriel@Solaris python-client % docker run --network my-network --rm python-client

Connected to MongoDB

Found 3 users.
{'_id': ObjectId('60cf67d6ebbf4d63c8199120'), 'id': 1.0, 'name': 'Ann Hunter'}
{'_id': ObjectId('60cf67eaebbf4d63c8199121'), 'id': 2.0, 'name': 'Suzanne Wellington'}
{'_id': ObjectId('60cf67fcebbf4d63c8199122'), 'id': 3.0, 'name': 'John Doe'}
```

Figure 4.23: Python client accessing our MongoDB server

Once again, let me reassure you if you do not yet understand the meaning of every single step. I promise we will address every detail in subsequent chapters of this book.

To end, clean up your Docker environment and remove the MongoDB container as well as the network. Use the following commands:

```
docker container rm -f my-mongo
docker network rm my-network
```

We have shown you how you can run any database—SQL or non-SQL—on your local computer with ease, as long as it is containerized. The good thing is that you can find curated Docker images of the most popular databases on Docker Hub. Here we want to name just a few:

- **SQL:** PostgreSQL, MySQL, MariaDB, MS SQL Server, and Oracle
- **No-SQL:** MongoDB, CouchDB, and RavenDB

With this, we want to turn our attention a bit more onto the structure and inner workings of container images.

Investigating the image layers model

You have been playing with quite a few Docker images so far without caring much about what they exactly are and how their inner structure looks like. That is OK, but now it is time that we put the focus on this and dive a bit deeper.

7. In the terminal, from within the folder Python-client, run:

   ```
   docker image build -t python-client .
   ```

   Note the point (.) at the end of the previous line!

8. Run the Python client with the following:

   ```
   docker container run --network my-network --rm python-client
   ```

   The output in your terminal should look similar to this:

```
gabriel@Solaris python-client % docker run --network my-network --rm python-client

Connected to MongoDB

Found 3 users.
{'_id': ObjectId('60cf67d6ebbf4d63c8199120'), 'id': 1.0, 'name': 'Ann Hunter'}
{'_id': ObjectId('60cf67eaebbf4d63c8199121'), 'id': 2.0, 'name': 'Suzanne Wellington'}
{'_id': ObjectId('60cf67fcebbf4d63c8199122'), 'id': 3.0, 'name': 'John Doe'}
```

Figure 4.23: Python client accessing our MongoDB server

Once again, let me reassure you if you do not yet understand the meaning of every single step. I promise we will address every detail in subsequent chapters of this book.

To end, clean up your Docker environment and remove the MongoDB container as well as the network. Use the following commands:

```
docker container rm -f my-mongo
docker network rm my-network
```

We have shown you how you can run any database—SQL or non-SQL—on your local computer with ease, as long as it is containerized. The good thing is that you can find curated Docker images of the most popular databases on Docker Hub. Here we want to name just a few:

- **SQL:** PostgreSQL, MySQL, MariaDB, MS SQL Server, and Oracle
- **No-SQL:** MongoDB, CouchDB, and RavenDB

With this, we want to turn our attention a bit more onto the structure and inner workings of container images.

Investigating the image layers model

You have been playing with quite a few Docker images so far without caring much about what they exactly are and how their inner structure looks like. That is OK, but now it is time that we put the focus on this and dive a bit deeper.

What is a Docker image?

But, before we dive deeper into the internals of images, we need to ask ourselves, what is a Docker image in the first place? For this, we need to look closer at Linux as containers have originally been evolved in the Linux ecosystem. In Linux, the operating system is made up of a filesystem where the files and directories are all stored on the local disk. Everything in Linux is a file. Having said that, a Docker image is a tarball containing such a filesystem, whereas a tarball is nothing else than a compressed set of files and folders popular on Linux. But as we will see shortly, there is a bit more than just that.

The layered filesystem

A Docker image is made up of a stack of immutable layers. You can think of it as a deck of cards. The simplest image can consist of a single layer, but usually, you will find that a given image consists of several layers. We can see that in the following image:

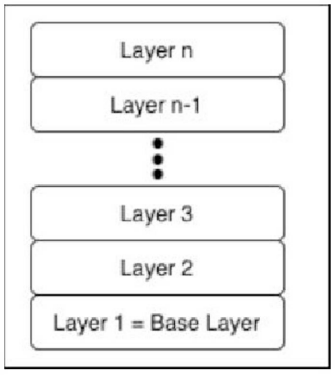


Figure 4.24: Layers of a Docker image

As we can see, the stack of layers starts with 1, which is also called the base layer. On top of Layer 1, we have Layer 2, and so on, until we get to the last layer; here it is Layer *n*. This is the topmost layer. Each of those layers contains a bunch of files and folders. Together they make up that filesystem we mentioned earlier.

Each layer only contains the difference or changes to the filesystem formed by the lower laying layers. This is also called a Union filesystem. The details of such a filesystem are not in the scope of this book, though, and are documented elsewhere pretty well.

A storage driver handles the way these layers interact with each other. Docker provides a set of different such drivers, and each one has its own specific advantages

What is a Docker image?

But, before we dive deeper into the internals of images, we need to ask ourselves, what is a Docker image in the first place? For this, we need to look closer at Linux as containers have originally been evolved in the Linux ecosystem. In Linux, the operating system is made up of a filesystem where the files and directories are all stored on the local disk. Everything in Linux is a file. Having said that, a Docker image is a tarball containing such a filesystem, whereas a tarball is nothing else than a compressed set of files and folders popular on Linux. But as we will see shortly, there is a bit more than just that.

The layered filesystem

A Docker image is made up of a stack of immutable layers. You can think of it as a deck of cards. The simplest image can consist of a single layer, but usually, you will find that a given image consists of several layers. We can see that in the following image:

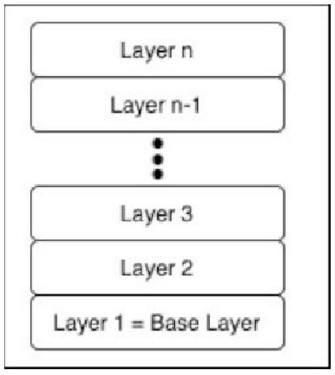


Figure 4.24: Layers of a Docker image

As we can see, the stack of layers starts with 1, which is also called the base layer. On top of Layer 1, we have Layer 2, and so on, until we get to the last layer; here it is Layer *n*. This is the topmost layer. Each of those layers contains a bunch of files and folders. Together they make up that filesystem we mentioned earlier.

Each layer only contains the difference or changes to the filesystem formed by the lower laying layers. This is also called a Union filesystem. The details of such a filesystem are not in the scope of this book, though, and are documented elsewhere pretty well.

A storage driver handles the way these layers interact with each other. Docker provides a set of different such drivers, and each one has its own specific advantages

or disadvantages. Normally, though, it is not your job to deal with such nuances, as the default driver is the one who will suit you the best in most cases other than very specific or exotic ones.

It is very important to note that each layer is immutable. Once created, a layer cannot be modified anymore. The only thing one can do is delete an existing layer. Since layers are immutable, it automatically follows that a Docker image as a whole is also immutable. This immutability of images and layers has a lot of advantages, the most important one of them being that a Docker image can be cached and will never become stale.

In the following graphic, we try to illustrate how a custom Docker image could look like that contains a simple Web application using Nginx as a Web server:

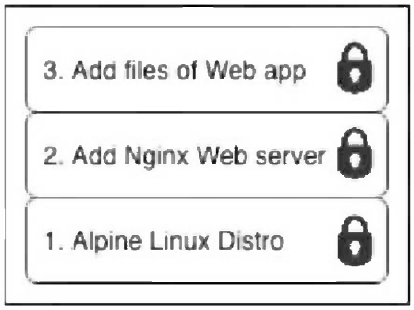


***Figure 4.25:** A sample custom Docker image containing a simple Web application*

The image uses a base layer containing all the files and folders needed for it to be an Alpine Linux distro. Then on top of that, we have Layer 2, which contains all files and folders that an Nginx Web server needs to run on top of Alpine. Finally, the third and last layer contains all the files and folders that make up your specific Web applications. These are the HTML, CSS, and JavaScript files, plus all the other resources, such as images, fonts, and more.

We have added a lock symbol to each of the layers to indicate that once built, the individual layers are immutable.

If you are more curious and want to investigate where Docker stores those files on the system, then the content of each layer is stored in a dedicated folder on the host system. Usually, you can find those sub-folders at `/var/lib/docker`.

Once again, I want to point out a nice side effect of the fact that layers are immutable. They can be cached without ever becoming stale. This is a tremendous advantage, as we will see throughout this book.

or disadvantages. Normally, though, it is not your job to deal with such nuances, as the default driver is the one who will suit you the best in most cases other than very specific or exotic ones.

It is very important to note that each layer is immutable. Once created, a layer cannot be modified anymore. The only thing one can do is delete an existing layer. Since layers are immutable, it automatically follows that a Docker image as a whole is also immutable. This immutability of images and layers has a lot of advantages, the most important one of them being that a Docker image can be cached and will never become stale.

In the following graphic, we try to illustrate how a custom Docker image could look like that contains a simple Web application using Nginx as a Web server:

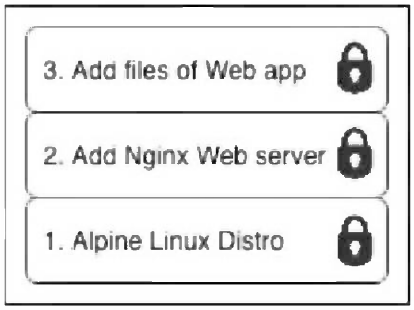


***Figure 4.25:** A sample custom Docker image containing a simple Web application*

The image uses a base layer containing all the files and folders needed for it to be an Alpine Linux distro. Then on top of that, we have Layer 2, which contains all files and folders that an Nginx Web server needs to run on top of Alpine. Finally, the third and last layer contains all the files and folders that make up your specific Web applications. These are the HTML, CSS, and JavaScript files, plus all the other resources, such as images, fonts, and more.

We have added a lock symbol to each of the layers to indicate that once built, the individual layers are immutable.

If you are more curious and want to investigate where Docker stores those files on the system, then the content of each layer is stored in a dedicated folder on the host system. Usually, you can find those sub-folders at `/var/lib/docker`.

Once again, I want to point out a nice side effect of the fact that layers are immutable. They can be cached without ever becoming stale. This is a tremendous advantage, as we will see throughout this book.

The mutable container layer

So far, we have seen that all the layers making up a container image are immutable. We also kind of know that a container image is a template from which a container is built at runtime. When we use a **`docker run`** command, we instruct Docker to load a copy of the image's layers into memory, where they will build an encapsulated filesystem containing a runnable version of the containerized application. But applications, when running normally need to mutate some files and folders in the context where they are running. How can they do that if, as we have stated, everything is immutable? The answer is quite easy, as it turns out. The Docker engine, when creating a container from an image, adds a writeable container layer on top of the stack of immutable layers originating from the image. Using the preceding custom image with the Nginx-based Web application, this would look as shown in the next graphic:



Figure 4.26: A container based on the customer image with a writeable container layer

We have marked the container layer as read/write (r/w) in the preceding image. The application, when running, is now free to make changes to the filesystem of the container. All those modifications will then be captured in that container layer.

Please be aware, though, that since this container layer is created at runtime, it is destroyed when a container is stopped and subsequently removed from the system. That is, all changes that the applications made to the filesystem are lost. We will show ways on how to avoid this in the next section when we talk about Docker volumes.

Due to the fact that the image layers are immutable, we can run several containers from a single image, and they will share the image layers. This is a huge optimization because it significantly reduces the number of resources that are used and, as such, contributes to the fact that containers are so versatile. Let us use another graphic to depict how this looks like:

The mutable container layer

So far, we have seen that all the layers making up a container image are immutable. We also kind of know that a container image is a template from which a container is built at runtime. When we use a **`docker run`** command, we instruct Docker to load a copy of the image's layers into memory, where they will build an encapsulated filesystem containing a runnable version of the containerized application. But applications, when running normally need to mutate some files and folders in the context where they are running. How can they do that if, as we have stated, everything is immutable? The answer is quite easy, as it turns out. The Docker engine, when creating a container from an image, adds a writeable container layer on top of the stack of immutable layers originating from the image. Using the preceding custom image with the Nginx-based Web application, this would look as shown in the next graphic:



Figure 4.26: A container based on the customer image with a writeable container layer

We have marked the container layer as read/write (r/w) in the preceding image. The application, when running, is now free to make changes to the filesystem of the container. All those modifications will then be captured in that container layer.

Please be aware, though, that since this container layer is created at runtime, it is destroyed when a container is stopped and subsequently removed from the system. That is, all changes that the applications made to the filesystem are lost. We will show ways on how to avoid this in the next section when we talk about Docker volumes.

Due to the fact that the image layers are immutable, we can run several containers from a single image, and they will share the image layers. This is a huge optimization because it significantly reduces the number of resources that are used and, as such, contributes to the fact that containers are so versatile. Let us use another graphic to depict how this looks like:

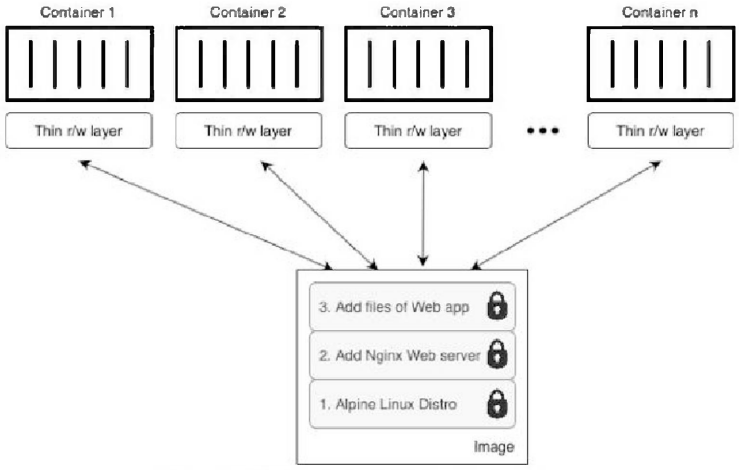


Figure 4.27: Containers sharing layers of an image

As you can see, we have *n* containers running on the system. They are all instantiated from the same Docker image. Thus, in this particular case, in memory, they share the same three image layers, but each container instance has its own thin read/write container layer. Yes, that is correct; each container has its own writeable layer. Thus, individual container instances cannot mess with each other's filesystem. This is important to understand!

Another benefit of this approach is that we will experience a significant decrease in the upfront loading time for any subsequent container created once the image layers have been loaded into memory, which happens when we run the first container.

Docker copy-on-write mechanism

Let us now try to understand what mechanism Docker uses when building up layers of an image. It is called the copy-on-write mechanism. This is a strategy commonly used for sharing files with maximum efficiency. Now, try to put yourself inside an image layer. If, within this layer, you need a file or a folder, and it is present in one of the lower-laying layers, then you just take that one. That is, you reach "down" in the stack and pick what you need.

If, on the other hand, the file or folder is not there yet in the stack below you, you will add that file or folder to the layer you are currently sitting in. Basically, you are performing an add operation.

But what if you want to modify an existing file? In that case, Docker creates a copy of the existing file from the lower layer(s) and applies the modifications, and adds it to the current layer. This is also called copy-on-write.

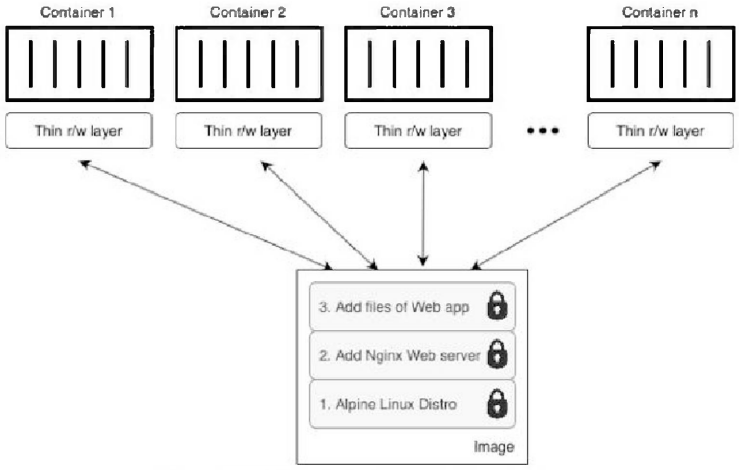


Figure 4.27: Containers sharing layers of an image

As you can see, we have *n* containers running on the system. They are all instantiated from the same Docker image. Thus, in this particular case, in memory, they share the same three image layers, but each container instance has its own thin read/write container layer. Yes, that is correct; each container has its own writeable layer. Thus, individual container instances cannot mess with each other's filesystem. This is important to understand!

Another benefit of this approach is that we will experience a significant decrease in the upfront loading time for any subsequent container created once the image layers have been loaded into memory, which happens when we run the first container.

Docker copy-on-write mechanism

Let us now try to understand what mechanism Docker uses when building up layers of an image. It is called the copy-on-write mechanism. This is a strategy commonly used for sharing files with maximum efficiency. Now, try to put yourself inside an image layer. If, within this layer, you need a file or a folder, and it is present in one of the lower-laying layers, then you just take that one. That is, you reach "down" in the stack and pick what you need.

If, on the other hand, the file or folder is not there yet in the stack below you, you will add that file or folder to the layer you are currently sitting in. Basically, you are performing an add operation.

But what if you want to modify an existing file? In that case, Docker creates a copy of the existing file from the lower layer(s) and applies the modifications, and adds it to the current layer. This is also called copy-on-write.

As always, to make it easier for us to understand, let us use a graphic:

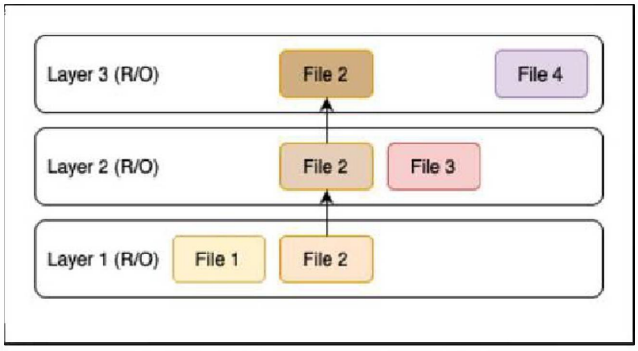


Figure 4.28: The copy-on-write mechanism

Our image starts with layer 1, where we have added two files, **File 1** and **File 2**. Then layer 2 adds a new file called **File 3**. It also contains a modified version of **File 2**. The up arrow indicates the copy-on-write operation that the Docker engine will perform. First, it copies **File 2** from the lower laying layer and then applies the changes.

The last Layer (3) in our example makes yet another change to **File 2**. Thus, the Docker engine copies the file from Layer 2, apply those changes, and then writes its own copy into Layer 3. Finally, we add **File 4** to the last layer, and then we are done.

Now that you got some insight into the inner workings of Docker images, let us turn our attention to Docker volumes.

Using volumes with containers

Each container is like a sandbox for the application running inside it. It is not possible to access the application from outside the container without special means, which you will get to know in *Chapter 6, Demystifying Container Networking*. It is also not possible to share data and files with an application running in a container. Or is it?

You will certainly agree with me that it would actually be a bad thing if we could, under no circumstances, share files with processes running inside containers. Often, we need to do exactly that. Just imagine you want to configure a containerized application at runtime with the help of a configuration file. Or imagine a situation where your containerized application needs to process some data that is only available at runtime in the form of files.

Luckily, Docker offers a mechanism that allows us to, figuratively speaking, poke a hole into the solid walls of the container, through which you can exchange files from outside with the containerized application. The technique Docker uses to share files

As always, to make it easier for us to understand, let us use a graphic:

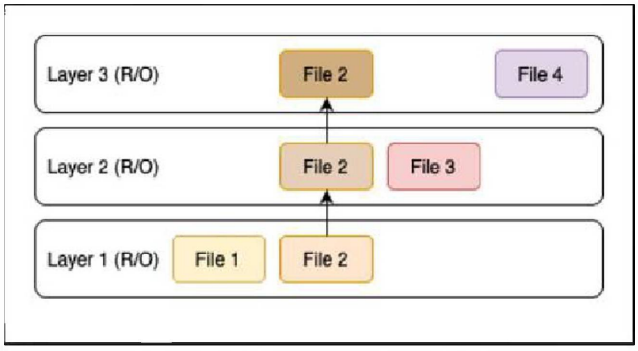


Figure 4.28: The copy-on-write mechanism

Our image starts with layer 1, where we have added two files, **File 1** and **File 2**. Then layer 2 adds a new file called **File 3**. It also contains a modified version of **File 2**. The up arrow indicates the copy-on-write operation that the Docker engine will perform. First, it copies **File 2** from the lower laying layer and then applies the changes.

The last Layer (3) in our example makes yet another change to **File 2**. Thus, the Docker engine copies the file from Layer 2, apply those changes, and then writes its own copy into Layer 3. Finally, we add **File 4** to the last layer, and then we are done.

Now that you got some insight into the inner workings of Docker images, let us turn our attention to Docker volumes.

Using volumes with containers

Each container is like a sandbox for the application running inside it. It is not possible to access the application from outside the container without special means, which you will get to know in *Chapter 6, Demystifying Container Networking*. It is also not possible to share data and files with an application running in a container. Or is it?

You will certainly agree with me that it would actually be a bad thing if we could, under no circumstances, share files with processes running inside containers. Often, we need to do exactly that. Just imagine you want to configure a containerized application at runtime with the help of a configuration file. Or imagine a situation where your containerized application needs to process some data that is only available at runtime in the form of files.

Luckily, Docker offers a mechanism that allows us to, figuratively speaking, poke a hole into the solid walls of the container, through which you can exchange files from outside with the containerized application. The technique Docker uses to share files

between a container and the outside world or between containers is by using so-called Docker volumes. Let us now investigate how these volumes work, shall we?

Creating, inspecting, and deleting Docker volumes

In this section, we will learn how to explicitly or implicitly create volumes that we can use to share data between a container and the host system or between containers. Let us start by explicitly creating a volume. The command to do so is as follows:

```
docker volume create my-volume
```

This will create a Docker volume called `my-volume`. Try to create another one called `my-demo-volume`. Once you have done so, you can list all volumes with the following command:

```
docker volume ls
```

We should see something like the following:

```
gabriel@Solaris ~ % docker volume ls
DRIVER    VOLUME NAME
local     my-demo-volume
local     my-volume
```

Figure 4.29: List of Docker volumes on my host

Great, we can see the two volumes we just created. But what does that `local` in the column `DRIVER` mean? This just tells us that the data that is contained in the respective volume resides on this computer, the so-called Docker host.

When using Docker, we can specify different types of volume drivers that will define where the data is going to be persisted to. By default, Docker uses the local driver, which, as we just said, stores all data locally. But there exists a plethora of different drivers that help you to put your data in a more safe location. As an example, think of EBS volumes or S3 blob storage on AWS. Drivers for the other cloud providers, MS Azure or Google cloud, exist as well.

If you are curious about where exactly the data sits when using the local volume driver, then I have to tell you that this is a bit tricky if you are on a Mac or Windows computer. It is simple if you are using a Linux computer, though. Then, by default, Docker puts the data in a sub-directory of `/var/lib/docker/volumes`. Go and have a look into this folder if you happen to be on a Linux machine. You should find two sub-folders named `my-volume` and `my-demo-volume`.

In the case of Mac and Windows, let me just say that Docker runs a little Linux VM under the hood of your system. This is fully transparent to you, and you will not

between a container and the outside world or between containers is by using so-called Docker volumes. Let us now investigate how these volumes work, shall we?

Creating, inspecting, and deleting Docker volumes

In this section, we will learn how to explicitly or implicitly create volumes that we can use to share data between a container and the host system or between containers. Let us start by explicitly creating a volume. The command to do so is as follows:

```
docker volume create my-volume
```

This will create a Docker volume called `my-volume`. Try to create another one called `my-demo-volume`. Once you have done so, you can list all volumes with the following command:

```
docker volume ls
```

We should see something like the following:

```
gabriel@Solaris ~ % docker volume ls
DRIVER    VOLUME NAME
local     my-demo-volume
local     my-volume
```

Figure 4.29: List of Docker volumes on my host

Great, we can see the two volumes we just created. But what does that `local` in the column `DRIVER` mean? This just tells us that the data that is contained in the respective volume resides on this computer, the so-called Docker host.

When using Docker, we can specify different types of volume drivers that will define where the data is going to be persisted to. By default, Docker uses the local driver, which, as we just said, stores all data locally. But there exists a plethora of different drivers that help you to put your data in a more safe location. As an example, think of EBS volumes or S3 blob storage on AWS. Drivers for the other cloud providers, MS Azure or Google cloud, exist as well.

If you are curious about where exactly the data sits when using the local volume driver, then I have to tell you that this is a bit tricky if you are on a Mac or Windows computer. It is simple if you are using a Linux computer, though. Then, by default, Docker puts the data in a sub-directory of `/var/lib/docker/volumes`. Go and have a look into this folder if you happen to be on a Linux machine. You should find two sub-folders named `my-volume` and `my-demo-volume`.

In the case of Mac and Windows, let me just say that Docker runs a little Linux VM under the hood of your system. This is fully transparent to you, and you will not

even realize this when you operate Docker. But in this particular case, it makes it a bit trickier to locate the volumes folder, and I will not discuss the details here since this is out of the scope of this book.

Next, let us use the inspect command to get some more information about a particular Docker volume. Use the following command:

```
docker volume inspect my-volume
```

And you should see something similar to this:

```
gabriel@Solaris ~ % docker volume inspect my-volume
[
    {
        "CreatedAt": "2021-06-27T10:55:55Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/my-volume/_data",
        "Name": "my-volume",
        "Options": {},
        "Scope": "local"
    }
]
```

Figure 4.30: Inspecting a Docker volume

As you should notice, there is a lot of helpful information about the volume presented in this output. Most notably, which driver we are using (local) and where the data can be found on the host (Mountpoint).

If you need to delete a volume, you can do so by using the following command:

```
docker volume rm my-volume
```

The preceding command will remove the volume called `my-volume` from the system and, with it, all its content. But be aware that this is a destructive operation since the data that was stored in that volume is gone after this. Docker engine prevents you from mistakenly deleting a volume that is still in use by at least one container. The delete command will fail in this case. But even if the volume is not currently used by any container, it may still contain useful data. Thus, make sure you know what you are doing!

On a lower system, which is a non-production system, you may accumulate a lot of volumes over time that are obsolete because they are artifacts from earlier tests or simulations. In this case, it is good to know that there exists a handy prune command that allows you to remove all not used volumes in one go:

```
docker volume prune
```

Since this is a destructive operation, you will be asked by Docker if you are sure about this. Answer with "y" if you know what you are doing. Alternatively, you can use the `--force` (or `-f`) parameter to avoid being asked:

even realize this when you operate Docker. But in this particular case, it makes it a bit trickier to locate the volumes folder, and I will not discuss the details here since this is out of the scope of this book.

Next, let us use the inspect command to get some more information about a particular Docker volume. Use the following command:

```
docker volume inspect my-volume
```

And you should see something similar to this:

```
gabriel@Solaris ~ % docker volume inspect my-volume
[
    {
        "CreatedAt": "2021-06-27T10:55:55Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/my-volume/_data",
        "Name": "my-volume",
        "Options": {},
        "Scope": "local"
    }
]
```

Figure 4.30: Inspecting a Docker volume

As you should notice, there is a lot of helpful information about the volume presented in this output. Most notably, which driver we are using (local) and where the data can be found on the host (Mountpoint).

If you need to delete a volume, you can do so by using the following command:

```
docker volume rm my-volume
```

The preceding command will remove the volume called `my-volume` from the system and, with it, all its content. But be aware that this is a destructive operation since the data that was stored in that volume is gone after this. Docker engine prevents you from mistakenly deleting a volume that is still in use by at least one container. The delete command will fail in this case. But even if the volume is not currently used by any container, it may still contain useful data. Thus, make sure you know what you are doing!

On a lower system, which is a non-production system, you may accumulate a lot of volumes over time that are obsolete because they are artifacts from earlier tests or simulations. In this case, it is good to know that there exists a handy prune command that allows you to remove all not used volumes in one go:

```
docker volume prune
```

Since this is a destructive operation, you will be asked by Docker if you are sure about this. Answer with "y" if you know what you are doing. Alternatively, you can use the `--force` (or `-f`) parameter to avoid being asked:

```
docker volume prune --force
```

Mounting a volume into a container

So far, we have learned how to create, list, inspect, and delete a Docker volume. But now we certainly want to know how to use such a volume with a container.

Let us use a simple Alpine container to show how we can use the volume `my-volume` that we just created. Use the following command:

```
docker container run -it \
    --name demo \
    -v my-volume:/data \
    alpine /bin/sh
```

Let us decomposition this command a bit, shall we? We are basically running a container named `demo` from a Docker image called `alpine`. We are running this container interactively with an attached TTY (`-it`), and inside the container, we are running a Linux shell (`/bin/sh`). This should already be very familiar to you. The new piece that comes into play in the preceding command is the volume part:

```
-v my-volume:/data
```

This part has the following meaning: A volume called `my-volume` is mounted into the filesystem of the container at the location `/data`. Instead of the parameter `-v`, we could also have used the more explicit `--volume`.

We are free to use any valid folder inside the container where we want to mount the volume to. Here we chose to use the folder `/data`. If this folder does not yet exist inside the container, then the Docker engine will automatically create it for us.

When you execute the preceding command, you should see this in your terminal window:

```
gabriel@Solaris ~ % docker container run -it \
    --name demo \
    -v my-volume:/data \
    alpine /bin/sh
/ #
```

Figure 4.31: Mounting a volume to a container

As you can see, we are at the prompt of a Linux shell inside the Alpine container. Now, we can navigate inside this container to the folder `/data`, which was created by Docker because of our mount command `-v my-volume:/data`.

Inside this container, we can then create a simple text file called `sample.txt` with some content. In your terminal, enter:

```
docker volume prune --force
```

Mounting a volume into a container

So far, we have learned how to create, list, inspect, and delete a Docker volume. But now we certainly want to know how to use such a volume with a container.

Let us use a simple Alpine container to show how we can use the volume `my-volume` that we just created. Use the following command:

```
docker container run -it \
    --name demo \
    -v my-volume:/data \
    alpine /bin/sh
```

Let us decomposition this command a bit, shall we? We are basically running a container named `demo` from a Docker image called `alpine`. We are running this container interactively with an attached TTY (`-it`), and inside the container, we are running a Linux shell (`/bin/sh`). This should already be very familiar to you. The new piece that comes into play in the preceding command is the volume part:

```
-v my-volume:/data
```

This part has the following meaning: A volume called `my-volume` is mounted into the filesystem of the container at the location `/data`. Instead of the parameter `-v`, we could also have used the more explicit `--volume`.

We are free to use any valid folder inside the container where we want to mount the volume to. Here we chose to use the folder `/data`. If this folder does not yet exist inside the container, then the Docker engine will automatically create it for us.

When you execute the preceding command, you should see this in your terminal window:



Figure 4.31: Mounting a volume to a container

As you can see, we are at the prompt of a Linux shell inside the Alpine container. Now, we can navigate inside this container to the folder `/data`, which was created by Docker because of our mount command `-v my-volume:/data`.

Inside this container, we can then create a simple text file called `sample.txt` with some content. In your terminal, enter:

```
cd /data
echo "Hello world!" > sample.txt
```

When done, press *CTRL-d* to quit the shell inside the container and thus quit the container.

Next, remove the container from the system with the following:

```
docker container rm demo
```

If you try to list all your container with `docker container ls --all`, the demo container should be gone. But we still have the data in the volume `my-volume` that we just created from within the demo container. It is persisted on our machine, the Docker host.

We can easily prove that the data is still there by mounting the volume `my-volume` to another container and then inspecting it there. Use the following command to do so:

```
docker container run -it --rm \
    -v my-volume:/my-data \
    ubuntu cat /my-data/sample.txt
```

Notice that we are using a different image, namely Ubuntu and that we are mounting the volume to the folder `/my-data` this time. When executing the preceding command, you should see this or similar output in your terminal:

```
gabriel@Solaris ~ % docker container run -it --rm \
    -v my-volume:/my-data \
    ubuntu cat /my-data/sample.txt
Hello world!
```

Figure 4.32: Accessing pre-existing data from the mounted volume

This, of course, proves that there indeed still exists the data that we had been creating from within our Alpine container before.

Often, we want to mount a volume into a container and give the application(s) running inside the container only read access. This can be done by extending the mount instruction slightly. If we wanted to do this for the previous command, it would look like the following:

```
docker container run -it --rm \
    -v my-volume:/my-data:ro \
    ubuntu cat /my-data/sample.txt
```

Note the `:ro`—as for read-only—at the end of the second line.

It is a good practice to only have one of the applications writing data and one to many applications consuming it if you want to share data by, for example, using this technique of sharing data via a volume that is mounted into different containers.

```
cd /data
echo "Hello world!" > sample.txt
```

When done, press *CTRL-d* to quit the shell inside the container and thus quit the container.

Next, remove the container from the system with the following:

```
docker container rm demo
```

If you try to list all your container with `docker container ls --all`, the demo container should be gone. But we still have the data in the volume `my-volume` that we just created from within the demo container. It is persisted on our machine, the Docker host.

We can easily prove that the data is still there by mounting the volume `my-volume` to another container and then inspecting it there. Use the following command to do so:

```
docker container run -it --rm \
    -v my-volume:/my-data \
    ubuntu cat /my-data/sample.txt
```

Notice that we are using a different image, namely Ubuntu and that we are mounting the volume to the folder `/my-data` this time. When executing the preceding command, you should see this or similar output in your terminal:

```
gabriel@Solaris ~ % docker container run -it --rm \
    -v my-volume:/my-data \
    ubuntu cat /my-data/sample.txt
Hello world!
```

Figure 4.32: Accessing pre-existing data from the mounted volume

This, of course, proves that there indeed still exists the data that we had been creating from within our Alpine container before.

Often, we want to mount a volume into a container and give the application(s) running inside the container only read access. This can be done by extending the mount instruction slightly. If we wanted to do this for the previous command, it would look like the following:

```
docker container run -it --rm \
    -v my-volume:/my-data:ro \
    ubuntu cat /my-data/sample.txt
```

Note the `:ro`—as for read-only—at the end of the second line.

It is a good practice to only have one of the applications writing data and one to many applications consuming it if you want to share data by, for example, using this technique of sharing data via a volume that is mounted into different containers.

Let me quickly recap what is most important from what we have just learned:

We can create a Docker volume and mount it to any desired folder inside of a container. Once we have mounted that volume to this particular folder, any data that is added to that folder by the application running inside this container will be persisted in the volume. By default, the data will reside in a local folder on the host. But we can use different volume drivers, and as a consequence, the data will be persisted elsewhere, for example, in an S3 bucket on AWS. This is pretty amazing, right? The application inside the container does not have to know anything about the details of the storage chosen by you, it can just read from or write to the folder you have been mounting the volume to, and the persisting will "automagically" be taken care of from the view of the application running in the container.

Mounting a host folder into a container

It is good to know that, next to mounting volumes into a container, there is also a possibility to mount local folders into a container without the need to use or define Docker volumes.

This feature comes in very handy when developing containerized applications, as we will see in *Chapter 8, Testing and debugging containerized applications*.

Let us see, how this can be achieved. First, create a new sub-folder inside your folder `src/ch04` called `my-code`. Navigate to that folder and then run the following command:

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

Please notice the part that mounts the local folder into the container:

```
--volume $PWD:/app
```

In Linux (and in Unix in general, as well as in Bash on Windows), the `$PWD` will output the current working directory, that is, the full path to the directory you are launching the preceding `docker` command from. The reason why we do this is because the `docker volume mount` command requires us to use absolute paths, and we cannot use relative paths. `$PWD` does this for us without us having to hard code a very specific path that may not work on a different system.

That said, how does Docker know when to work with Docker volumes and when to work with local folders? Well, it is simple, Docker just analyses what is on the left side of the colon (`:`), and if it resolves to a filesystem path, Docker assumes that latter; otherwise, it assumes it is (the name of) a Docker volume.

Run the preceding command, and once inside the container, navigate to the `/app` folder. Inside that folder, create a simple file:

Let me quickly recap what is most important from what we have just learned:

We can create a Docker volume and mount it to any desired folder inside of a container. Once we have mounted that volume to this particular folder, any data that is added to that folder by the application running inside this container will be persisted in the volume. By default, the data will reside in a local folder on the host. But we can use different volume drivers, and as a consequence, the data will be persisted elsewhere, for example, in an S3 bucket on AWS. This is pretty amazing, right? The application inside the container does not have to know anything about the details of the storage chosen by you, it can just read from or write to the folder you have been mounting the volume to, and the persisting will "automagically" be taken care of from the view of the application running in the container.

Mounting a host folder into a container

It is good to know that, next to mounting volumes into a container, there is also a possibility to mount local folders into a container without the need to use or define Docker volumes.

This feature comes in very handy when developing containerized applications, as we will see in *Chapter 8, Testing and debugging containerized applications*.

Let us see, how this can be achieved. First, create a new sub-folder inside your folder `src/ch04` called `my-code`. Navigate to that folder and then run the following command:

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

Please notice the part that mounts the local folder into the container:

```
--volume $PWD:/app
```

In Linux (and in Unix in general, as well as in Bash on Windows), the `$PWD` will output the current working directory, that is, the full path to the directory you are launching the preceding `docker` command from. The reason why we do this is because the `docker volume mount` command requires us to use absolute paths, and we cannot use relative paths. `$PWD` does this for us without us having to hard code a very specific path that may not work on a different system.

That said, how does Docker know when to work with Docker volumes and when to work with local folders? Well, it is simple, Docker just analyses what is on the left side of the colon (`:`), and if it resolves to a filesystem path, Docker assumes that latter; otherwise, it assumes it is (the name of) a Docker volume.

Run the preceding command, and once inside the container, navigate to the `/app` folder. Inside that folder, create a simple file:

```
cd /app
echo "example data..." > demo.txt
```

Open a second terminal window and navigate to your `src/ch04/my-code` folder. List its content, and you should see that there is a file called `demo.txt` present. Use the `cat` command to output its content, and you will see that it corresponds to the file that you previously created from within the container:

```
gabriel@Solaris ~ % cd src/ch04/my-code
gabriel@Solaris my-code % ls
demo.txt
gabriel@Solaris my-code % cat demo.txt
example data...
gabriel@Solaris my-code %
```

Figure 4.33: Finding data resulting from mounting a local folder into a container

Next, create a file in the local folder with, for example, the following content:

```
echo "content created on host..." > sample.txt
```

Switch to the terminal where you still have your container running and locate the file and print its content with these two commands:

```
ls
cat sample.txt
```

First, you should find the file created on the host now also in your container and also see that its content is indeed what you have expected:

```
/app # ls
demo.txt    sample.txt
/app # cat sample.txt
content created on host...
/app #
```

Figure 4.34: Locating and verifying content created on the host inside the container

Terminate your container by pressing *CTRL-d*.

We have seen that when mounting a host folder into a container, we can share data bidirectionally from the host to the container and vice versa. As mentioned before, this comes in especially handy when developing and testing code for applications running inside a container.

In the last section of this chapter, let us now look into how we can provide runtime configuration to an application running inside a container.

```
cd /app
echo "example data..." > demo.txt
```

Open a second terminal window and navigate to your `src/ch04/my-code` folder. List its content, and you should see that there is a file called `demo.txt` present. Use the `cat` command to output its content, and you will see that it corresponds to the file that you previously created from within the container:

```
gabriel@Solaris ~ % cd src/ch04/my-code
gabriel@Solaris my-code % ls
demo.txt
gabriel@Solaris my-code % cat demo.txt
example data...
gabriel@Solaris my-code %
```

Figure 4.33: Finding data resulting from mounting a local folder into a container

Next, create a file in the local folder with, for example, the following content:

```
echo "content created on host..." > sample.txt
```

Switch to the terminal where you still have your container running and locate the file and print its content with these two commands:

```
ls
cat sample.txt
```

First, you should find the file created on the host now also in your container and also see that its content is indeed what you have expected:

```
/app # ls
demo.txt    sample.txt
/app # cat sample.txt
content created on host...
/app #
```

Figure 4.34: Locating and verifying content created on the host inside the container

Terminate your container by pressing *CTRL-d*.

We have seen that when mounting a host folder into a container, we can share data bidirectionally from the host to the container and vice versa. As mentioned before, this comes in especially handy when developing and testing code for applications running inside a container.

In the last section of this chapter, let us now look into how we can provide runtime configuration to an application running inside a container.

Configuring an app running inside a container

Most applications need some kind of configuration at runtime. Let us say we have an application that accesses a database. It is a best practice to never hard code in the application, which connection string the application uses, but rather develop the application in a way that it uses a configuration setting whose value we can pass at runtime. This way, we keep the application flexible. The application developer does not have to know what DB the app is using during runtime, nor does she have to know the credentials that are used to access the particular database.

When you containerize an application, the same best practices apply. Thus, we need to find a way how we can pass configuration settings to the containerized application at runtime. There are various possibilities to do so. In this section, we will learn the most common and recommended ones.

In Linux, it is a best practice to provide configuration settings via environment variables. But since an application running inside a container is completely shielded from the outside world and specifically from the host system on which it runs, we cannot use the environment variables defined on the host to configure our containerized applications. Let us prove this. If in a terminal, we execute the following command to export all defined environment variables:

```
export
```

then on our MacOS, we see a long list that starts like the following:

```
COLORFGBG='7;0'
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:
ITERM_PROFILE=Default
...
```

On the other hand, if we are exporting all environment variables defined in, say, an Alpine container:

```
docker container run alpine /bin/sh -c "export"
```

Configuring an app running inside a container

Most applications need some kind of configuration at runtime. Let us say we have an application that accesses a database. It is a best practice to never hard code in the application, which connection string the application uses, but rather develop the application in a way that it uses a configuration setting whose value we can pass at runtime. This way, we keep the application flexible. The application developer does not have to know what DB the app is using during runtime, nor does she have to know the credentials that are used to access the particular database.

When you containerize an application, the same best practices apply. Thus, we need to find a way how we can pass configuration settings to the containerized application at runtime. There are various possibilities to do so. In this section, we will learn the most common and recommended ones.

In Linux, it is a best practice to provide configuration settings via environment variables. But since an application running inside a container is completely shielded from the outside world and specifically from the host system on which it runs, we cannot use the environment variables defined on the host to configure our containerized applications. Let us prove this. If in a terminal, we execute the following command to export all defined environment variables:

```
export
```

then on our MacOS, we see a long list that starts like the following:

```
COLORFGBG='7;0'
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:
ITERM_PROFILE=Default
...
```

On the other hand, if we are exporting all environment variables defined in, say, an Alpine container:

```
docker container run alpine /bin/sh -c "export"
```

We will see this:

```
gabriel@Solaris my-code % docker container run alpine /bin/sh -c "export"
export HOME='/root'
export HOSTNAME='543f3a093b96'
export PATH='/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'
export PWD='/'
export SHLVL='1'
gabriel@Solaris my-code %
```

Figure 4.35: Exporting all environment variables in an Alpine container

This is definitely different from what we see on our host.

Luckily, there is a way to define an environment variable and its value in the **docker run** command. Let us say we want to define a variable **FOO** with the value **bar**, then we can modify our command from the preceding like the following:

```
docker container run \
    -e FOO=bar \
    alpine /bin/sh -c "export"
```

The output now looks like this:

```
gabriel@Solaris my-code % docker container run \
>     -e FOO=bar \
>     alpine /bin/sh -c "export"
export FOO='bar'
export HOME='/root'
export HOSTNAME='9a08fad026d0'
export PATH='/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'
export PWD='/'
export SHLVL='1'
gabriel@Solaris my-code %
```

Figure 4.36: Defining an environment variable in the Docker run command

We use the parameter **-e** (or the more explicit **--env**) to define a key-value pair, here **FOO=bar**. Notice that we can define as many variables as we want by repeating the second line in the preceding command. Thus, if we have three variables **ONE**, **TWO**, and **THREE** to define, the command would look like the following:

```
docker container run \
    -e ONE=value1 \
    -e TWO=value2 \
    -e THREE="value3 containing spaces" \
    alpine /bin/sh -c "export"
```

Note the variable **THREE**, whose value contains spaces or other problematic characters. You can use single or double quotes to wrap the value.

If you have a complex application that you need to containerize, the list of values that you will want to configure may grow quite a bit. Quickly you find yourself

We will see this:

```
gabriel@Solaris my-code % docker container run alpine /bin/sh -c "export"
export HOME='/root'
export HOSTNAME='543f3a093b96'
export PATH='/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'
export PWD='/'
export SHLVL='1'
gabriel@Solaris my-code %
```

Figure 4.35: Exporting all environment variables in an Alpine container

This is definitely different from what we see on our host.

Luckily, there is a way to define an environment variable and its value in the **docker run** command. Let us say we want to define a variable **FOO** with the value **bar**, then we can modify our command from the preceding like the following:

```
docker container run \
    -e FOO=bar \
    alpine /bin/sh -c "export"
```

The output now looks like this:

```
gabriel@Solaris my-code % docker container run \
>     -e FOO=bar \
>     alpine /bin/sh -c "export"
export FOO='bar'
export HOME='/root'
export HOSTNAME='9a08fad026d0'
export PATH='/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'
export PWD='/'
export SHLVL='1'
gabriel@Solaris my-code %
```

Figure 4.36: Defining an environment variable in the Docker run command

We use the parameter **-e** (or the more explicit **--env**) to define a key-value pair, here **FOO=bar**. Notice that we can define as many variables as we want by repeating the second line in the preceding command. Thus, if we have three variables **ONE**, **TWO**, and **THREE** to define, the command would look like the following:

```
docker container run \
    -e ONE=value1 \
    -e TWO=value2 \
    -e THREE="value3 containing spaces" \
    alpine /bin/sh -c "export"
```

Note the variable **THREE**, whose value contains spaces or other problematic characters. You can use single or double quotes to wrap the value.

If you have a complex application that you need to containerize, the list of values that you will want to configure may grow quite a bit. Quickly you find yourself

with dozens of values. Do we really have to list every single value in the Docker run command explicitly? It turns out that there is a better way than that. We can use a file in which we list all the configuration values as `<key>=<value>` pairs, one per line. To mimic the preceding example with the variables `ONE`, `TWO`, and `THREE`, you could define a file named `app.conf` and add the following content:

```
ONE=value1
TWO=value2
THREE=value3 containing spaces
```

Then, you can modify the `docker run` command to use the `--env-file` parameter such as that it looks like the following:

```
docker container run \
    --env-file app.conf \
    alpine /bin/sh -c "export"
```

and the output should look like the following:

```
gabriel@Solaris config-demo % docker container run \
    --env-file app.conf \
    alpine /bin/sh -c "export"
export HOME='/root'
export HOSTNAME='a8d03c6ce4c4'
export ONE='value1 '
export PATH='/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'
export PWD='/'
export SHLVL='1'
export THREE='value3 containing spaces'
export TWO='value2 '
gabriel@Solaris config-demo %
```

Figure 4.37: Using configuration files in the Docker run command

The latter approach is really much more convenient as soon as you have more than a handful of variables to define.

Conclusion

In this chapter, we have learned how to use existing images and leverage them either as is or to build custom images upon them. We showed this with the example of a relational database and a no SQL database that we both ran and accessed from a different container running on the same network. We also showed with simple hello world style of applications how you would access a containerized database from a Node JS and a Python application that themselves were containerized.

Next, we learned more about what Docker images are and about their inner workings. We described how images are built up of an immutable stack of layers that each contains the changes to the filesystem in relation to the lower-laying images. As a

with dozens of values. Do we really have to list every single value in the Docker run command explicitly? It turns out that there is a better way than that. We can use a file in which we list all the configuration values as `<key>=<value>` pairs, one per line. To mimic the preceding example with the variables `ONE`, `TWO`, and `THREE`, you could define a file named `app.conf` and add the following content:

```
ONE=value1
TWO=value2
THREE=value3 containing spaces
```

Then, you can modify the `docker run` command to use the `--env-file` parameter such as that it looks like the following:

```
docker container run \
    --env-file app.conf \
    alpine /bin/sh -c "export"
```

and the output should look like the following:

```
gabriel@Solaris config-demo % docker container run \
    --env-file app.conf \
    alpine /bin/sh -c "export"
export HOME='/root'
export HOSTNAME='a8d03c6ce4c4'
export ONE='value1 '
export PATH='/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'
export PWD='/'
export SHLVL='1'
export THREE='value3 containing spaces'
export TWO='value2 '
gabriel@Solaris config-demo %
```

Figure 4.37: Using configuration files in the Docker run command

The latter approach is really much more convenient as soon as you have more than a handful of variables to define.

Conclusion

In this chapter, we have learned how to use existing images and leverage them either as is or to build custom images upon them. We showed this with the example of a relational database and a no SQL database that we both ran and accessed from a different container running on the same network. We also showed with simple hello world style of applications how you would access a containerized database from a Node JS and a Python application that themselves were containerized.

Next, we learned more about what Docker images are and about their inner workings. We described how images are built up of an immutable stack of layers that each contains the changes to the filesystem in relation to the lower-laying images. As a

whole, they make up a complete filesystem that is needed to run the containerized application.

Finally, we got to know what Docker volumes are, how those volumes are used, and what kind of scenarios they are providing a benefit to.

With that, it is time for you to assess your learnings. We have collected a few sample questions you should try to answer before you proceed. To each question, we also provide the (or a possible, if there is more than one) correct answer.

Questions

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

1. **How would you run a Postgres container named `my-db` in detach mode?**
    a. `docker container run postgres:latest -detach -n my-db`
    b. `docker run -d -n my-db postgres`
    c. `docker container run -d --name postgres my-db`
    d. `docker container run --detach --name my-db postgres`

2. **What is a precondition that you can access a Postgres database running locally inside a container from an application running in another container that also runs locally? Multiple answers are possible.**
    a. This is not possible since one cannot run Postgres locally in a container
    b. You need to make sure that in the connection string, you use as the hostname for the database the name of the container in which Postgres runs
    c. To make this possible, Postgres and your application need to both run in the same container
    d. To make this possible, your network admin needs to provide you with special permissions
    e. The two containers, the one with Postgres and the one with your application, need to be attached to the same network

3. **Which is the correct command to mount a local `src` folder into a container `my-app` at its folder `/app/src`?**
    a. `docker container run -v $PWD/src:/app/src my-app`
    b. `docker container exec --volume $PWD/src:/app/src my-app`
    c. `docker -v $PWD/src:/app/src my-app run my-app`
    d. `docker container run my-app --volume $PWD/src:/app/src`

whole, they make up a complete filesystem that is needed to run the containerized application.

Finally, we got to know what Docker volumes are, how those volumes are used, and what kind of scenarios they are providing a benefit to.

With that, it is time for you to assess your learnings. We have collected a few sample questions you should try to answer before you proceed. To each question, we also provide the (or a possible, if there is more than one) correct answer.

Questions

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

1. **How would you run a Postgres container named `my-db` in detach mode?**
    a. `docker container run postgres:latest -detach -n my-db`
    b. `docker run -d -n my-db postgres`
    c. `docker container run -d --name postgres my-db`
    d. `docker container run --detach --name my-db postgres`

2. **What is a precondition that you can access a Postgres database running locally inside a container from an application running in another container that also runs locally? Multiple answers are possible.**
    a. This is not possible since one cannot run Postgres locally in a container
    b. You need to make sure that in the connection string, you use as the hostname for the database the name of the container in which Postgres runs
    c. To make this possible, Postgres and your application need to both run in the same container
    d. To make this possible, your network admin needs to provide you with special permissions
    e. The two containers, the one with Postgres and the one with your application, need to be attached to the same network

3. **Which is the correct command to mount a local `src` folder into a container `my-app` at its folder `/app/src`?**
    a. `docker container run -v $PWD/src:/app/src my-app`
    b. `docker container exec --volume $PWD/src:/app/src my-app`
    c. `docker -v $PWD/src:/app/src my-app run my-app`
    d. `docker container run my-app --volume $PWD/src:/app/src`

4. **What are scenarios in which you want to use Docker volumes?**
    a. Share data between containers
    b. Persist data outside of a container, e.g., in an S3 bucket on AWS
    c. Run stateful applications such as databases inside containers
    d. All of the above
    e. None of the above

5. **You want to pass a configuration setting called `MAX_SUM_INSURED` with value `1M` to the app `my-web` running inside a container. How can you do that if the application expects the value to be set as an environment variable?**
    a. `docker run --env MAX_SUM_INSURED=1M my-web`
    b. `docker run -e MAX_SUM_INSURED=1M my-web`
    c. Define a file `.env` with the content `MAX_SUM_INSURED=1M` and use the following command: `docker run --env-file .env my-web`
    d. None of the above

6. **What are the valid advantages of the fact that Docker images are immutable? Multiple answers are possible.**
    a. A Docker image does not get stale when cached
    b. A Docker image can easily be modified
    c. A Docker image cannot easily be tampered with
    d. I do not have to repeatedly download the same image from the image registry

Answers

1. The correct answer is d
2. b and e are correct
3. Answer a is correct
4. Answer d is correct
5. a, b, and c are correct
6. Answers a, c, and d are correct

4. **What are scenarios in which you want to use Docker volumes?**
    a. Share data between containers
    b. Persist data outside of a container, e.g., in an S3 bucket on AWS
    c. Run stateful applications such as databases inside containers
    d. All of the above
    e. None of the above

5. **You want to pass a configuration setting called `MAX_SUM_INSURED` with value `1M` to the app `my-web` running inside a container. How can you do that if the application expects the value to be set as an environment variable?**
    a. `docker run --env MAX_SUM_INSURED=1M my-web`
    b. `docker run -e MAX_SUM_INSURED=1M my-web`
    c. Define a file `.env` with the content `MAX_SUM_INSURED=1M` and use the following command: `docker run --env-file .env my-web`
    d. None of the above

6. **What are the valid advantages of the fact that Docker images are immutable? Multiple answers are possible.**
    a. A Docker image does not get stale when cached
    b. A Docker image can easily be modified
    c. A Docker image cannot easily be tampered with
    d. I do not have to repeatedly download the same image from the image registry

Answers

1. The correct answer is d
2. b and e are correct
3. Answer a is correct
4. Answer d is correct
5. a, b, and c are correct
6. Answers a, c, and d are correct

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 very sought-after company. During the interview, you are confronted with the following questions:

1. You write an application A that continuously produces some data, which it writes into a file. Application A is supposed to run inside a container. Now, you want to write a second application B, that also will run in a container, but a different one. This application B shall read from the file produced by A and process the data within it. Is this possible? And if yes, what technique would you use?
2. The interviewer says: "We tried to make our development process as friction-free as possible and thus run everything locally using Docker containers. Yet we struggle with one key element, the database. We continue to use a shared database for all developers residing in the cloud. We know that is not the best way of doing it, but we have not found a better solution yet...". What would you answer the interviewer?
3. Imagine you had written a new great Web application. The application needs a lot of configuration settings at runtime. Tell me what techniques you would use to provide the configuration to the web application that is running in a Docker container.
4. Have you ever worked with Docker volumes? If yes, name me some typical scenarios where the use of Docker volumes is a good solution.
5. A Docker image is immutable. What are some advantages of this? Are there also disadvantages that you can identify.

Job interview sample answers

Here are possible answers to the given questions:

1. Yes, the scenario described is possible and can easily be solved by using a Docker volume that is mounted into both containers. We may also want to mount the volume for application B in read-only mode since B is only consuming data that is written by application A. It is always a good practice to only have one party modifying the data and one to many that just read the data.
2. For all popular, there exist Docker images on Docker Hub. This includes SQL and no SQL databases such as Oracle, MS SQL, Postgres, MongoDB, and many more. It is easy and straightforward to run all these databases locally on a developer's machine, given that she has enough RAM (we recommend

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 very sought-after company. During the interview, you are confronted with the following questions:

1. You write an application A that continuously produces some data, which it writes into a file. Application A is supposed to run inside a container. Now, you want to write a second application B, that also will run in a container, but a different one. This application B shall read from the file produced by A and process the data within it. Is this possible? And if yes, what technique would you use?
2. The interviewer says: "We tried to make our development process as friction-free as possible and thus run everything locally using Docker containers. Yet we struggle with one key element, the database. We continue to use a shared database for all developers residing in the cloud. We know that is not the best way of doing it, but we have not found a better solution yet...". What would you answer the interviewer?
3. Imagine you had written a new great Web application. The application needs a lot of configuration settings at runtime. Tell me what techniques you would use to provide the configuration to the web application that is running in a Docker container.
4. Have you ever worked with Docker volumes? If yes, name me some typical scenarios where the use of Docker volumes is a good solution.
5. A Docker image is immutable. What are some advantages of this? Are there also disadvantages that you can identify.

Job interview sample answers

Here are possible answers to the given questions:

1. Yes, the scenario described is possible and can easily be solved by using a Docker volume that is mounted into both containers. We may also want to mount the volume for application B in read-only mode since B is only consuming data that is written by application A. It is always a good practice to only have one party modifying the data and one to many that just read the data.
2. For all popular, there exist Docker images on Docker Hub. This includes SQL and no SQL databases such as Oracle, MS SQL, Postgres, MongoDB, and many more. It is easy and straightforward to run all these databases locally on a developer's machine, given that she has enough RAM (we recommend

at least 16GB). The advantage of local databases is that there is no need to use shared environments with their ever-changing boundary conditions. As a developer, I can work with my database as I see fit and without negatively affecting other developers, which is always a problem when using shared databases.

3. I would definitely collect all settings in a settings file in the form of a list of key-value pairs. I would then pass that content to the container in the run command. The parameter to do so is `--env-file <path-to-file>`. Technically I could also use volumes to share configuration values, but the former is a bit more standard.

4. Typical scenarios where the use of Docker volumes makes sense:
    a. My app needs to read or write files to an EBS volume or an S3 bucket on AWS
    b. My application needs to share files with another application running in a different container
    c. Although it is a special technique, it is worth mentioning: when developing, testing, and debugging applications running inside a container, I may want to mount a local folder containing the code or other artifacts into the container
    d. I have a stateful application that needs to survive crashes, and I do not want to use other means to persist data, such as a database

5. These are some of the advantages of Docker images or image layers being immutable:
    a. I can cache image layers indefinitely without them becoming stale
    b. I do not have to repeatedly download an image and can save bandwidth
    c. Multiple container instances can share the same image layers in memory and thus require less resources, and thus, can also start-up quicker.
    d. It is harder to accidentally tamper with an existing Docker image
    e. If a Docker image runs in one environment, there is a (high) guarantee that it runs in any other environment

at least 16GB). The advantage of local databases is that there is no need to use shared environments with their ever-changing boundary conditions. As a developer, I can work with my database as I see fit and without negatively affecting other developers, which is always a problem when using shared databases.

3. I would definitely collect all settings in a settings file in the form of a list of key-value pairs. I would then pass that content to the container in the run command. The parameter to do so is `--env-file <path-to-file>`. Technically I could also use volumes to share configuration values, but the former is a bit more standard.

4. Typical scenarios where the use of Docker volumes makes sense:
    a. My app needs to read or write files to an EBS volume or an S3 bucket on AWS
    b. My application needs to share files with another application running in a different container
    c. Although it is a special technique, it is worth mentioning: when developing, testing, and debugging applications running inside a container, I may want to mount a local folder containing the code or other artifacts into the container
    d. I have a stateful application that needs to survive crashes, and I do not want to use other means to persist data, such as a database

5. These are some of the advantages of Docker images or image layers being immutable:
    a. I can cache image layers indefinitely without them becoming stale
    b. I do not have to repeatedly download an image and can save bandwidth
    c. Multiple container instances can share the same image layers in memory and thus require less resources, and thus, can also start-up quicker.
    d. It is harder to accidentally tamper with an existing Docker image
    e. If a Docker image runs in one environment, there is a (high) guarantee that it runs in any other environment

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



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 5

Creating Your Own Docker Images

Introduction

In the previous chapter, we have used existing Docker images and demonstrated how to use them for our applications. We particularly were working with containerized SQL and no-SQL databases that we then accessed from another container running in the same container network as the database. We also learned how to use Docker volumes to persist or share data produced or consumed by applications running inside a container. Finally, we were told how to configure an application running in a container at runtime.

Now, it is time that we get to know how we can build our own custom Docker images. This comes in handy when we want to containerize our own applications or custom processes.

Structure

In this chapter, we will discuss the following topics:

- Creating a custom Docker image
- Building a Docker image
- Leveraging the Docker build kit

Chapter 5
Creating Your Own Docker Images

Introduction

In the previous chapter, we have used existing Docker images and demonstrated how to use them for our applications. We particularly were working with containerized SQL and no-SQL databases that we then accessed from another container running in the same container network as the database. We also learned how to use Docker volumes to persist or share data produced or consumed by applications running inside a container. Finally, we were told how to configure an application running in a container at runtime.

Now, it is time that we get to know how we can build our own custom Docker images. This comes in handy when we want to containerize our own applications or custom processes.

Structure

In this chapter, we will discuss the following topics:

- Creating a custom Docker image
- Building a Docker image
- Leveraging the Docker build kit

- Tagging Docker images
- Creating an account on Docker Hub
- Publishing Docker images

Objective

Once you have finished this chapter, you will be able to create your own custom Docker image, explain to an interested layperson in detail how the Docker build process works and leverage the more advanced and higher performant build-kit of Docker to build your custom image. You will also learn how to tag your custom image and publish it to Docker Hub.

Without further ado, let us dive into the first topic and learn how to create our own Docker images.

Creating a custom Docker image

There are basically the following three ways on how one can create a custom Docker image in the system:

1. Interactively build a container and then create an image from it
2. Create a Dockerfile describing the image to be built
3. Create an image by importing it into the system from a tarball

Let us look at each of these three methods in detail but mostly concentrate on the second one, creating a Dockerfile, since it is by far the most important one. It is also the recommended approach in most situations.

We will start with the method where we interactively or manually create a container that will then be used as a template for a custom Docker image.

Creating an image interactively

This method is often used in an exploratory phase when we are not quite sure yet on what we want to build and how we should build it. In other words, we might want to experiment and use a trial-and-error method to achieve the goal. Although this method of creating a custom Docker image is really powerful, it is not recommended for production use. Why?

To make the process of building a Docker image repeatable, and thus, reliable and auditable, we want to automate all the steps. But the process that we are going to investigate right now is very much manual. Thus, we do not recommend it for use in

- Tagging Docker images
- Creating an account on Docker Hub
- Publishing Docker images

Objective

Once you have finished this chapter, you will be able to create your own custom Docker image, explain to an interested layperson in detail how the Docker build process works and leverage the more advanced and higher performant build-kit of Docker to build your custom image. You will also learn how to tag your custom image and publish it to Docker Hub.

Without further ado, let us dive into the first topic and learn how to create our own Docker images.

Creating a custom Docker image

There are basically the following three ways on how one can create a custom Docker image in the system:

1. Interactively build a container and then create an image from it
2. Create a Dockerfile describing the image to be built
3. Create an image by importing it into the system from a tarball

Let us look at each of these three methods in detail but mostly concentrate on the second one, creating a Dockerfile, since it is by far the most important one. It is also the recommended approach in most situations.

We will start with the method where we interactively or manually create a container that will then be used as a template for a custom Docker image.

Creating an image interactively

This method is often used in an exploratory phase when we are not quite sure yet on what we want to build and how we should build it. In other words, we might want to experiment and use a trial-and-error method to achieve the goal. Although this method of creating a custom Docker image is really powerful, it is not recommended for production use. Why?

To make the process of building a Docker image repeatable, and thus, reliable and auditable, we want to automate all the steps. But the process that we are going to investigate right now is very much manual. Thus, we do not recommend it for use in

production but only in a phase where we do proof of concepts or while a developer or DevOps engineer does a spike.

Let us illustrate this process with an example:

1. Open a new terminal window
2. Start an Ubuntu container interactively, run a Bash shell inside this container, and name the container **demo** using the following command:

   ```
   docker container run -it \
       --name demo \
       ubuntu:20.04 /bin/bash
   ```

   Your output in the terminal should look similar to this:

```
gabriel@Solaris ~ % docker container run -it \
    --name demo \
    ubuntu:20.04 /bin/bash
Unable to find image 'ubuntu:20.04' locally
20.04: Pulling from library/ubuntu
6f86eded34a1: Pull complete
Digest: sha256:aba80b77e27148d99c034a987e7da3a287ed455390352663418c0f2ed40417fe
Status: Downloaded newer image for ubuntu:20.04
root@2bc9d37fb73f:/#
```

Figure 5.1: Running a Bash session inside an Ubuntu container

3. Inside the container, we will use the **apt** package manager to install tools and frameworks. Though first, we need to update its package repositories with the following:

   ```
   apt-get update
   ```

   Executing the preceding command results in this output:

```
root@149ef0416714:/# apt update
Get:1 http://security.ubuntu.com/ubuntu focal-security InRelease [114 kB]
Get:2 http://archive.ubuntu.com/ubuntu focal InRelease [265 kB]
Get:3 http://security.ubuntu.com/ubuntu focal-security/multiverse amd64 Packages [27.6 kB]
Get:4 http://security.ubuntu.com/ubuntu focal-security/main amd64 Packages [925 kB]
Get:5 http://security.ubuntu.com/ubuntu focal-security/universe amd64 Packages [778 kB]
Get:6 http://security.ubuntu.com/ubuntu focal-security/restricted amd64 Packages [360 kB]
Get:7 http://archive.ubuntu.com/ubuntu focal-updates InRelease [114 kB]
Get:8 http://archive.ubuntu.com/ubuntu focal-backports InRelease [101 kB]
Get:9 http://archive.ubuntu.com/ubuntu focal/multiverse amd64 Packages [177 kB]
Get:10 http://archive.ubuntu.com/ubuntu focal/main amd64 Packages [1275 kB]
Get:11 http://archive.ubuntu.com/ubuntu focal/restricted amd64 Packages [33.4 kB]
Get:12 http://archive.ubuntu.com/ubuntu focal/universe amd64 Packages [11.3 MB]
Get:13 http://archive.ubuntu.com/ubuntu focal-updates/restricted amd64 Packages [416 kB]
Get:14 http://archive.ubuntu.com/ubuntu focal-updates/universe amd64 Packages [1040 kB]
Get:15 http://archive.ubuntu.com/ubuntu focal-updates/multiverse amd64 Packages [32.0 kB]
Get:16 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 Packages [1361 kB]
Get:17 http://archive.ubuntu.com/ubuntu focal-backports/universe amd64 Packages [6315 B]
Get:18 http://archive.ubuntu.com/ubuntu focal-backports/main amd64 Packages [2668 B]
Fetched 18.4 MB in 3s (6343 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
11 packages can be upgraded. Run 'apt list --upgradable' to see them.
root@149ef0416714:/#
```

Figure 5.2: Updating the apt package manager inside the Ubuntu container

production but only in a phase where we do proof of concepts or while a developer or DevOps engineer does a spike.

Let us illustrate this process with an example:

1. Open a new terminal window
2. Start an Ubuntu container interactively, run a Bash shell inside this container, and name the container **demo** using the following command:

   ```
   docker container run -it \
       --name demo \
       ubuntu:20.04 /bin/bash
   ```

   Your output in the terminal should look similar to this:

```
gabriel@Solaris ~ % docker container run -it \
    --name demo \
    ubuntu:20.04 /bin/bash
Unable to find image 'ubuntu:20.04' locally
20.04: Pulling from library/ubuntu
6f86eded34a1: Pull complete
Digest: sha256:aba80b77e27148d99c034a987e7da3a287ed455390352663418c0f2ed40417fe
Status: Downloaded newer image for ubuntu:20.04
root@2bc9d37fb73f:/#
```

Figure 5.1: Running a Bash session inside an Ubuntu container

3. Inside the container, we will use the **apt** package manager to install tools and frameworks. Though first, we need to update its package repositories with the following:

   ```
   apt-get update
   ```

   Executing the preceding command results in this output:

```
root@149ef0416714:/# apt update
Get:1 http://security.ubuntu.com/ubuntu focal-security InRelease [114 kB]
Get:2 http://archive.ubuntu.com/ubuntu focal InRelease [265 kB]
Get:3 http://security.ubuntu.com/ubuntu focal-security/multiverse amd64 Packages [27.6 kB]
Get:4 http://security.ubuntu.com/ubuntu focal-security/main amd64 Packages [925 kB]
Get:5 http://security.ubuntu.com/ubuntu focal-security/universe amd64 Packages [778 kB]
Get:6 http://security.ubuntu.com/ubuntu focal-security/restricted amd64 Packages [360 kB]
Get:7 http://archive.ubuntu.com/ubuntu focal-updates InRelease [114 kB]
Get:8 http://archive.ubuntu.com/ubuntu focal-backports InRelease [101 kB]
Get:9 http://archive.ubuntu.com/ubuntu focal/multiverse amd64 Packages [177 kB]
Get:10 http://archive.ubuntu.com/ubuntu focal/main amd64 Packages [1275 kB]
Get:11 http://archive.ubuntu.com/ubuntu focal/restricted amd64 Packages [33.4 kB]
Get:12 http://archive.ubuntu.com/ubuntu focal/universe amd64 Packages [11.3 MB]
Get:13 http://archive.ubuntu.com/ubuntu focal-updates/restricted amd64 Packages [416 kB]
Get:14 http://archive.ubuntu.com/ubuntu focal-updates/universe amd64 Packages [1040 kB]
Get:15 http://archive.ubuntu.com/ubuntu focal-updates/multiverse amd64 Packages [32.0 kB]
Get:16 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 Packages [1361 kB]
Get:17 http://archive.ubuntu.com/ubuntu focal-backports/universe amd64 Packages [6315 B]
Get:18 http://archive.ubuntu.com/ubuntu focal-backports/main amd64 Packages [2668 B]
Fetched 18.4 MB in 3s (6343 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
11 packages can be upgraded. Run 'apt list --upgradable' to see them.
root@149ef0416714:/#
```

Figure 5.2: Updating the apt package manager inside the Ubuntu container

4. Now, we are ready to install **ping** from the Linux library **iputils**, which contains a few useful Linux network utilities:

   ```
   apt-get install -y iputils-ping
   ```

   This time we will not show the output of the preceding command here because it is pretty long. But basically, you will be told that **apt** is installing **ping** and some additional support libraries.

5. Verify that **ping** has been installed successfully by trying to access the Google DSN server at **8.8.8.8** for five times:

   ```
   ping 8.8.8.8 -c 5
   ```

   You should see this:

   ```
   root@149ef0416714:/# ping 8.8.8.8 -c 5
   PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
   64 bytes from 8.8.8.8: icmp_seq=1 ttl=110 time=1.21 ms
   64 bytes from 8.8.8.8: icmp_seq=2 ttl=110 time=1.26 ms
   64 bytes from 8.8.8.8: icmp_seq=3 ttl=110 time=1.19 ms
   64 bytes from 8.8.8.8: icmp_seq=4 ttl=110 time=1.22 ms
   64 bytes from 8.8.8.8: icmp_seq=5 ttl=110 time=1.11 ms

   --- 8.8.8.8 ping statistics ---
   5 packets transmitted, 5 received, 0% packet loss, time 4005ms
   rtt min/avg/max/mdev = 1.110/1.197/1.259/0.049 ms
   root@149ef0416714:/#
   ```

   Figure 5.3: Using ping inside an Ubuntu container

Note: If you are using a Mac PowerBook with an M1 processor, the preceding command does not work (at the time of this writing). It is a known problem that may be resolved by the time you are reading this. If it is still not working for you, just try to ping the loopback endpoint at 127.0.0.1.

6. Now, try to install the latest Node JS and the Node package manager **npm** using the Ubuntu package manager:

   ```
   apt-get install -y nodejs npm
   ```

   This will take a moment; please be patient. You can monitor the progress of your terminal output. Note that you will have to answer two questions about the preferred region you are in and the locale during the installation of the node.

7. Now make sure **node** and **npm** are available and working:

   ```
   node --version
   ```
   and
   ```
   npm --version
   ```
   you should see something like this:

4. Now, we are ready to install **ping** from the Linux library **iputils**, which contains a few useful Linux network utilities:

   ```
   apt-get install -y iputils-ping
   ```

   This time we will not show the output of the preceding command here because it is pretty long. But basically, you will be told that **apt** is installing **ping** and some additional support libraries.

5. Verify that **ping** has been installed successfully by trying to access the Google DSN server at **8.8.8.8** for five times:

   ```
   ping 8.8.8.8 -c 5
   ```

   You should see this:

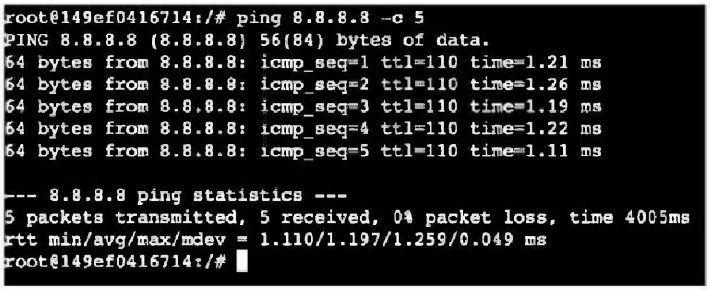


Figure 5.3: Using ping inside an Ubuntu container

Note: If you are using a Mac PowerBook with an M1 processor, the preceding command does not work (at the time of this writing). It is a known problem that may be resolved by the time you are reading this. If it is still not working for you, just try to ping the loopback endpoint at 127.0.0.1.

6. Now, try to install the latest Node JS and the Node package manager **npm** using the Ubuntu package manager:

   ```
   apt-get install -y nodejs npm
   ```

   This will take a moment; please be patient. You can monitor the progress of your terminal output. Note that you will have to answer two questions about the preferred region you are in and the locale during the installation of the node.

7. Now make sure **node** and **npm** are available and working:

   ```
   node --version
   ```
   and
   ```
   npm --version
   ```
   you should see something like this:

```
root@94065f8dacfc:/# node --version
v10.19.0
root@94065f8dacfc:/# npm --version
6.14.4
```

Figure 5.4: Testing that Node JS has been installed correctly

8. If you would rather use Python in your container, try to install it with the following command:

   ```
   apt install -y build-essential libssl-dev libffi-dev python3-dev
   ```

9. Make sure Python 3 has been installed by issuing this command:

   ```
   python3 -V
   ```

   Note the uppercase "V" in the preceding command. In our case, the output is as follows:

   ```
   Python 3.8.10
   ```

10. We can also create files and folders inside the container. Try:

    ```
    mkdir -p /app/src
    cd /app/src
    touch sample.txt
    ```

11. When done installing tools and experimenting, exit the container with Ctrl-D.

12. The container at this time should still be in the system's memory, as you can easily test with the following command:

    ```
    docker container ls -a | grep demo
    ```

    and you should see something similar to this:

```
gabriel@Solaris ~ % docker container ls -a | grep demo
2bc9d37fb73f   ubuntu:20.04   "/bin/bash"   6 hours ago   Exited (0) 29 seconds ago   demo
```

Figure 5.5: The demo container in memory

So far, we have used interactive commands to create a custom container. We have installed additional tools and a whole development environment, namely, Node JS. We also have created custom folders and files inside this container.

Now, let us make a new image from this container, shall we?

1. To do this, use the `docker container commit` command as follows:

   ```
   docker container commit demo my-demo-image:1.0
   ```

   This will create a new image called `my-demo-image:1.0`, taking the Ubuntu as a base layer and converting the container layer of the demo container into a new read-only image layer on top of the base layer.

```
root@94065f8dacfc:/# node --version
v10.19.0
root@94065f8dacfc:/# npm --version
6.14.4
```

Figure 5.4: Testing that Node JS has been installed correctly

8. If you would rather use Python in your container, try to install it with the following command:

   ```
   apt install -y build-essential libssl-dev libffi-dev python3-dev
   ```

9. Make sure Python 3 has been installed by issuing this command:

   ```
   python3 -V
   ```

   Note the uppercase "V" in the preceding command. In our case, the output is as follows:

   ```
   Python 3.8.10
   ```

10. We can also create files and folders inside the container. Try:

    ```
    mkdir -p /app/src
    cd /app/src
    touch sample.txt
    ```

11. When done installing tools and experimenting, exit the container with Ctrl-D.

12. The container at this time should still be in the system's memory, as you can easily test with the following command:

    ```
    docker container ls -a | grep demo
    ```

    and you should see something similar to this:

```
gabriel@Solaris ~ % docker container ls -a | grep demo
2bc9d37fb73f   ubuntu:20.04   "/bin/bash"   6 hours ago   Exited (0) 29 seconds ago   demo
```

Figure 5.5: The demo container in memory

So far, we have used interactive commands to create a custom container. We have installed additional tools and a whole development environment, namely, Node JS. We also have created custom folders and files inside this container.

Now, let us make a new image from this container, shall we?

1. To do this, use the `docker container commit` command as follows:

   ```
   docker container commit demo my-demo-image:1.0
   ```

   This will create a new image called `my-demo-image:1.0`, taking the Ubuntu as a base layer and converting the container layer of the demo container into a new read-only image layer on top of the base layer.

2. Look for the container image in your cache with the following:
   `docker image ls | grep my-demo`
   and you should see something like this:

```
gabriel@Solaris ~ % docker image ls | grep my-demo
my-demo-image            1.0       c26865bd9cd7   33 seconds ago   557MB
```

Figure 5.6: New image created from interactively create demo container

Finally, we can use the history command to see the layers of the image we just created:

```
docker image history my-demo-image:1.0
```

and we should see this:

```
gabriel@Solaris ~ % docker image history my-demo-image:1.0
IMAGE          CREATED         CREATED BY                                      SIZE      COMMENT
c26865bd9cd7   6 minutes ago   /bin/bash                                       491MB
8707eaf21636   2 weeks ago     /bin/sh -c #(nop)  CMD ["bash"]                 0B
<missing>      2 weeks ago     /bin/sh -c #(nop) ADD file:9e94a579f0495fce8…   65.6MB
```

Figure 5.7: History of the newly created image

   You can see in the preceding image that we have three layers. The last layer in the list is the base layer containing the Ubuntu 20.04 distro. The middle layer is also part of the Ubuntu image. Finally, the first layer, whose IMAGE column contains the value `c26865...`, was created by our commit command preceding. Since we have installed quite a bit of software, the layer grew considerably to a size of 491 MB.

3. **Optional:** Try running a container from the `my-demo-image:1.0` image interactively with a Bash shell and verify that the `ping` tool, Node JS and Python are still there and usable. Also, look for the folders and files that you created in the demo container. They should also be present in this new container.

With this, we conclude this section, where we explored how one can create an image from a container that has been configured interactively. This is specifically interesting when you are in an investigation phase and do not yet know how exactly you will be constructing a container in a fully automated way.

How to construct an image in a fully automated way is the next topic we are going to investigate.

Authoring a Dockerfile

The Dockerfile is a text file, and by default, it is literally called `Dockerfile`. The Dockerfile is a manifest of what is going to be part of the Docker image we want

2. Look for the container image in your cache with the following:
   `docker image ls | grep my-demo`
   and you should see something like this:

```
gabriel@Solaris ~ % docker image ls | grep my-demo
my-demo-image            1.0       c26865bd9cd7   33 seconds ago   557MB
```

Figure 5.6: New image created from interactively create demo container

Finally, we can use the history command to see the layers of the image we just created:

```
docker image history my-demo-image:1.0
```

and we should see this:

```
gabriel@Solaris ~ % docker image history my-demo-image:1.0
IMAGE          CREATED         CREATED BY                                      SIZE      COMMENT
c26865bd9cd7   6 minutes ago   /bin/bash                                       491MB
8707eaf21636   2 weeks ago     /bin/sh -c #(nop)  CMD ["bash"]                 0B
<missing>      2 weeks ago     /bin/sh -c #(nop) ADD file:9e94a579f0495fce8…   65.6MB
```

Figure 5.7: History of the newly created image

   You can see in the preceding image that we have three layers. The last layer in the list is the base layer containing the Ubuntu 20.04 distro. The middle layer is also part of the Ubuntu image. Finally, the first layer, whose IMAGE column contains the value `c26865...`, was created by our commit command preceding. Since we have installed quite a bit of software, the layer grew considerably to a size of 491 MB.

3. **Optional:** Try running a container from the `my-demo-image:1.0` image interactively with a Bash shell and verify that the `ping` tool, Node JS and Python are still there and usable. Also, look for the folders and files that you created in the demo container. They should also be present in this new container.

With this, we conclude this section, where we explored how one can create an image from a container that has been configured interactively. This is specifically interesting when you are in an investigation phase and do not yet know how exactly you will be constructing a container in a fully automated way.

How to construct an image in a fully automated way is the next topic we are going to investigate.

Authoring a Dockerfile

The Dockerfile is a text file, and by default, it is literally called `Dockerfile`. The Dockerfile is a manifest of what is going to be part of the Docker image we want

to build. It is a declarative instruction to the Docker builder tool on how to create the image. Due to the fact that the content of the Dockerfile is declarative, it is very readable, and the Dockerfile can, thus, serve as a documentation of the image.

In this section, we will get to know various keywords that we can use in a Dockerfile and their meaning. Some of them are `FROM`, `COPY`, or `RUN`. But let us first start by explaining how the image is actually built by the Docker image builder.

The Docker image build process

In this section, we look at the details of how a Docker image is actually built. For this, we start with a very simple Dockerfile that serves as a manifest for the image we want to build as an example.

Note, for our description, we use the default Docker image builder, which is part of Docker Desktop. We are not using the also included image builder based on the build kit since that builder is highly optimized and more involved to explain. The basic principles are the same, though, and that is why here we stick with the original builder.

Here is the content of the Dockerfile we are going to use is as follows:

```
FROM node:latest
WORKDIR /app
COPY package.json ./
RUN npm install
COPY . .
CMD node server.js
```

This is one of the Dockerfiles we have used in the previous chapters to build a Node JS client for accessing a Postgres database.

As you can see, the Dockerfile consists of exactly six instructions, each one on a separate line. This is important as the image builder will build a layer for each instruction in the Dockerfile. The builder will start with the first line containing the `FROM node:latest` instruction. It will create the first few layers from this. How many that will depend on how many layers the base image consists of. In our case,

to build. It is a declarative instruction to the Docker builder tool on how to create the image. Due to the fact that the content of the Dockerfile is declarative, it is very readable, and the Dockerfile can, thus, serve as a documentation of the image.

In this section, we will get to know various keywords that we can use in a Dockerfile and their meaning. Some of them are `FROM`, `COPY`, or `RUN`. But let us first start by explaining how the image is actually built by the Docker image builder.

The Docker image build process

In this section, we look at the details of how a Docker image is actually built. For this, we start with a very simple Dockerfile that serves as a manifest for the image we want to build as an example.

Note, for our description, we use the default Docker image builder, which is part of Docker Desktop. We are not using the also included image builder based on the build kit since that builder is highly optimized and more involved to explain. The basic principles are the same, though, and that is why here we stick with the original builder.

Here is the content of the Dockerfile we are going to use is as follows:

```
FROM node:latest
WORKDIR /app
COPY package.json ./
RUN npm install
COPY . .
CMD node server.js
```

This is one of the Dockerfiles we have used in the previous chapters to build a Node JS client for accessing a Postgres database.

As you can see, the Dockerfile consists of exactly six instructions, each one on a separate line. This is important as the image builder will build a layer for each instruction in the Dockerfile. The builder will start with the first line containing the `FROM node:latest` instruction. It will create the first few layers from this. How many that will depend on how many layers the base image consists of. In our case,

the `node:latest` image consists of nine layers, which we can see when we try to pull the image from Docker Hub as shown as follows:

```
gabriel@solaris ~ % docker image pull node:latest
latest: Pulling from library/node
310b368da982: Downloading [==========>                                ]  11.11MB/49.22MB
86422c44ee00: Download complete
9137877e0c26: Download complete
785171b903c4: Downloading [==========>                                ]  10.02MB/52.17MB
0373cebfde2c: Downloading [>                                          ]    535kB/184MB
86a7b2799124: Waiting
026d4a372991: Waiting
99f213140006: Waiting
49285d4db505: Waiting
```

Figure 5.8: Pulling the latest Node image from Docker Hub

Thus, we will have nine layers as a result of this first instruction.

Next, the builder will process the second line, in this case, `WORKDIR /app`, and create a new layer from it. This new layer will go "on top" of the previous nine layers and build a new intermediary image with 10 layers.

After this, the builder will take the third line with the instruction `COPY package.json./` and create yet another layer. The intermediary image will now have 11 layers.

This process will go on until the image builder has processed all instructions in the Dockerfile. In the end, we have a resulting image that will have 14 layers.

We will now try to illustrate this with the following graphic:

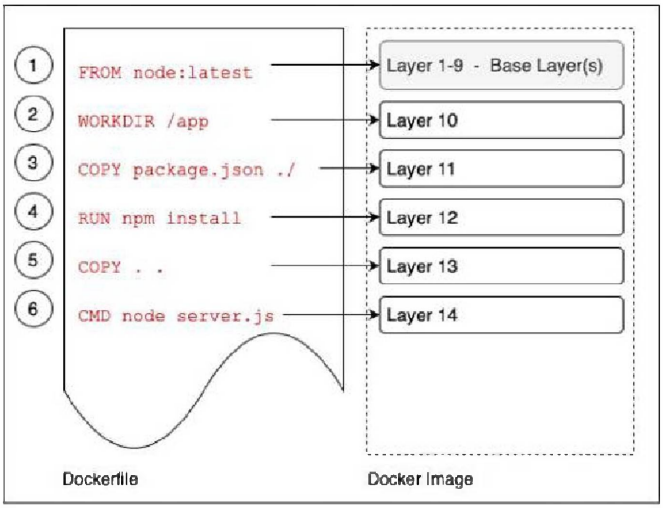


Figure 5.9: The relation between a Dockerfile and the layers in an image

the `node:latest` image consists of nine layers, which we can see when we try to pull the image from Docker Hub as shown as follows:

```
gabriel@solaris ~ % docker image pull node:latest
latest: Pulling from library/node
310b368da982: Downloading [==========>                                ]  11.11MB/49.22MB
86422c44ee00: Download complete
9137877e0c26: Download complete
785171b903c4: Downloading [==========>                                ]  10.02MB/52.17MB
0373cebfde2c: Downloading [>                                          ]   535kB/184MB
86a7b2799124: Waiting
026d4a372991: Waiting
99f213140006: Waiting
49285d4db505: Waiting
```

Figure 5.8: Pulling the latest Node image from Docker Hub

Thus, we will have nine layers as a result of this first instruction.

Next, the builder will process the second line, in this case, `WORKDIR /app`, and create a new layer from it. This new layer will go "on top" of the previous nine layers and build a new intermediary image with 10 layers.

After this, the builder will take the third line with the instruction `COPY package.json./` and create yet another layer. The intermediary image will now have 11 layers.

This process will go on until the image builder has processed all instructions in the Dockerfile. In the end, we have a resulting image that will have 14 layers.

We will now try to illustrate this with the following graphic:

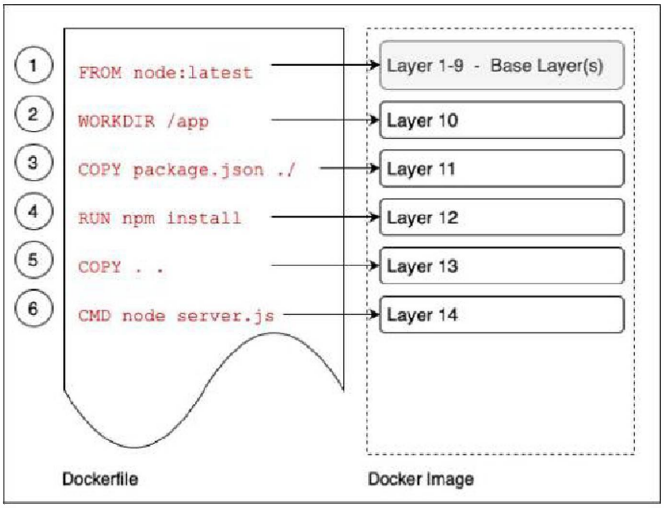


Figure 5.9: The relation between a Dockerfile and the layers in an image

In the preceding graphic, we can see the Dockerfile on the left with the six lines, each containing an instruction. On the right side, we see the resulting Docker image. The arrows in the middle associate each instruction in the Dockerfile with the corresponding layer of the image. Note that usually, we say that an image consists of a stack of image layers, with the base layer at the bottom. In the preceding picture, the direction of the stack is reversed, and the base layer is at the top.

Now, you might wonder what exactly the builder is doing when building the image. Let us explain this again using the preceding Dockerfile as an example:

1. The builder sees the `FROM` instruction and peeks into the local image cache to see if there is an image called `node:latest`. If not, then he pulls the image from Docker Hub.

2. The builder goes to the second instruction, here `WORKDIR /app`. The builder creates a temporary container from the result of Step 1 (which are Layers 1–9) and executes the `WORKDIR` command inside this temporary container. The result of this instruction is applied to the writeable container layer of this temporary container.

3. Now, the builder creates a new image layer (here, Layer 10) from the preceding container layer using the technique we learned in the previous section when we discussed the `docker container commit` command.

4. Next, the builder takes instruction 3, which is `COPY package.json ./`. For this, it again creates an intermediary container from the result of the previous step—in this case, it will be Layers 1–10. Inside this new temporary container, the builder will execute the `COPY` instruction. Again, the results of this instruction are applied to the container layer.

5. The builder once again creates an image layer (Layer 11) from that container layer.

6. The process continues until the whole Dockerfile is processed.

The build process is thus an iterative creation of temporary containers inside which the builder executes the respective instructions. Each of those steps leads to the creation of a new layer.

In the preceding graphic, we can see the Dockerfile on the left with the six lines, each containing an instruction. On the right side, we see the resulting Docker image. The arrows in the middle associate each instruction in the Dockerfile with the corresponding layer of the image. Note that usually, we say that an image consists of a stack of image layers, with the base layer at the bottom. In the preceding picture, the direction of the stack is reversed, and the base layer is at the top.

Now, you might wonder what exactly the builder is doing when building the image. Let us explain this again using the preceding Dockerfile as an example:

1. The builder sees the `FROM` instruction and peeks into the local image cache to see if there is an image called `node:latest`. If not, then he pulls the image from Docker Hub.

2. The builder goes to the second instruction, here `WORKDIR /app`. The builder creates a temporary container from the result of Step 1 (which are Layers 1–9) and executes the `WORKDIR` command inside this temporary container. The result of this instruction is applied to the writeable container layer of this temporary container.

3. Now, the builder creates a new image layer (here, Layer 10) from the preceding container layer using the technique we learned in the previous section when we discussed the `docker container commit` command.

4. Next, the builder takes instruction 3, which is `COPY package.json ./`. For this, it again creates an intermediary container from the result of the previous step—in this case, it will be Layers 1–10. Inside this new temporary container, the builder will execute the `COPY` instruction. Again, the results of this instruction are applied to the container layer.

5. The builder once again creates an image layer (Layer 11) from that container layer.

6. The process continues until the whole Dockerfile is processed.

The build process is thus an iterative creation of temporary containers inside which the builder executes the respective instructions. Each of those steps leads to the creation of a new layer.

We have tried to illustrate what we discussed earlier in words with the following graphic:

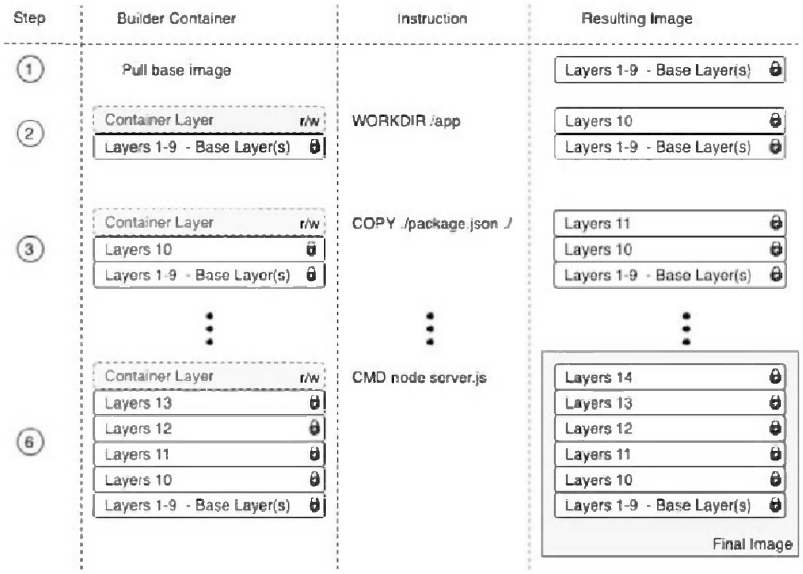


Figure 5.10: Build process of a Docker image

Now that we have a general understanding of the build process, let us have a more detailed look at the individual keywords that can be used in a Dockerfile. We will start with the FROM keyword.

The FROM keyword

Each image that we want to build has to start from a base. We call this baseline the base image. Often this base image corresponds to a popular Linux distro such as Alpine, Centos, or Ubuntu, to just name a few. Other times, we want to use a base image that contains a particular framework or development environment such as Python, Node JS, .NET, or Java.

Usually, we declare from which base image we want to start at the very beginning of the Dockerfile. So, if, for example, we want to start with Ubuntu 20.04 as our base, then the statement in the Dockerfile would look like the following:

```
FROM ubuntu:20.04
```

Analogous, if we want to containerize a Python application and consequently start from the official Python 3.x base image, we will use this statement:

We have tried to illustrate what we discussed earlier in words with the following graphic:

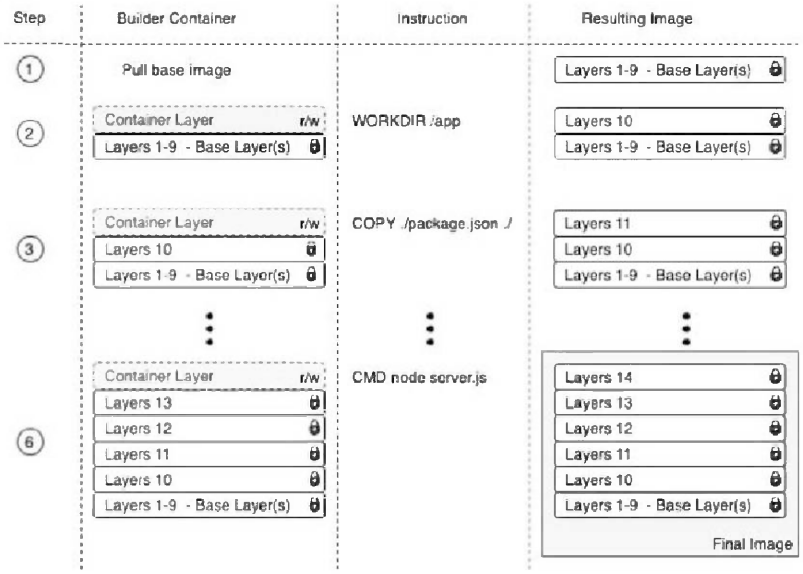


Figure 5.10: Build process of a Docker image

Now that we have a general understanding of the build process, let us have a more detailed look at the individual keywords that can be used in a Dockerfile. We will start with the FROM keyword.

The FROM keyword

Each image that we want to build has to start from a base. We call this baseline the base image. Often this base image corresponds to a popular Linux distro such as Alpine, Centos, or Ubuntu, to just name a few. Other times, we want to use a base image that contains a particular framework or development environment such as Python, Node JS, .NET, or Java.

Usually, we declare from which base image we want to start at the very beginning of the Dockerfile. So, if, for example, we want to start with Ubuntu 20.04 as our base, then the statement in the Dockerfile would look like the following:

```
FROM ubuntu:20.04
```

Analogous, if we want to containerize a Python application and consequently start from the official Python 3.x base image, we will use this statement:

```
FROM python:3
```

At times, we do not want to use any base layer at all, for example, when we want to create a minimal image where we just want to pack an executable such as a C++ or a Go application into a container that has no external dependencies. In this case, we can use the keyword:

```
FROM scratch.
```

During the build operation `FROM scratch` is the equivalent of a `no-op` operation and, as such, does not generate a layer.

The COPY and ADD keywords

The `COPY` and `ADD` keywords are used to copy files and folders from a source to a destination. The source is usually files and folders present on the host computer, and the target is the filesystem that we want to build up inside the container. The two keywords are somewhat similar in what they do but have subtle differences. The two main differences are as follows:

- When using the `ADD` keyword, one can copy and, at the same unpack tar files
- Furthermore, with the `ADD` keyword, one can use URLs to describe the source of files and folders to copy to the destination inside the container.

Let us present and analyze a few typical `ADD` and `COPY` operations that you may encounter in existing Dockerfiles or that you can use in your own Dockerfiles. First of all, the command is used as follows:

```
ADD <source> <destination>
```

or

```
COPY <source> <destination>
```

where `<source>` represents the source files and/or folders and `<target>` describes where the files and folders should be copied to inside the filesystem of the container. Note that the `<source>` and `<destination>` parts are separated by space.

Here are a few specific samples:

1. `COPY ./package.json /app/src/`

   Often when dealing with Dockerfiles written to package a Node JS application, one might encounter the preceding statement. What it does is to copy a file called `package.json` from the local directory on the host computer into a destination folder called `app/src` defined in the root of the container's filesystem. The copied file will have the same name as the source; that is, it will also be called `package.json`

```
FROM python:3
```

At times, we do not want to use any base layer at all, for example, when we want to create a minimal image where we just want to pack an executable such as a C++ or a Go application into a container that has no external dependencies. In this case, we can use the keyword:

```
FROM scratch.
```

During the build operation `FROM scratch` is the equivalent of a `no-op` operation and, as such, does not generate a layer.

The COPY and ADD keywords

The `COPY` and `ADD` keywords are used to copy files and folders from a source to a destination. The source is usually files and folders present on the host computer, and the target is the filesystem that we want to build up inside the container. The two keywords are somewhat similar in what they do but have subtle differences. The two main differences are as follows:

- When using the `ADD` keyword, one can copy and, at the same unpack tar files
- Furthermore, with the `ADD` keyword, one can use URLs to describe the source of files and folders to copy to the destination inside the container.

Let us present and analyze a few typical `ADD` and `COPY` operations that you may encounter in existing Dockerfiles or that you can use in your own Dockerfiles. First of all, the command is used as follows:

```
ADD <source> <destination>
```

or

```
COPY <source> <destination>
```

where `<source>` represents the source files and/or folders and `<target>` describes where the files and folders should be copied to inside the filesystem of the container. Note that the `<source>` and `<destination>` parts are separated by space.

Here are a few specific samples:

1. `COPY ./package.json /app/src/`

   Often when dealing with Dockerfiles written to package a Node JS application, one might encounter the preceding statement. What it does is to copy a file called `package.json` from the local directory on the host computer into a destination folder called `app/src` defined in the root of the container's filesystem. The copied file will have the same name as the source; that is, it will also be called `package.json`

2. `COPY ./foo.txt /data/bar.txt`

   In the previous statement, the resulting file had the same name as the copied one. With this command here, on the other hand, Docker copies a file called `foo.txt` from the local directory on the host into a folder `/data` in the container's filesystem. At the same time, it renames the file from `foo.txt` to `bar.txt`.

3. `COPY . /app/src`

   If we want to copy all the content of the current folder on the host with all its nested subfolders, then we can use the preceding statement. The files and folders are copied into the destination folder `/app/src` in the container's filesystem.

4. `ADD https://acme.com/static/data.xml /data/`

   This command copies a file called `data.xml` from an external source identified by a complete URL (`https://acme.com/static/data.xml`). The file is copied to a target folder `/data` maintaining the same file name. Please note the slash (`/`) at the end of the target (`/data/`)

5. `ADD ./bin/my-app.tar /app/bin/`

   Here, we copy the files and folders compressed into the file `bin/my-app.tar` from the host into the target folder `/app/bin` in the container's filesystem. The content of `my-app.tar` is decompressed during this operation. Thus, in the target folder, we will see the decompressed files and folders (and not the `tar` file)

We can also work with wildcards to identify a set of source files and folders:

1. `COPY ./docs/tangram-*.doc /docs/`

   Here, we copy all Word documents whose name starts with `tangram-` from a source folder `docs` into the `/docs` folder in the target file system

2. `COPY ./**/*.js /scripts/`

   NOTE: The preceding statement does not work!

   Here, we want to copy all JavaScript files that are somewhere nested inside the current working folder on the host to the target file system. Unfortunately, this **does not work** because Docker would not know how to handle files from different subfolders having the same name. Similarly, an analogous Bash copy command (`cp`) would also not work.

Sometimes we need to change the ownership of files and folders that we want to copy into a container. It is important to point out from a security perspective that, by default, all files and folders will have a user ID (`UID`) and a group ID (`GID`) of zero.

2. `COPY ./foo.txt /data/bar.txt`

   In the previous statement, the resulting file had the same name as the copied one. With this command here, on the other hand, Docker copies a file called `foo.txt` from the local directory on the host into a folder `/data` in the container's filesystem. At the same time, it renames the file from `foo.txt` to `bar.txt`.

3. `COPY . /app/src`

   If we want to copy all the content of the current folder on the host with all its nested subfolders, then we can use the preceding statement. The files and folders are copied into the destination folder `/app/src` in the container's filesystem.

4. `ADD https://acme.com/static/data.xml /data/`

   This command copies a file called `data.xml` from an external source identified by a complete URL (`https://acme.com/static/data.xml`). The file is copied to a target folder `/data` maintaining the same file name. Please note the slash (`/`) at the end of the target (`/data/`)

5. `ADD ./bin/my-app.tar /app/bin/`

   Here, we copy the files and folders compressed into the file `bin/my-app.tar` from the host into the target folder `/app/bin` in the container's filesystem. The content of `my-app.tar` is decompressed during this operation. Thus, in the target folder, we will see the decompressed files and folders (and not the `tar` file)

We can also work with wildcards to identify a set of source files and folders:

1. `COPY ./docs/tangram-*.doc /docs/`

   Here, we copy all Word documents whose name starts with `tangram-` from a source folder `docs` into the `/docs` folder in the target file system

2. `COPY ./**/*.js /scripts/`

   NOTE: The preceding statement does not work!

   Here, we want to copy all JavaScript files that are somewhere nested inside the current working folder on the host to the target file system. Unfortunately, this **does not work** because Docker would not know how to handle files from different subfolders having the same name. Similarly, an analogous Bash copy command (`cp`) would also not work.

Sometimes we need to change the ownership of files and folders that we want to copy into a container. It is important to point out from a security perspective that, by default, all files and folders will have a user ID (`UID`) and a group ID (`GID`) of zero.

On a Unix system, the root account has `UID=0`. Of course, the root user inside the container has nothing to do with the root user on the host, as we will see in more detail later in this book.

Fortunately, the ADD and COPY commands both allow us to change ownership during the copy operation. Let us assume there exists a user with `UID=11` and a group with `GID=22` inside the container. We can then change the ownership of the files and folders to be copied using the optional `--chown` parameter as follows:

```
ADD --chown 11:22 ./personal/*.doc /docs/
```

This will copy all Word documents from the `personal` folder into the target `/docs` folder and, at the same time, assign them to `UID=11` and `GID=22`.

Now that we are able to copy files and folders from outside into our custom Docker image, we want to investigate how we can run commands or whole scripts inside the builder container during the build process.

The RUN keyword

The `RUN` keyword is used whenever we want to execute a command or a script inside the container image to build. Any valid Linux command can be used. It can be simple commands or commands that are made up from multiple sub-commands concatenated with the logical AND operator (`&&`).

Let us look at a few examples:

1. `RUN mkdir -p /app/src`

   The preceding statement instructs the Docker image builder to run the command

   `mkdir -p /app/src`

   inside the temporary container from which the image layer will be built.

2. `RUN npm install`

   The preceding statement you will often encounter in Dockerfiles that are used to build and package Node JS applications. With this statement, the Docker image builder will run `npm install` from within the current working folder inside the container. This, as any Node JS developer knows, will install all dependencies of the node application that is containerized.

3. Similarly, in a Dockerfile used for building and packaging a Python app, you may encounter this statement:

   `RUN pip install -r requirements.txt`

   Analogous to the previous statement (2.), this one executes the command

   `pip install -r requirements.txt`

   in the current working folder inside the container.

On a Unix system, the root account has `UID=0`. Of course, the root user inside the container has nothing to do with the root user on the host, as we will see in more detail later in this book.

Fortunately, the ADD and COPY commands both allow us to change ownership during the copy operation. Let us assume there exists a user with `UID=11` and a group with `GID=22` inside the container. We can then change the ownership of the files and folders to be copied using the optional `--chown` parameter as follows:

```
ADD --chown 11:22 ./personal/*.doc /docs/
```

This will copy all Word documents from the `personal` folder into the target `/docs` folder and, at the same time, assign them to `UID=11` and `GID=22`.

Now that we are able to copy files and folders from outside into our custom Docker image, we want to investigate how we can run commands or whole scripts inside the builder container during the build process.

The RUN keyword

The `RUN` keyword is used whenever we want to execute a command or a script inside the container image to build. Any valid Linux command can be used. It can be simple commands or commands that are made up from multiple sub-commands concatenated with the logical AND operator (`&&`).

Let us look at a few examples:

1. `RUN mkdir -p /app/src`

   The preceding statement instructs the Docker image builder to run the command

   `mkdir -p /app/src`

   inside the temporary container from which the image layer will be built.

2. `RUN npm install`

   The preceding statement you will often encounter in Dockerfiles that are used to build and package Node JS applications. With this statement, the Docker image builder will run `npm install` from within the current working folder inside the container. This, as any Node JS developer knows, will install all dependencies of the node application that is containerized.

3. Similarly, in a Dockerfile used for building and packaging a Python app, you may encounter this statement:

   `RUN pip install -r requirements.txt`

   Analogous to the previous statement (2.), this one executes the command

   `pip install -r requirements.txt`

   in the current working folder inside the container.

Now, let us look at a somewhat more complex sample where we want to execute a set of commands in one go:

```
RUN apt-get update && apt-get install -y iputils-ping
```

Here the two commands `apt-get update` and `apt-get install-y iputils-ping` are executed inside the (temporary) container used by the builder during the build process. It is important to notice that although we have two commands that are executed, they only result in a single new image layer since they are handled in a single `RUN` statement.

5. And now, to conclude, we give you the following extremely long command:

```
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/*
```

Note it is valid and even recommended for readability to format the preceding command using multiple virtual lines. In Linux, the line continuation character is a backslash (\). If you are dealing with PowerShell, then you would have to use a backtick (`) instead.

Challenge: Try to decipher what the preceding command is doing.

Now on to the next important keyword.

The WORKDIR keyword

The `WORKDIR` keyword defines the working directory in the context of the container file system. All subsequent operations and commands will be executed in relation to that directory. Thus, the following statement:

```
WORKDIR /app/src
```

It signifies that the execution context in the container is now set to the folder `/app/src`. That is if we have the following combination of instructions in a Dockerfile:

```
WORKDIR /app/src
RUN npm install
COPY . .
```

Now, let us look at a somewhat more complex sample where we want to execute a set of commands in one go:

```
RUN apt-get update && apt-get install -y iputils-ping
```

Here the two commands `apt-get update` and `apt-get install-y iputils-ping` are executed inside the (temporary) container used by the builder during the build process. It is important to notice that although we have two commands that are executed, they only result in a single new image layer since they are handled in a single `RUN` statement.

5. And now, to conclude, we give you the following extremely long command:

```
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/*
```

Note it is valid and even recommended for readability to format the preceding command using multiple virtual lines. In Linux, the line continuation character is a backslash (\). If you are dealing with PowerShell, then you would have to use a backtick (`) instead.

Challenge: Try to decipher what the preceding command is doing.

Now on to the next important keyword.

The WORKDIR keyword

The `WORKDIR` keyword defines the working directory in the context of the container file system. All subsequent operations and commands will be executed in relation to that directory. Thus, the following statement:

```
WORKDIR /app/src
```

It signifies that the execution context in the container is now set to the folder `/app/src`. That is if we have the following combination of instructions in a Dockerfile:

```
WORKDIR /app/src
RUN npm install
COPY . .
```

This would execute the command `npm install` in the context of the working directory `/app/src`. That is, the `npm` tool would expect the corresponding `package.json` file that is needed to successfully execute the `install` command to reside in that folder. Also, the subsequent `COPY` instruction would copy all files and folders from the current directory on the host to the current working folder—which is `/app/src`—in the container file system.

Note that it is completely legal to use the `WORKDIR` command multiple times inside a Dockerfile and thus change the execution context several times.

Please also note that the instruction:

```
WORKDIR /some/path
```

It is permanently changing the context to said path, but that the following command does not change the working folder permanently.

```
RUN cd /some/path
```

The latter command would only make sense in a single logical statement consisting of multiple sub-commands, for example, a command sequence like the following one:

```
RUN cd /app/src && npm install
```

Only since this command is a single logical statement in the context of building a Docker image, the `cd` sub-command has any influence on where subsequent commands are executed.

Thus, we recommend, as a best practice, to always use the `WORKDIR` keyword if you want to change the working folder inside the container file system and not to rely on the `cd` command.

The CMD and ENTRYPOINT keywords

The `CMD` and `ENTRYPOINT` keywords are different than all the other valid keywords we got to know so far. While all the latter keywords define some operation that happens at the build time of the container image, the `CMD` and `ENTRYPOINT` keywords define what shall happen when a container is created and then started from that image. So `CMD` and `ENTRYPOINT` define what shall happen at runtime and not at build time.

The two are somewhat related and, on the surface, do the same thing. But there exist subtle yet important differences. Most people do not know the difference and use the two keywords interchangeably. Luckily in most cases, that is not a problem.

Let us explore them. The correct usage is to define the command that shall be executed when a container is started with the `ENTRYPOINT` keyword and the parameters to

This would execute the command `npm install` in the context of the working directory `/app/src`. That is, the `npm` tool would expect the corresponding `package.json` file that is needed to successfully execute the `install` command to reside in that folder. Also, the subsequent `COPY` instruction would copy all files and folders from the current directory on the host to the current working folder—which is `/app/src`—in the container file system.

Note that it is completely legal to use the `WORKDIR` command multiple times inside a Dockerfile and thus change the execution context several times.

Please also note that the instruction:

```
WORKDIR /some/path
```

It is permanently changing the context to said path, but that the following command does not change the working folder permanently.

```
RUN cd /some/path
```

The latter command would only make sense in a single logical statement consisting of multiple sub-commands, for example, a command sequence like the following one:

```
RUN cd /app/src && npm install
```

Only since this command is a single logical statement in the context of building a Docker image, the `cd` sub-command has any influence on where subsequent commands are executed.

Thus, we recommend, as a best practice, to always use the `WORKDIR` keyword if you want to change the working folder inside the container file system and not to rely on the `cd` command.

The CMD and ENTRYPOINT keywords

The `CMD` and `ENTRYPOINT` keywords are different than all the other valid keywords we got to know so far. While all the latter keywords define some operation that happens at the build time of the container image, the `CMD` and `ENTRYPOINT` keywords define what shall happen when a container is created and then started from that image. So `CMD` and `ENTRYPOINT` define what shall happen at runtime and not at build time.

The two are somewhat related and, on the surface, do the same thing. But there exist subtle yet important differences. Most people do not know the difference and use the two keywords interchangeably. Luckily in most cases, that is not a problem.

Let us explore them. The correct usage is to define the command that shall be executed when a container is started with the `ENTRYPOINT` keyword and the parameters to

the command with the `CMD` keyword. Thus, if you have the following typical Linux expression

```
ping 8.8.8.8 -c 5
```

then `ping` is the command, and `8.8.8.8 -c 5` are the parameters to this command. Consequently, one would define this in a Dockerfile as follows:

```
...
ENTRYPOINT ping
CMD 8.8.8.8 -c
```

Another example could be the following expression that downloads a script from a given URL:

```
wget -O - https://www.acme.com/cool-tool/install.sh
```

In this case, `wget` is the command and

```
-O - https://www.acme.com/cool-tool/install.sh
```

are the parameters to that command. Consequently, the Dockerfile would look like the following:

```
...
ENTRYPOINT wget
CMD -O - https://www.acme.com/cool-tool/install.sh
```

If it is not defined explicitly, that is, if `ENTRYPOINT` is omitted in the Dockerfile, its default value is `/bin/sh -c`.

Thus, if you only define a value for `CMD` like:

```
CMD <some value>
```

then the resulting container start command will be

```
/bin/sh -c "<some value>"
```

which is a valid Linux command and will just use the Bourne shell to execute whatever command you defined for `<some value>`.

Thus, if you have this snippet in the Dockerfile:

```
...
CMD ping 8.8.8.8 -c 5
```

then Docker will execute the following command when a container from this image is started:

```
sh -c "ping 8.8.8.8 -c 5"
```

This has the same effect as executing the `ping` command directly. On a more detailed level, `sh` will be the main process that is started inside the container, and the `ping` command will be spawned in a child process.

the command with the `CMD` keyword. Thus, if you have the following typical Linux expression

```
ping 8.8.8.8 -c 5
```

then `ping` is the command, and `8.8.8.8 -c 5` are the parameters to this command. Consequently, one would define this in a Dockerfile as follows:

```
...
ENTRYPOINT ping
CMD 8.8.8.8 -c
```

Another example could be the following expression that downloads a script from a given URL:

```
wget -O - https://www.acme.com/cool-tool/install.sh
```

In this case, `wget` is the command and

```
-O - https://www.acme.com/cool-tool/install.sh
```

are the parameters to that command. Consequently, the Dockerfile would look like the following:

```
...
ENTRYPOINT wget
CMD -O - https://www.acme.com/cool-tool/install.sh
```

If it is not defined explicitly, that is, if `ENTRYPOINT` is omitted in the Dockerfile, its default value is `/bin/sh -c`.

Thus, if you only define a value for `CMD` like:

```
CMD <some value>
```

then the resulting container start command will be

```
/bin/sh -c "<some value>"
```

which is a valid Linux command and will just use the Bourne shell to execute whatever command you defined for `<some value>`.

Thus, if you have this snippet in the Dockerfile:

```
...
CMD ping 8.8.8.8 -c 5
```

then Docker will execute the following command when a container from this image is started:

```
sh -c "ping 8.8.8.8 -c 5"
```

This has the same effect as executing the `ping` command directly. On a more detailed level, `sh` will be the main process that is started inside the container, and the `ping` command will be spawned in a child process.

An alternative form of defining the values used with **ENTRYPOINT** and **CMD** is to format them as JSON arrays of strings. Have a look at this sample Dockerfile to understand what we mean by it:

```
FROM ubuntu:20.04
RUN apt-get update && apt-get install -y iputils-ping
ENTRYPOINT ["ping"]
CMD ["8.8.8.8","-c", "5"]
```

The individual items in the JSON array correspond to the tokens of the ping expression. This is the preferred form of defining **ENTRYPOINT** and **CMD**. It is called the *exec* form, whereas the way we used to define the expression before is called the *shell* form. Use the exec form of ENTRYPOINT to set stable default commands and arguments. Then use either form of CMD to set additional defaults that are more likely to be changed during run time.

If we build an image called **pinger** from the preceding Dockerfile with this command:

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

we can then run a container from that image with this command:

```
docker container run pinger
```

and we should see this:

```
$ docker container run pinger
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=110 time=2.05 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=110 time=1.62 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=110 time=1.43 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=110 time=1.35 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=110 time=1.59 ms

--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4004ms
rtt min/avg/max/mdev = 1.348/1.607/2.054/0.245 ms
```

Figure 5.11: Running the pinger container as is

It is now easy to override the values defined in the **CMD** instruction by just passing a new value in the run command. For example, if we try this:

```
docker container run pinger 8.8.4.4 -c 3
```

An alternative form of defining the values used with **ENTRYPOINT** and **CMD** is to format them as JSON arrays of strings. Have a look at this sample Dockerfile to understand what we mean by it:

```
FROM ubuntu:20.04
RUN apt-get update && apt-get install -y iputils-ping
ENTRYPOINT ["ping"]
CMD ["8.8.8.8","-c", "5"]
```

The individual items in the JSON array correspond to the tokens of the ping expression. This is the preferred form of defining **ENTRYPOINT** and **CMD**. It is called the *exec* form, whereas the way we used to define the expression before is called the *shell* form. Use the exec form of ENTRYPOINT to set stable default commands and arguments. Then use either form of CMD to set additional defaults that are more likely to be changed during run time.

If we build an image called **pinger** from the preceding Dockerfile with this command:

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

we can then run a container from that image with this command:

```
docker container run pinger
```

and we should see this:

```
$ docker container run pinger
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=110 time=2.05 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=110 time=1.62 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=110 time=1.43 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=110 time=1.35 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=110 time=1.59 ms

--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4004ms
rtt min/avg/max/mdev = 1.348/1.607/2.054/0.245 ms
```

Figure 5.11: Running the pinger container as is

It is now easy to override the values defined in the **CMD** instruction by just passing a new value in the run command. For example, if we try this:

```
docker container run pinger 8.8.4.4 -c 3
```

we will see this:

```
$ docker container run pinger 8.8.4.4 -c 3
PING 8.8.4.4 (8.8.4.4) 56(84) bytes of data.
64 bytes from 8.8.4.4: icmp_seq=1 ttl=111 time=1.53 ms
64 bytes from 8.8.4.4: icmp_seq=2 ttl=111 time=1.48 ms
64 bytes from 8.8.4.4: icmp_seq=3 ttl=111 time=1.50 ms

--- 8.8.4.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2006ms
rtt min/avg/max/mdev = 1.483/1.502/1.527/0.018 ms
```

Figure 5.12: Running the pinger container with custom parameters

As you can see, the container now pings the address passed, and it pings it three instead of five times until it stops.

Finally, if we want to override what is defined in the **ENTRYPOINT**, we can do so in the **docker run** command by explicitly overriding it with the parameter **--entrypoint**.

The following example shows how:

```
docker container run --entrypoint ls pinger -a -l /var
```

This command will run the pinger container but execute the following command as a start command:

```
ls -a -l /var
```

as you can see in the following image:

```
$ docker container run --entrypoint ls pinger -a -l /var
/var:
.
..
backups
cache
lib
local
lock
```

Figure 5.13: Overriding the ENTRYPOINT in the run command

Note that the **--entrypoint** override can only set the binary to execute in the start command.

The EXPOSE keyword

The **EXPOSE** keyword is used to tell the Docker builder that our application running inside the container will be listening on the port that we declare, for example:

```
EXPOSE 3000
```

we will see this:

```
$ docker container run pinger 8.8.4.4 -c 3
PING 8.8.4.4 (8.8.4.4) 56(84) bytes of data.
64 bytes from 8.8.4.4: icmp_seq=1 ttl=111 time=1.53 ms
64 bytes from 8.8.4.4: icmp_seq=2 ttl=111 time=1.48 ms
64 bytes from 8.8.4.4: icmp_seq=3 ttl=111 time=1.50 ms

--- 8.8.4.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2006ms
rtt min/avg/max/mdev = 1.483/1.502/1.527/0.018 ms
```

Figure 5.12: Running the pinger container with custom parameters

As you can see, the container now pings the address passed, and it pings it three instead of five times until it stops.

Finally, if we want to override what is defined in the **ENTRYPOINT**, we can do so in the **docker run** command by explicitly overriding it with the parameter **--entrypoint**.

The following example shows how:

```
docker container run --entrypoint ls pinger -a -l /var
```

This command will run the pinger container but execute the following command as a start command:

```
ls -a -l /var
```

as you can see in the following image:

```
$ docker container run --entrypoint ls pinger -a -l /var
/var:
.
..
backups
cache
lib
local
lock
```

Figure 5.13: Overriding the ENTRYPOINT in the run command

Note that the **--entrypoint** override can only set the binary to execute in the start command.

The EXPOSE keyword

The **EXPOSE** keyword is used to tell the Docker builder that our application running inside the container will be listening on the port that we declare, for example:

```
EXPOSE 3000
```

is telling us that the application is listening at port `3000`. This could be a self-hosted Node JS, Python, Java, or .NET application that we containerize. If we then use the `-P` parameter in the Docker run command, Docker will map that port to a random free port on the host:

```
docker container run -P --detach my-web-app
```

We will explain this concept in more detail in *Chapter 6, Demystifying container networking*.

Note that we can expose multiple ports by repeating the EXPOSE statement in the Dockerfile with the different ports.

Multi-step Dockerfiles

It is a recommended best practice to keep a custom Docker image as small as possible. You should only package into a container whatever is absolutely needed to run your application. Everything else should be avoided. Thus, do not leave source code or any unused libraries or frameworks inside the container. Do not leave SDKs with their compilers and debuggers inside the container. Your target image should contain the application and the minimal infrastructure to make it run.

For this reason, Docker has introduced multi-step builds. We will show this using an example. We will write a hello world type minimal application in Go and containerize it with the following steps. We will then show the result when not using a multi-step build and when using this technique.

1. Open a new terminal and navigate to your `src` folder.
2. From within the `src` folder, create a new folder for our Go application and navigate to it:

   ```
   mkdir -p ch05/multi-step-go && cd ch05/multi-step-go
   ```

3. Open VS Code from inside this folder:

   ```
   code.
   ```

4. Add a file called `main.go` with this content:

   ```
   package main
   import "fmt"
   func main() {
       fmt.Println("hello world")
   }
   ```

is telling us that the application is listening at port `3000`. This could be a self-hosted Node JS, Python, Java, or .NET application that we containerize. If we then use the `-P` parameter in the Docker run command, Docker will map that port to a random free port on the host:

```
docker container run -P --detach my-web-app
```

We will explain this concept in more detail in *Chapter 6, Demystifying container networking*.

Note that we can expose multiple ports by repeating the EXPOSE statement in the Dockerfile with the different ports.

Multi-step Dockerfiles

It is a recommended best practice to keep a custom Docker image as small as possible. You should only package into a container whatever is absolutely needed to run your application. Everything else should be avoided. Thus, do not leave source code or any unused libraries or frameworks inside the container. Do not leave SDKs with their compilers and debuggers inside the container. Your target image should contain the application and the minimal infrastructure to make it run.

For this reason, Docker has introduced multi-step builds. We will show this using an example. We will write a hello world type minimal application in Go and containerize it with the following steps. We will then show the result when not using a multi-step build and when using this technique.

1. Open a new terminal and navigate to your `src` folder.
2. From within the `src` folder, create a new folder for our Go application and navigate to it:

   ```
   mkdir -p ch05/multi-step-go && cd ch05/multi-step-go
   ```

3. Open VS Code from inside this folder:

   ```
   code.
   ```

4. Add a file called `main.go` with this content:

   ```
   package main
   import "fmt"
   func main() {
       fmt.Println("hello world")
   }
   ```

5. Now, add a file called **go.mod** to the application folder and add this content:

```
module main
go 1.16
```

6. Finally, add a file called **Dockerfile** to the application folder with this content:

```
FROM golang:1.16
RUN mkdir /app
COPY . /app
WORKDIR /app
RUN go build -o main .
CMD ["/app/main"]
```

7. Now from within your console, build the Docker image with the following command:

```
docker image build -t hello-go .
```

8. Run a container from the newly built image to make sure the app works:

```
docker container run hello-go
```

and you should get the output:

```
hello world
```

9. List the newly built image and observe its size:

```
docker image ls | grep hello-go
```

you should see something similar to this:

```
gabriel@solaris ~ % docker image ls | grep hello-go
hello-go              latest    da4d7f65b13b   4 minutes ago       730MB
```

Figure 5.14: Size of not optimized the CO hello world application image

As we can see, the generated Docker image is quite big, 730 MB to be precise, although the application that we containerized is extremely simple. That is a problem, as you will certainly agree. But we can do better, as follows:

1. In VS Code, create another file called **Dockerfile.optimized** and add the following content to it:

```
FROM golang:1.16 as builder
RUN mkdir /app
COPY . /app
WORKDIR /app
RUN go build -o main .

FROM alpine:latest as production
```

5. Now, add a file called `go.mod` to the application folder and add this content:

```
module main
go 1.16
```

6. Finally, add a file called `Dockerfile` to the application folder with this content:

```
FROM golang:1.16
RUN mkdir /app
COPY . /app
WORKDIR /app
RUN go build -o main .
CMD ["/app/main"]
```

7. Now from within your console, build the Docker image with the following command:

```
docker image build -t hello-go .
```

8. Run a container from the newly built image to make sure the app works:

```
docker container run hello-go
```

and you should get the output:

```
hello world
```

9. List the newly built image and observe its size:

```
docker image ls | grep hello-go
```

you should see something similar to this:

```
gabriel@solaris ~ % docker image ls | grep hello-go
hello-go              latest    da4d7f65b13b   4 minutes ago       730MB
```

Figure 5.14: Size of not optimized the CO hello world application image

As we can see, the generated Docker image is quite big, 730 MB to be precise, although the application that we containerized is extremely simple. That is a problem, as you will certainly agree. But we can do better, as follows:

1. In VS Code, create another file called `Dockerfile.optimized` and add the following content to it:

```
FROM golang:1.16 as builder
RUN mkdir /app
COPY . /app
WORKDIR /app
RUN go build -o main .

FROM alpine:latest as production
```

```
COPY --from=builder /app .
CMD ["./main"]
```

2. Use the preceding Dockerfile to build an optimized image using this command:

   ```
   docker image build -f Dockerfile.optimized -t hello-go .
   ```

   Note that we are using the `-f` parameter in the preceding command to tell the builder not to look for the default Dockerfile but to look for a custom Dockerfile called `Dockerfile.optimized`.

3. Once again, list the built images and compare their sizes:

   ```
   docker image ls | grep hello-go
   ```

   you should see something similar to this:

```
gabriel@Solaris ~ % docker image ls | grep hello-go
hello-go-optimized      latest    de09cb8198f1   21 seconds ago      7.27MB
hello-go                latest    da4d7f65b13b   10 minutes ago      730MB
```

***Figure 5.15:** Sizes of the GO hello world application images*

When analyzing the `Dockerfile.optimized` from Step 1, you will notice that we now have two stages, each one starting with a `FROM` statement. The first stage is called `builder`, and the second one is called `production`. In the first stage, we use the `golang` image that contains the full Golang SDK to build the application. Then, in the second stage, we use a minimal base image, in this case, the `alpine` image. We then copy the compiled Go application from the builder into the target image using the following:

```
COPY --from=builder /app .
```

or more general:

```
COPY --from=<alias> <source> <target>
```

Where `<alias>` is the name or alias of the stage from which we want to copy and `<source>` denotes the source files and folders to copy, and `<target>` denotes their destination in the filesystem of the destination image. In the preceding example, we are copying the folder `/app` with all its content to the current working folder of the target image's filesystem.

That is all. The target `alpine` image is small, and the compiled application is also small. We do not have the compiler or any other pieces from the SDK in the target image.

```
COPY --from=builder /app .
CMD ["./main"]
```

2. Use the preceding Dockerfile to build an optimized image using this command:

   ```
   docker image build -f Dockerfile.optimized -t hello-go .
   ```

   Note that we are using the `-f` parameter in the preceding command to tell the builder not to look for the default Dockerfile but to look for a custom Dockerfile called `Dockerfile.optimized`.

3. Once again, list the built images and compare their sizes:

   ```
   docker image ls | grep hello-go
   ```

   you should see something similar to this:

```
gabriel@Solaris ~ % docker image ls | grep hello-go
hello-go-optimized      latest    de09cb8198f1   21 seconds ago      7.27MB
hello-go                latest    da4d7f65b13b   10 minutes ago      730MB
```

***Figure 5.15:** Sizes of the GO hello world application images*

When analyzing the `Dockerfile.optimized` from Step 1, you will notice that we now have two stages, each one starting with a `FROM` statement. The first stage is called `builder`, and the second one is called `production`. In the first stage, we use the `golang` image that contains the full Golang SDK to build the application. Then, in the second stage, we use a minimal base image, in this case, the `alpine` image. We then copy the compiled Go application from the builder into the target image using the following:

```
COPY --from=builder /app .
```

or more general:

```
COPY --from=<alias> <source> <target>
```

Where `<alias>` is the name or alias of the stage from which we want to copy and `<source>` denotes the source files and folders to copy, and `<target>` denotes their destination in the filesystem of the destination image. In the preceding example, we are copying the folder `/app` with all its content to the current working folder of the target image's filesystem.

That is all. The target `alpine` image is small, and the compiled application is also small. We do not have the compiler or any other pieces from the SDK in the target image.

When looking at *figure 5.15*, you will notice that the optimized image is significantly smaller, which is a good thing. In fact, the optimized size is nearly 100 times smaller than the original image.

With this, the attack surface of the image is decreased, the required disk space to store the image is reduced, and the necessary bandwidth to upload the image to and download from a container registry is also reduced. Finally, since the image is so small, the load time of a container is also dramatically reduced. You see, a lot of benefits.

Dockerfile best practices

Now that you have learned a lot about authoring Dockerfiles, we want to add a few recommendations to your tool belt. The following best practices have shown to be very helpful:

1. Ordering of the instructions inside a Dockerfile matters. Only with the correct ordering can we maximize the use of caching. This particularly matters during development, where the same image might be built over and over again. We want to keep the time needed to build an image at a minimum. We will talk more about this in *Chapter 8, Testing and debugging containerized apps*.
2. Use multi-step builds to streamline your build process and, at the same time, keep the final image as small as possible.
3. Containers are supposed to be ephemeral. That is, a container can be stopped or killed at any time without further notice. And in its place, a new instance can be started from the same container image. As a consequence, we should make sure that the start-up process of a container does not take a long time. Or with other words, the time needed to initialize the application running inside a container should be minimal.

So, you have learned a lot about how to author a Dockerfile. This is the most popular method used to create a customer Docker image. But now let us have a look at the third and last method, importing an image from a file.

Importing a Docker image

This technique is particularly useful if you want to have a Docker image on your system, but cannot access it from Docker Hub—or any other container registry—due to say security reasons. In such a case, you can pull the image to one system, export it from that system as a tarball (a file containing the compressed content of the image), and then import the image into the destination system using the following command:

```
docker image import <path-to-tarball> <image-name>
```

When looking at *figure 5.15*, you will notice that the optimized image is significantly smaller, which is a good thing. In fact, the optimized size is nearly 100 times smaller than the original image.

With this, the attack surface of the image is decreased, the required disk space to store the image is reduced, and the necessary bandwidth to upload the image to and download from a container registry is also reduced. Finally, since the image is so small, the load time of a container is also dramatically reduced. You see, a lot of benefits.

Dockerfile best practices

Now that you have learned a lot about authoring Dockerfiles, we want to add a few recommendations to your tool belt. The following best practices have shown to be very helpful:

1. Ordering of the instructions inside a Dockerfile matters. Only with the correct ordering can we maximize the use of caching. This particularly matters during development, where the same image might be built over and over again. We want to keep the time needed to build an image at a minimum. We will talk more about this in *Chapter 8, Testing and debugging containerized apps*.
2. Use multi-step builds to streamline your build process and, at the same time, keep the final image as small as possible.
3. Containers are supposed to be ephemeral. That is, a container can be stopped or killed at any time without further notice. And in its place, a new instance can be started from the same container image. As a consequence, we should make sure that the start-up process of a container does not take a long time. Or with other words, the time needed to initialize the application running inside a container should be minimal.

So, you have learned a lot about how to author a Dockerfile. This is the most popular method used to create a customer Docker image. But now let us have a look at the third and last method, importing an image from a file.

Importing a Docker image

This technique is particularly useful if you want to have a Docker image on your system, but cannot access it from Docker Hub—or any other container registry—due to say security reasons. In such a case, you can pull the image to one system, export it from that system as a tarball (a file containing the compressed content of the image), and then import the image into the destination system using the following command:

```
docker image import <path-to-tarball> <image-name>
```

where **<path-to-tarball>** corresponds to the file name (including path) of the tarball you want to import, and **<image-name>** is the name of the target image that will be created on your system. Thus, the following command:

```
docker image import ./my-exported-image.tar my-image:1.0
```

will look for a tar file called **my-exported-image.tar** at the current location and import it as a new image and name it **my-image:1.0**. Note that **1.0** is the tag of the image.

Let us make an example. We will pull the **postgres:alpine** image, export it and then re-import it with a different name, following the given steps:

1. Pull the Docker image:

   ```
   docker image pull postgres:alpine
   ```

2. Export the image to a tarball called **my-postgres.tar:**

   ```
   docker image save postgres:alpine -o my-postgres.tar
   ```

3. To double-check that the export has succeeded, list the folder content and grep for the tar file:

   ```
   ls -al | grep my-postgres
   ```

4. Load the tarball into the system as a new image with the name **my-postgres:latest:**

   ```
   docker image import ./my-postgres.tar my-postgres:latest
   ```

The result of the preceding commands should look like the following:

```
gabriel@Solaris ~ % docker image pull postgres:alpine
alpine: Pulling from library/postgres
58ab47519297: Already exists
014ca0527e17: Pull complete
ee9ed2adb5ac: Pull complete
112d0ed39e19: Pull complete
298ff48a3e3e: Pull complete
262ec7068fec: Pull complete
bcb045f6e631: Pull complete
56bca726d175: Pull complete
Digest: sha256:ff384947eb9f5939b7fc5ef2ce620fad088999590973f05e6812037d163c770e
Status: Downloaded newer image for postgres:alpine
docker.io/library/postgres:alpine
gabriel@Solaris ~ % docker image save postgres:alpine -o my-postgres.tar
gabriel@Solaris ~ % ls -al | grep my-postgres
-rw-------   1 gabriel  staff  186420224  3 Jul 16:04 my-postgres.tar
gabriel@Solaris ~ % docker image import ./my-postgres.tar my-postgres:latest
sha256:2b0927aea078d0d1bb030af87c571ee99eb03b94cec027e84058c6f95aa00a06
gabriel@Solaris ~ %
```

Figure 5.16: Exporting and importing a Docker image

where **<path-to-tarball>** corresponds to the file name (including path) of the tarball you want to import, and **<image-name>** is the name of the target image that will be created on your system. Thus, the following command:

```
docker image import ./my-exported-image.tar my-image:1.0
```

will look for a tar file called **my-exported-image.tar** at the current location and import it as a new image and name it **my-image:1.0**. Note that **1.0** is the tag of the image.

Let us make an example. We will pull the **postgres:alpine** image, export it and then re-import it with a different name, following the given steps:

1. Pull the Docker image:

   ```
   docker image pull postgres:alpine
   ```

2. Export the image to a tarball called **my-postgres.tar:**

   ```
   docker image save postgres:alpine -o my-postgres.tar
   ```

3. To double-check that the export has succeeded, list the folder content and grep for the tar file:

   ```
   ls -al | grep my-postgres
   ```

4. Load the tarball into the system as a new image with the name **my-postgres:latest:**

   ```
   docker image import ./my-postgres.tar my-postgres:latest
   ```

The result of the preceding commands should look like the following:

```
gabriel@Solaris ~ % docker image pull postgres:alpine
alpine: Pulling from library/postgres
58ab47519297: Already exists
014ca0527e17: Pull complete
ee9ed2adb5ac: Pull complete
112d0ed39e19: Pull complete
298ff48a3e3e: Pull complete
262ec7068fec: Pull complete
bcb045f6e631: Pull complete
56bca726d175: Pull complete
Digest: sha256:ff384947eb9f5939b7fc5ef2ce620fad088999590973f05e6812037d163c770e
Status: Downloaded newer image for postgres:alpine
docker.io/library/postgres:alpine
gabriel@Solaris ~ % docker image save postgres:alpine -o my-postgres.tar
gabriel@Solaris ~ % ls -al | grep my-postgres
-rw-------   1 gabriel  staff  186420224  3 Jul 16:04 my-postgres.tar
gabriel@Solaris ~ % docker image import ./my-postgres.tar my-postgres:latest
sha256:2b0927aea078d0d1bb030af87c571ee99eb03b94cec027e84058c6f95aa00a06
gabriel@Solaris ~ %
```

Figure 5.16: Exporting and importing a Docker image

Building a Docker image

In the preceding sections, we learned already quite a bit about how a Docker image is built by the builder. Let us reinforce what we have learned so far by looking more closely at the output produced by the Docker builder during the build process. As an example, we want to use the Dockerfile that we were using in the previous chapter when building the Node JS sample client. The content of that Dockerfile looks like the following:

```
FROM node:latest
WORKDIR /app
COPY package.json ./
RUN npm install
COPY . .
CMD node server.js
```

Now to build the custom image based on the preceding Dockerfile, proceed as follows:

1. Navigate to the folder containing the code for the Node client

   ```
   cd src/ch04/node-client
   ```

2. Build the Docker image for the Node client:

   ```
   DOCKER_BUILDKIT=0 docker image build --no-cache -t node-client .
   ```

   Note we use the `--no-cache` parameter to make sure the Docker builder uses no caching and, for example, pulls the base image from Docker Hub even if it is already in the local cache. This is only relevant for this exercise, and you should normally not have to use it. We also prefix the Docker build command with `DOCKER_BUILDKIT=0` to make sure we do not accidentally use the new build kit to build the container. More about the build kit in the next section.

3. Observe the output generated in the terminal window. We start with Step 1 of 6:

```
Sending build context to Docker daemon  826.4kB
Step 1/6 : FROM node:latest
latest: Pulling from library/node
b721438f56fc: Pull complete
268ad7651993: Pull complete
2c93fd903c94: Pull complete
9f0eab8670a9: Pull complete
510a11c58681: Pull complete
ac59cf613e61: Pull complete
99d15f46a1cf: Pull complete
e42558b93d09: Pull complete
a98d976752e4: Pull complete
Digest: sha256:987be0a7ca165b6b19bf77ca812e980f603a10c1000bf25095cd89e2b146f9ad
Status: Downloaded newer image for node:latest
 ---> b406d3362cac
```

Figure 5.17: Step 1/6 of the build process

Building a Docker image

In the preceding sections, we learned already quite a bit about how a Docker image is built by the builder. Let us reinforce what we have learned so far by looking more closely at the output produced by the Docker builder during the build process. As an example, we want to use the Dockerfile that we were using in the previous chapter when building the Node JS sample client. The content of that Dockerfile looks like the following:

```
FROM node:latest
WORKDIR /app
COPY package.json ./
RUN npm install
COPY . .
CMD node server.js
```

Now to build the custom image based on the preceding Dockerfile, proceed as follows:

1. Navigate to the folder containing the code for the Node client

   ```
   cd src/ch04/node-client
   ```

2. Build the Docker image for the Node client:

   ```
   DOCKER_BUILDKIT=0 docker image build --no-cache -t node-client .
   ```

   Note we use the `--no-cache` parameter to make sure the Docker builder uses no caching and, for example, pulls the base image from Docker Hub even if it is already in the local cache. This is only relevant for this exercise, and you should normally not have to use it. We also prefix the Docker build command with `DOCKER_BUILDKIT=0` to make sure we do not accidentally use the new build kit to build the container. More about the build kit in the next section.

3. Observe the output generated in the terminal window. We start with Step 1 of 6:

```
Sending build context to Docker daemon  826.4kB
Step 1/6 : FROM node:latest
latest: Pulling from library/node
b721438f56fc: Pull complete
268ad7651993: Pull complete
2c93fd903c94: Pull complete
9f0eab8670a9: Pull complete
510a11c58681: Pull complete
ac59cf613e61: Pull complete
99d15f46a1cf: Pull complete
e42558b93d09: Pull complete
a98d976752e4: Pull complete
Digest: sha256:987be0a7ca165b6b19bf77ca812e980f603a10c1000bf25095cd89e2b146f9ad
Status: Downloaded newer image for node:latest
 ---> b406d3362cac
```

Figure 5.17: Step 1/6 of the build process

Docker recognized that the `node:latest` image used as the base image was not yet in the local image cache and thus automatically downloaded it. The last line shows the ID (`b406d3362cac`) that Docker assigned to this image. This can be verified by using the following command:

```
gabriel@Solaris node-client % docker image ls | grep b406d3362cac
node                                latest        b406d3362cac   2 days ago       853MB
```

Figure 5.18: Verifying the image ID of the downloaded `node:Latest` image

And indeed, we find that the downloaded Node image has this ID.

4. Now, we look at Step 2 of 6:

```
Step 2/6 : WORKDIR /app
 ---> Running in f9b86031e871
Removing intermediate container f9b86031e871
 ---> 51f89f46051d
```

Figure 5.19: Step 2/6 of the build process

Here, we see on the second line that the builder is running a (temporary) container with the id `f9b86031e871` based on the image from the last step and then applying the `WORKDIR` command inside it. In Line 3, the builder tells us that it removed the temporary container (the builder had persisted with the new container layer prior to that). The resulting image has the ID `51f89f46051d`, as stated in the last line. This new image with ID `51f89f46051d` has no name, and if we want to create a container from it (as the builder will do in the next step), we had to use a Docker command similar to this: `docker container run ... 51f89f46051d ...`

5. Next is Step 3 of 6:

```
Step 3/6 : COPY package.json ./
 ---> 9eacade1a1d0
```

Figure 5.20: Step 3/6 of the build process

Here the file `package.json` is copied into the (temporary) container, and a new image layer is then created from this container layer. The resulting image with the additional layer has `ID=9eacade1a1d0`.

Docker recognized that the `node:latest` image used as the base image was not yet in the local image cache and thus automatically downloaded it. The last line shows the ID (`b406d3362cac`) that Docker assigned to this image. This can be verified by using the following command:

```
gabriel@Solaris node-client % docker image ls | grep b406d3362cac
node                                latest        b406d3362cac   2 days ago       853MB
```

Figure 5.18: Verifying the image ID of the downloaded `node:Latest` image

And indeed, we find that the downloaded Node image has this ID.

4. Now, we look at Step 2 of 6:

```
Step 2/6 : WORKDIR /app
 ---> Running in f9b86031e871
Removing intermediate container f9b86031e871
 ---> 51f89f46051d
```

Figure 5.19: Step 2/6 of the build process

Here, we see on the second line that the builder is running a (temporary) container with the id `f9b86031e871` based on the image from the last step and then applying the `WORKDIR` command inside it. In Line 3, the builder tells us that it removed the temporary container (the builder had persisted with the new container layer prior to that). The resulting image has the ID `51f89f46051d`, as stated in the last line. This new image with ID `51f89f46051d` has no name, and if we want to create a container from it (as the builder will do in the next step), we had to use a Docker command similar to this: `docker container run ... 51f89f46051d ...`

5. Next is Step 3 of 6:

```
Step 3/6 : COPY package.json ./
 ---> 9eacade1a1d0
```

Figure 5.20: Step 3/6 of the build process

Here the file `package.json` is copied into the (temporary) container, and a new image layer is then created from this container layer. The resulting image with the additional layer has `ID=9eacade1a1d0`.

6. Then, comes Step 4 of 6:

```
Step 4/6 : RUN npm install
 ---> Running in 56e16c79855b

added 20 packages, and audited 21 packages in 4s

1 package is looking for funding
  run `npm fund` for details
npm notice
npm notice New patch version of npm available! 7.21.0 -> 7.21.1
npm notice Changelog: <https://github.com/npm/cli/releases/tag/v7.21.1>
npm notice Run `npm install -g npm@7.21.1` to update!
npm notice

found 0 vulnerabilities
Removing intermediate container 56e16c79855b
 ---> 888f9df3fff8
```

Figure 5.21: Step 4/6 of the build process

Here, the builder runs the **npm install** command inside the temporary container with ID **56e16c79855b**. We can see in the output preceding that the command installs 20 packages. Most of the preceding output is actually created by the **npm** tool while it is running. To be precise, all than the first and the last two lines. The resulting image of this build step has **ID=888f9df3fff8**.

7. Now on to Step 5 of 6:

```
Step 5/6 : COPY . .
 ---> eda2164f0127
```

Figure 5.22: Step 5/6 of the build process

This build step is once again a simple one. All files from the local host folder are copied into the working directory inside the container. The result of the build step is an image with **ID=eda2164f0127**.

8. And finally, Step 6 of 6:

```
Step 6/6 : CMD node server.js
 ---> Running in af8dff97f311
Removing intermediate container af8dff97f311
 ---> dccdad14088b
Successfully built dccdad14088b
Successfully tagged node-client:latest
```

Figure 5.23: Step 6/6 of the build process

6. Then, comes Step 4 of 6:

```
Step 4/6 : RUN npm install
 ---> Running in 56e16c79855b

added 20 packages, and audited 21 packages in 4s

1 package is looking for funding
  run `npm fund` for details
npm notice
npm notice New patch version of npm available! 7.21.0 -> 7.21.1
npm notice Changelog: <https://github.com/npm/cli/releases/tag/v7.21.1>
npm notice Run `npm install -g npm@7.21.1` to update!
npm notice

found 0 vulnerabilities
Removing intermediate container 56e16c79855b
 ---> 888f9df3fff8
```

Figure 5.21: Step 4/6 of the build process

Here, the builder runs the **npm install** command inside the temporary container with ID **56e16c79855b**. We can see in the output preceding that the command installs 20 packages. Most of the preceding output is actually created by the **npm** tool while it is running. To be precise, all than the first and the last two lines. The resulting image of this build step has **ID=888f9df3fff8**.

7. Now on to Step 5 of 6:

```
Step 5/6 : COPY . .
 ---> eda2164f0127
```

Figure 5.22: Step 5/6 of the build process

This build step is once again a simple one. All files from the local host folder are copied into the working directory inside the container. The result of the build step is an image with **ID=eda2164f0127**.

8. And finally, Step 6 of 6:

```
Step 6/6 : CMD node server.js
 ---> Running in af8dff97f311
Removing intermediate container af8dff97f311
 ---> dccdad14088b
Successfully built dccdad14088b
Successfully tagged node-client:latest
```

Figure 5.23: Step 6/6 of the build process

The final step defines the start command of the container. The resulting image did receive the image ID=dccdad14088b, as we can easily verify with this command:

```
gabriel@Solaris node-client % docker image ls | grep dccdad14088b
node-client                    latest       dccdad14088b   29 minutes ago   857MB
```

Figure 5.24: Identifying the resulting image by its ID

Now that we understand how the default Docker image builder works let us quickly discuss the more recent and also more powerful builder that is based on the so-called build kit.

Leveraging the Docker Build Kit

The default Docker builder has served us well over all those years, but it shows its age. The Docker team has built a new and highly optimized builder that is part of the Docker setup. But the new builder is turned off by default. You can turn it on by defining the following environment variable:

```
export DOCKER_BUILDKIT=1
```

Once you have done so, all subsequent build operations should be executed much faster. The gain is greater the bigger the complexity of your build process is.

If you want to permanently use the new build kit, then you should add the preceding expression to your profile, for example, by adding it to the file `.bash_profile`.

Running the build command for the Node client application we used in the previous section with the build kit-based image builder produces the following output from the author's computer:

```
[+] Building 6.9s (11/11) FINISHED
 => [internal] load build definition from Dockerfile                          0.0s
 => => transferring dockerfile: 564B                                          0.0s
 => [internal] load .dockerignore                                             0.0s
 => => transferring context: 2B                                               0.0s
 => [internal] load metadata for docker.io/library/node:latest                2.5s
 => [auth] library/node:pull token for registry-1.docker.io                   0.0s
 => [1/5] FROM docker.io/library/node:latest@sha256:987be0a7ca165b6b19bf77ca812e980f603a10c1000bf250  0.1s
 => => resolve docker.io/library/node:latest@sha256:987be0a7ca165b6b19bf77ca812e980f603a10c1000bf250  0.0s
 => [internal] load build context                                             0.1s
 => => transferring context: 762.87kB                                         0.1s
 => [2/5] WORKDIR /app                                                        0.0s
 => [3/5] COPY package.json ./                                                0.0s
 => [4/5] RUN npm install                                                     4.0s
 => [5/5] COPY . .                                                            0.0s
 => exporting to image                                                        0.1s
 => => exporting layers                                                       0.1s
 => => writing image sha256:767765114585959f6713da60f67d6493687287f3b79013d6504ldffe9bd8ee1334  0.0s
 => => naming to docker.io/library/node-client                                0.0s
```

Figure 5.25: Output of build kit-based builder

The final step defines the start command of the container. The resulting image did receive the image ID=dccdad14088b, as we can easily verify with this command:

```
gabriel@Solaris node-client % docker image ls | grep dccdad14088b
node-client                    latest       dccdad14088b   29 minutes ago   857MB
```

Figure 5.24: Identifying the resulting image by its ID

Now that we understand how the default Docker image builder works let us quickly discuss the more recent and also more powerful builder that is based on the so-called build kit.

Leveraging the Docker Build Kit

The default Docker builder has served us well over all those years, but it shows its age. The Docker team has built a new and highly optimized builder that is part of the Docker setup. But the new builder is turned off by default. You can turn it on by defining the following environment variable:

```
export DOCKER_BUILDKIT=1
```

Once you have done so, all subsequent build operations should be executed much faster. The gain is greater the bigger the complexity of your build process is.

If you want to permanently use the new build kit, then you should add the preceding expression to your profile, for example, by adding it to the file `.bash_profile`.

Running the build command for the Node client application we used in the previous section with the build kit-based image builder produces the following output from the author's computer:

```
[+] Building 6.9s (11/11) FINISHED
 => [internal] load build definition from Dockerfile                          0.0s
 => => transferring dockerfile: 564B                                          0.0s
 => [internal] load .dockerignore                                             0.0s
 => => transferring context: 2B                                               0.0s
 => [internal] load metadata for docker.io/library/node:latest                2.5s
 => [auth] library/node:pull token for registry-1.docker.io                   0.0s
 => [1/5] FROM docker.io/library/node:latest@sha256:987be0a7ca165b6b19bf77ca812e980f603a10c1000bf250  0.1s
 => => resolve docker.io/library/node:latest@sha256:987be0a7ca165b6b19bf77ca812e980f603a10c1000bf250  0.0s
 => [internal] load build context                                             0.1s
 => => transferring context: 762.87kB                                         0.1s
 => [2/5] WORKDIR /app                                                        0.0s
 => [3/5] COPY package.json ./                                                0.0s
 => [4/5] RUN npm install                                                     4.0s
 => [5/5] COPY . .                                                            0.0s
 => exporting to image                                                        0.1s
 => => exporting layers                                                       0.1s
 => => writing image sha256:767765114585959f6713da60f67d6493687287f3b79013d6504ldffe9bd8ee1334  0.0s
 => => naming to docker.io/library/node-client                                0.0s
```

Figure 5.25: Output of build kit-based builder

The new builder is much faster that the old builder. This is specifically true for complex Dockerfiles. The builder parallelizes tasks as much as possible. The output generated also includes the timing of individual steps, as one can see in the preceding image. It is highly recommended that you enable the new builder by default.

Tagging Docker images

Now that we know how to author a Dockerfile and build an image using this Dockerfile, we have to understand a bit more about the tags used for container images. The fully qualified name of a container image is composed of the following parts:

```
<registry>/<company or user>/<repository name>:<tag>
```

where the individual parts have the following meaning:

<registry>	This is the URL to the image registry where the Docker image is hosted. By default, and if not specified explicitly, this is Docker Hub at https://hub.Docker.com
<organization or user>	This is the name of an organization (company), e.g., *microsoft*, *oracle*, *google*, or the name of a registered user such as *gnschenker*, which is the author's account name on Docker Hub.
<repository name>	This is the name of a Docker image. It is also called a repository, and multiple versions of the image can exist at the same time in a single repository. Examples are alpine, centos, python, or node.
<tag>	A tag can be used to describe a particular version of a Docker image. For example, there exist multiple versions of the ubuntu image on Docker Hub, such as 21.04, 20.04, 16.04, and so on. There is also a special tag called *latest*. It is usually used to designate the latest version of an image. If one omits the tag in Docker command, then the Docker CLI will automatically assume that it is *latest*.

Let us look at a few sample image names:

- **`alpine:`**

  This is equivalent to **`alpine:latest`**. In this example, we have only the **`<repository name>`** part defined. Thus, the Docker CLI assumes that the registry is the default registry, namely, Docker Hub. Note that the default registry can be changed in the configuration settings for the Docker CLI. In the preceding image name, we do also not define a value for **`<organization or user>`** part, which indicates that the image we are referring to is a so-called

The new builder is much faster that the old builder. This is specifically true for complex Dockerfiles. The builder parallelizes tasks as much as possible. The output generated also includes the timing of individual steps, as one can see in the preceding image. It is highly recommended that you enable the new builder by default.

Tagging Docker images

Now that we know how to author a Dockerfile and build an image using this Dockerfile, we have to understand a bit more about the tags used for container images. The fully qualified name of a container image is composed of the following parts:

```
<registry>/<company or user>/<repository name>:<tag>
```

where the individual parts have the following meaning:

<registry>	This is the URL to the image registry where the Docker image is hosted. By default, and if not specified explicitly, this is Docker Hub at https://hub.Docker.com
<organization or user>	This is the name of an organization (company), e.g., *microsoft*, *oracle*, *google*, or the name of a registered user such as *gnschenker*, which is the author's account name on Docker Hub.
<repository name>	This is the name of a Docker image. It is also called a repository, and multiple versions of the image can exist at the same time in a single repository. Examples are alpine, centos, python, or node.
<tag>	A tag can be used to describe a particular version of a Docker image. For example, there exist multiple versions of the ubuntu image on Docker Hub, such as 21.04, 20.04, 16.04, and so on. There is also a special tag called *latest*. It is usually used to designate the latest version of an image. If one omits the tag in Docker command, then the Docker CLI will automatically assume that it is *latest*.

Let us look at a few sample image names:

- **`alpine:`**

  This is equivalent to **`alpine:latest`**. In this example, we have only the **`<repository name>`** part defined. Thus, the Docker CLI assumes that the registry is the default registry, namely, Docker Hub. Note that the default registry can be changed in the configuration settings for the Docker CLI. In the preceding image name, we do also not define a value for **`<organization or user>`** part, which indicates that the image we are referring to is a so-called

official image that is curated by Docker and the official community behind that image; in this case, the community curating the Alpine Linux distro. There exist many other "official" images, mostly some Linux distros such as Centos and Ubuntu, or some development SDKs or language runtimes such as for Node, Python, Java, Golang, and many more.

- `ubuntu:20.04:`

  Here, we define the repository name and the tag of the image. Once again, since the registry is not defined, one assumes the image is stored on Docker Hub, and since no organization or user name is defined in the image name, the image must be an official image.

- `confluentinc/cp-kafka:6.2.0:`

  This image name designates the Apache Kafka image curated by the company Confluent Inc. The version we are referring to is `6.2.0`. The image is located on Docker Hub since no registry name is defined in the name. Here, `confluentinc` refers to an organization of the same name defined on Docker Hub.

- `gnschenker/pinger:latest:`

  This name refers to the *latest* version of the *pinger* image owned by the user `gnschenker`, which happens to be the author's user account on Docker Hub.

Creating an account on Docker Hub

To be able to push images to Docker Hub, you need to have an account for it. Creating an account is free, and no strings are attached to it. If you want to use Docker Hub professionally, though, it is recommended to use one of the paid subscriptions they offer.

To create a new account on Docker Hub, follow the following steps:

1. Navigate to the Docker Hub signup page: **https://hub.docker.com/signup**
2. Enter your desired **docker ID** (the account name), your **email**, and **password**. Once you have reconfirmed that you are not a robot, click the **Sign-UP** button.
3. Follow the instructions. You will have to validate your email to be able to login into your new account.
4. Login to your new account.
5. On the main page of your Docker Hub account, you should find, among other information:

   (A) Access to your profile

official image that is curated by Docker and the official community behind that image; in this case, the community curating the Alpine Linux distro. There exist many other "official" images, mostly some Linux distros such as Centos and Ubuntu, or some development SDKs or language runtimes such as for Node, Python, Java, Golang, and many more.

- `ubuntu:20.04`:

  Here, we define the repository name and the tag of the image. Once again, since the registry is not defined, one assumes the image is stored on Docker Hub, and since no organization or user name is defined in the image name, the image must be an official image.

- `confluentinc/cp-kafka:6.2.0`:

  This image name designates the Apache Kafka image curated by the company Confluent Inc. The version we are referring to is `6.2.0`. The image is located on Docker Hub since no registry name is defined in the name. Here, `confluentinc` refers to an organization of the same name defined on Docker Hub.

- `gnschenker/pinger:latest`:

  This name refers to the *latest* version of the *pinger* image owned by the user `gnschenker`, which happens to be the author's user account on Docker Hub.

Creating an account on Docker Hub

To be able to push images to Docker Hub, you need to have an account for it. Creating an account is free, and no strings are attached to it. If you want to use Docker Hub professionally, though, it is recommended to use one of the paid subscriptions they offer.

To create a new account on Docker Hub, follow the following steps:

1. Navigate to the Docker Hub signup page: **https://hub.docker.com/signup**
2. Enter your desired **docker ID** (the account name), your **email**, and **password**. Once you have reconfirmed that you are not a robot, click the **Sign-UP** button.
3. Follow the instructions. You will have to validate your email to be able to login into your new account.
4. Login to your new account.
5. On the main page of your Docker Hub account, you should find, among other information:

   (A) Access to your profile

(B) A list of your repositories

(C) The list of organizations you are a member of

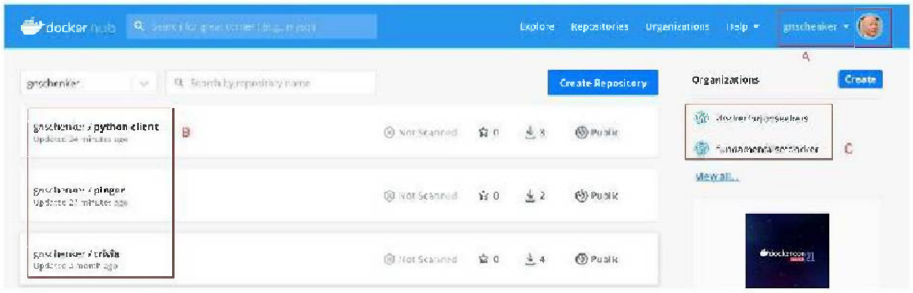


Figure 5.26: Your private account on Docker Hub

Now that you have acquired an account on Docker Hub, it is time to push some Docker images to it.

Publishing Docker images

To be able to share a Docker image that we custom-built, we need to upload it to some registry from where other interested parties can download it. There exist many such image registries. The most popular one is Docker Hub. Others include:

- Artifactory (**https://jfrog.com/artifactory**)
- Amazon Elastic Container Registry (**https://aws.amazon.com/de/ecr/**)
- Microsoft Azure Container Registry (**https://azure.microsoft.com/services/container-registry/**)
- Google Container Registry (**https://cloud.google.com/container-registry**)

To be able to push an image to a registry, you need the necessary permissions. Here, we concentrate on Docker Hub, but the other registries work similarly. Please read the respective documentation at the links provided previously.

Publishing to a personal account

First, we want to show how you can publish any custom image to your private account, from where it can be shared with the world. Please follow the following simple steps to do so:

1. Login to Docker Hub at **https://docker.com** using your personal account

(B) A list of your repositories

(C) The list of organizations you are a member of

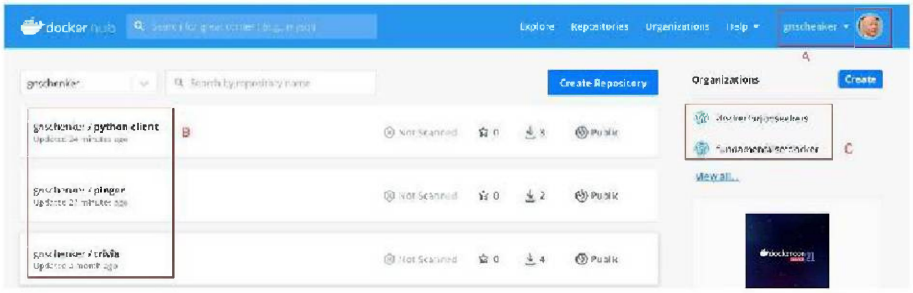


Figure 5.26: Your private account on Docker Hub

Now that you have acquired an account on Docker Hub, it is time to push some Docker images to it.

Publishing Docker images

To be able to share a Docker image that we custom-built, we need to upload it to some registry from where other interested parties can download it. There exist many such image registries. The most popular one is Docker Hub. Others include:

- Artifactory (**https://jfrog.com/artifactory**)
- Amazon Elastic Container Registry (**https://aws.amazon.com/de/ecr/**)
- Microsoft Azure Container Registry (**https://azure.microsoft.com/services/container-registry/**)
- Google Container Registry (**https://cloud.google.com/container-registry**)

To be able to push an image to a registry, you need the necessary permissions. Here, we concentrate on Docker Hub, but the other registries work similarly. Please read the respective documentation at the links provided previously.

Publishing to a personal account

First, we want to show how you can publish any custom image to your private account, from where it can be shared with the world. Please follow the following simple steps to do so:

1. Login to Docker Hub at **https://docker.com** using your personal account

2. Define an environment variable containing the name of your user account:

   ```
   export USER_NAME=gnschenker
   ```

   IMPORTANT: Please make sure that you use the name or your own user instead of **gnschenker** in the preceding statement!

3. First, we have to tag an existing image such as that it will be part of our account. In this example, we will use the Docker image called **python-client** that we created in the previous chapter:

   ```
   docker image tag python-client:latest $USER_NAME/python-client:1.0
   ```

4. Now, we can push the image to Docker Hub with the following:

   ```
   docker image push $USER_NAME/python-client:1.0
   ```

   You should see something like this:

```
gabriel@Solaris pinger % export USER_NAME=gnschenker
gabriel@Solaris pinger % docker image tag python-client:latest $USER_NAME/python-client:1.0
gabriel@Solaris pinger % docker image push $USER_NAME/python-client:1.0
The push refers to repository [docker.io/gnschenker/python-client]
b713f7f9119d: Pushed
cfec1f88021b: Pushed
ac7918c93e60: Pushed
0a2290f1fdfe: Pushed
d77b267ce3a6: Mounted from library/python
d199731f4103: Mounted from library/python
d32baaa3c3b9: Mounted from library/python
8fa76243d1f7: Mounted from library/python
f664faeafab9: Mounted from library/python
39b7d1cda0ba: Mounted from library/python
34b28b28bc44: Mounted from library/python
c850efa0dc51: Mounted from library/python
f3c97e3bd1e2: Mounted from library/python
1.0: digest: sha256:a0e10aa5bcbaf4263b81d4d3ab990a7ed59f550b4926cecf85c9e6da9d5ebbbf size: 3049
```

Figure 5.27: Pushing a docker image to our private account

   As you can see in the preceding output, the image has been successfully pushed to the account **gnschenker**. Since image layers are immutable, Docker uses some optimization and does only push layers that are new. You can recognize that because Docker is telling us that the last nine layers have been mounted from an existing library (or repository). In this case, they were mounted from the *python* repository.

Since, by default, all repositories of a user are public, anybody with internet access can now pull the image we just pushed.

Next, we want to discuss what organizations are and how you can push images to an organization.

2. Define an environment variable containing the name of your user account:

   ```
   export USER_NAME=gnschenker
   ```

   IMPORTANT: Please make sure that you use the name or your own user instead of **gnschenker** in the preceding statement!

3. First, we have to tag an existing image such as that it will be part of our account. In this example, we will use the Docker image called **python-client** that we created in the previous chapter:

   ```
   docker image tag python-client:latest $USER_NAME/python-client:1.0
   ```

4. Now, we can push the image to Docker Hub with the following:

   ```
   docker image push $USER_NAME/python-client:1.0
   ```

   You should see something like this:

```
gabriel@Solaris pinger % export USER_NAME=gnschenker
gabriel@Solaris pinger % docker image tag python-client:latest $USER_NAME/python-client:1.0
gabriel@Solaris pinger % docker image push $USER_NAME/python-client:1.0
The push refers to repository [docker.io/gnschenker/python-client]
b713f7f9119d: Pushed
cfec1f88021b: Pushed
ac7918c93e60: Pushed
0a2290f1fdfe: Pushed
d77b267ce3a6: Mounted from library/python
d199731f4103: Mounted from library/python
d32baaa3c3b9: Mounted from library/python
8fa76243d1f7: Mounted from library/python
f664faeafab9: Mounted from library/python
39b7d1cda0ba: Mounted from library/python
34b28b28bc44: Mounted from library/python
c850efa0dc51: Mounted from library/python
f3c97e3bd1e2: Mounted from library/python
1.0: digest: sha256:a0e10aa5bcbaf4263b81d4d3ab990a7ed59f550b4926cecf85c9e6da9d5ebbbf size: 3049
```

Figure 5.27: Pushing a docker image to our private account

   As you can see in the preceding output, the image has been successfully pushed to the account **gnschenker**. Since image layers are immutable, Docker uses some optimization and does only push layers that are new. You can recognize that because Docker is telling us that the last nine layers have been mounted from an existing library (or repository). In this case, they were mounted from the *python* repository.

Since, by default, all repositories of a user are public, anybody with internet access can now pull the image we just pushed.

Next, we want to discuss what organizations are and how you can push images to an organization.

Publishing to an organization

Now, if you are using Docker Hub commercially, then you might want to define and use an organization on Docker Hub to which you will then push your images. The advantages of organizations are that you can make the repositories private and that you can associate many accounts with a single organization. Thus, whole teams can push images to an organization.

Let us provide a step-by-step guide on how to define and prepare an organization on Docker Hub and then how to tag and push a custom image. Please follow the following steps:

1. Login to Docker Hub at **https://docker.com** using your personal account
2. Navigate to the Organizations tab (**https://hub.docker.com/orgs**)
3. Click **Create Organization** to start the creation process. Unfortunately, this will lead you to a dialog where you are required to select a paid subscription. Thus, this is not an option for us.

Since it is not possible to create and use an organization for free, we will simulate this by using another private account on Docker Hub and then converting it to an organization and assigning it to our own private account. To do so, follow the following steps:

1. Register a new temporary user on Docker Hub. In our case, we call it `dockerforjobseekers`.
2. Once the registration process is completed login to that new account on Docker Hub
3. Navigate to **Account Settings** via the context menu (to display the context menu shown as follows click on your account name in the upper left of the Docker Hub page):

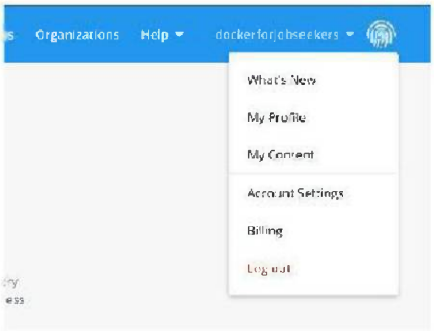


Figure 5.28: Account settings on Docker Hub

Publishing to an organization

Now, if you are using Docker Hub commercially, then you might want to define and use an organization on Docker Hub to which you will then push your images. The advantages of organizations are that you can make the repositories private and that you can associate many accounts with a single organization. Thus, whole teams can push images to an organization.

Let us provide a step-by-step guide on how to define and prepare an organization on Docker Hub and then how to tag and push a custom image. Please follow the following steps:

1. Login to Docker Hub at **https://docker.com** using your personal account
2. Navigate to the Organizations tab (**https://hub.docker.com/orgs**)
3. Click **Create Organization** to start the creation process. Unfortunately, this will lead you to a dialog where you are required to select a paid subscription. Thus, this is not an option for us.

Since it is not possible to create and use an organization for free, we will simulate this by using another private account on Docker Hub and then converting it to an organization and assigning it to our own private account. To do so, follow the following steps:

1. Register a new temporary user on Docker Hub. In our case, we call it **`dockerforjobseekers`**.
2. Once the registration process is completed login to that new account on Docker Hub
3. Navigate to **Account Settings** via the context menu (to display the context menu shown as follows click on your account name in the upper left of the Docker Hub page):

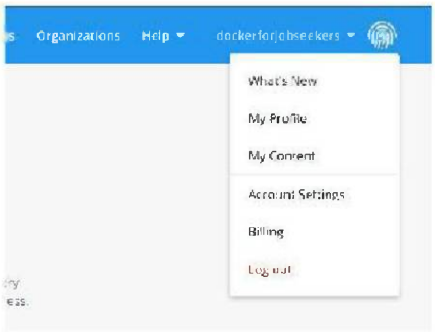


Figure 5.28: Account settings on Docker Hub

4. On the next page, select the Convert Account tab and then follow the instructions to convert the account to an organization. When asked to which account to assign the organization enter the name of your private account. In our case, this was **gnschenker**, the author's private account.

Login to your private account, and you should notice that a new organization is listed:

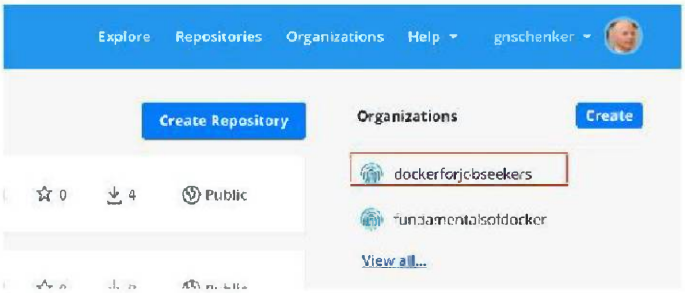


Figure 5.29: `Dockerforjobseeker` organization listed in the author's private account

From this moment on, you are able to push images not only to your own private account but also to the organization you just created. Let us try this:

4. Define an environment variable containing the name of the organization:

   ```
   export ORG_NAME=dockerforjobseekers
   ```

   IMPORTANT: Please make sure that you use the name or your own organization that you just defined instead of **`dockerforjobseekers`** in the preceding statement!

5. First, we have to tag an existing image such that it will be part of the new organization:

   ```
   docker image tag pinger:latest $ORG_NAME/pinger:1.0
   ```

6. Push the image to Docker Hub with the following:

   ```
   docker image push $ORG_NAME/pinger:1.0
   ```

   you should see something like this:

```
gabriel@Solaris pinger % docker image push $ORG_NAME/pinger:1.0
The push refers to repository [docker.io/dockerforjobseekers/pinger]
656d01d3□fe3: Pushed
edefbcfd02e5: Mounted from library/ubuntu
1.0: digest: sha256:e15a16662b7029ea3fa22fc88141e603b23d55bf443107e34e1a42019cfb6611 size: 741
```

Figure 5.30: Pushing the pinger image to the organization

4. On the next page, select the Convert Account tab and then follow the instructions to convert the account to an organization. When asked to which account to assign the organization enter the name of your private account. In our case, this was **gnschenker**, the author's private account.

Login to your private account, and you should notice that a new organization is listed:

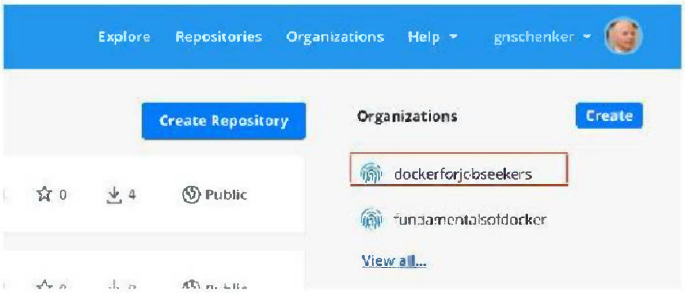


Figure 5.29: `Dockerforjobseeker` organization listed in the author's private account

From this moment on, you are able to push images not only to your own private account but also to the organization you just created. Let us try this:

4. Define an environment variable containing the name of the organization:

   ```
   export ORG_NAME=dockerforjobseekers
   ```

   IMPORTANT: Please make sure that you use the name or your own organization that you just defined instead of **`dockerforjobseekers`** in the preceding statement!

5. First, we have to tag an existing image such that it will be part of the new organization:

   ```
   docker image tag pinger:latest $ORG_NAME/pinger:1.0
   ```

6. Push the image to Docker Hub with the following:

   ```
   docker image push $ORG_NAME/pinger:1.0
   ```

   you should see something like this:

```
gabriel@Solaris pinger % docker image push $ORG_NAME/pinger:1.0
The push refers to repository [docker.io/dockerforjobseekers/pinger]
656d01d3□fe3: Pushed
edefbcfd02e5: Mounted from library/ubuntu
1.0: digest: sha256:e15a16662b7029ea3□a22fc88141e603b23d55bf443107e34e1a42019cfb6611 size: 741
```

Figure 5.30: Pushing the pinger image to the organization

This proves that you indeed have the right to push to that organization.

7. If you want to give the right to others to push to the organization, you need to invite them to the organization first:

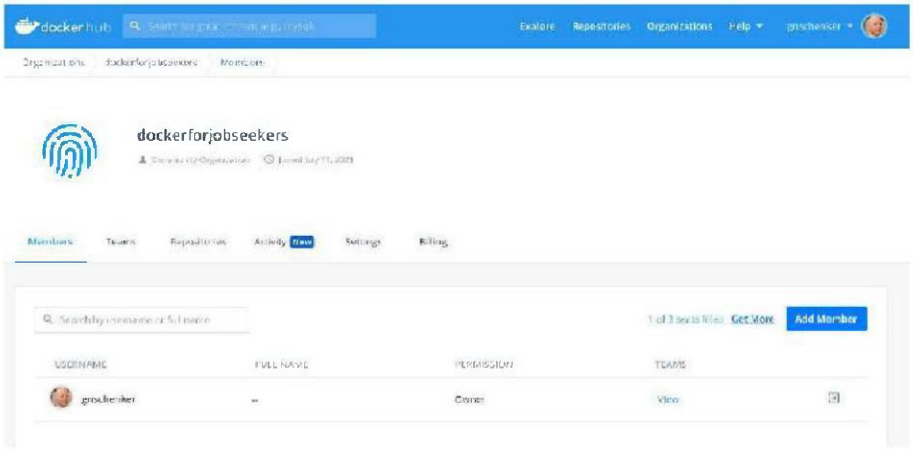


Figure 5.31: Adding another member to the organization

As you can see in the preceding figure, you have the option to add up to three members to an organization in the free subscription. To add a new member, just click on the **Add Member** button and follow the instruction.

With that, we are at the end of this chapter. In the next section, we will recap what we have learned so far.

Conclusion

In this chapter, we have learned three possible ways of how one can create a custom Docker image. The three options are to:

1. manually and interactively build a container from which one can then commit an image.
2. author a Dockerfile, which will be manifest for the Docker builder containing the exact list of instructions on how to build the image
3. create a new image in the system by importing a tar file

We were told that the second method, using a Dockerfile, is the preferred method in most cases since it enables automation, and thus, is repeatable and auditable.

This proves that you indeed have the right to push to that organization.

7. If you want to give the right to others to push to the organization, you need to invite them to the organization first:

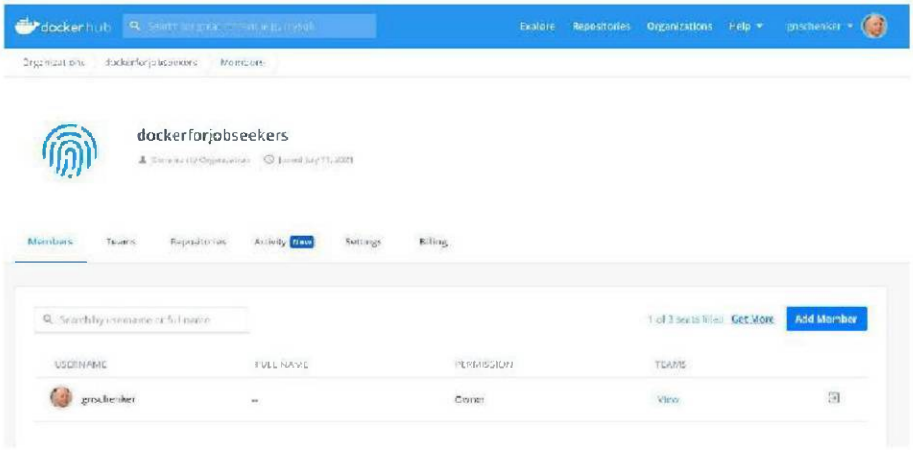


Figure 5.31: Adding another member to the organization

As you can see in the preceding figure, you have the option to add up to three members to an organization in the free subscription. To add a new member, just click on the **Add Member** button and follow the instruction.

With that, we are at the end of this chapter. In the next section, we will recap what we have learned so far.

Conclusion

In this chapter, we have learned three possible ways of how one can create a custom Docker image. The three options are to:

1. manually and interactively build a container from which one can then commit an image.
2. author a Dockerfile, which will be manifest for the Docker builder containing the exact list of instructions on how to build the image
3. create a new image in the system by importing a tar file

We were told that the second method, using a Dockerfile, is the preferred method in most cases since it enables automation, and thus, is repeatable and auditable.

We learned how to use the most frequently needed keywords that can be used in a Dockerfile, and we were introduced to the details on how the Docker builder is actually building up the layers of a custom image.

We then learned of which parts the full name of a Docker image is composed of and what their respective meaning is.

Last but not least, we got to know how one can share a custom image by pushing it to a container registry. We were taught on how to do this using Docker Hub as our registry.

In the upcoming chapter, you will learn all the important details about Docker networks; what they are, how they work, and how one creates and uses them. Stay tuned.

Questions

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

1. Which is the preferred way of creating a custom Docker image, and why is this the recommended approach in most cases?
2. What are the main differences between pushing an image to a user account or to an organization on Docker Hub?
3. Of which parts is the full name of a Docker image composed of? Give a short description of each part.
4. Explain what the CMD and ENTRYPOINT keywords stand for.
5. List two to three best practices when authoring a Dockerfile.

Answers

Here are possible answers to the questions:

1. Authoring a Dockerfile is the preferred way. A Dockerfile is a basis for automation, and once automation is used, the process of building a Docker image becomes repeatable and auditable. Even as a developer, during the development process, it is preferable to build an image from a Dockerfile.
2. Only the owner of the account is allowed to push images to the account. When using an organization, many individual users can be made members of the org and, as such, can push images to the org. Using organizations is the preferred way when using the registry professionally.
3. The full name of a Docker image is composed of four parts: `<registry>/<org or user>/<repository name>:<tag>`

We learned how to use the most frequently needed keywords that can be used in a Dockerfile, and we were introduced to the details on how the Docker builder is actually building up the layers of a custom image.

We then learned of which parts the full name of a Docker image is composed of and what their respective meaning is.

Last but not least, we got to know how one can share a custom image by pushing it to a container registry. We were taught on how to do this using Docker Hub as our registry.

In the upcoming chapter, you will learn all the important details about Docker networks; what they are, how they work, and how one creates and uses them. Stay tuned.

Questions

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

1. Which is the preferred way of creating a custom Docker image, and why is this the recommended approach in most cases?
2. What are the main differences between pushing an image to a user account or to an organization on Docker Hub?
3. Of which parts is the full name of a Docker image composed of? Give a short description of each part.
4. Explain what the CMD and ENTRYPOINT keywords stand for.
5. List two to three best practices when authoring a Dockerfile.

Answers

Here are possible answers to the questions:

1. Authoring a Dockerfile is the preferred way. A Dockerfile is a basis for automation, and once automation is used, the process of building a Docker image becomes repeatable and auditable. Even as a developer, during the development process, it is preferable to build an image from a Dockerfile.
2. Only the owner of the account is allowed to push images to the account. When using an organization, many individual users can be made members of the org and, as such, can push images to the org. Using organizations is the preferred way when using the registry professionally.
3. The full name of a Docker image is composed of four parts: `<registry>/<org or user>/<repository name>:<tag>`

   a. `<registry>`: the URL to the Docker image registry. Many different options are available, e.g., from AWS, Azure, Google, or Artifactory, to just name a few

   b. `<org or user>`: the name of an organization or a user name to which the Docker image belongs. Examples are Microsoft, Oracle, Confluentinc, and so on.

   c. `<repository name>`: the name of the repository (also referred to as the image name)

   d. `<tag>`: the tag of an image. This can be a version number such as `20.04`, e.g., the ubuntu image

4. ENTRYPOINT is used to define the command that is triggered when a container is started. CMD usually defines the arguments to the command defined in the ENTRYPOINT. For example, the following Linux command

   ```
   ping 8.8.8.8 -c 5
   ```

   would ideally be defined in a Dockerfile like the following:

   ```
   ...
   ENTRYPOINT ping
   CMD 8.8.8.8 -c 5
   ```

5. The following are some best practices recommended when authoring Dockerfiles:

   a. Use multi-stage Dockerfiles to keep the size of the final image as small as possible by including only the minimal needed executables, libraries, and other artifacts.

   b. Ordering matters in a Dockerfile since the Docker CLI uses caching aggressively. Try to have the instructions that change the least frequent at the top of the Dockerfile and those that change frequently toward the bottom

   c. Keep the number of layers small by combining (Linux) commands in a single RUN statement.

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 start-up that develops a highly acclaimed new mobile app for iOS and Android. During the interview, you are confronted with the following questions:

   a. `<registry>`: the URL to the Docker image registry. Many different options are available, e.g., from AWS, Azure, Google, or Artifactory, to just name a few

   b. `<org or user>`: the name of an organization or a user name to which the Docker image belongs. Examples are Microsoft, Oracle, Confluentinc, and so on.

   c. `<repository name>`: the name of the repository (also referred to as the image name)

   d. `<tag>`: the tag of an image. This can be a version number such as `20.04`, e.g., the ubuntu image

4. ENTRYPOINT is used to define the command that is triggered when a container is started. CMD usually defines the arguments to the command defined in the ENTRYPOINT. For example, the following Linux command

   ```
   ping 8.8.8.8 -c 5
   ```

   would ideally be defined in a Dockerfile like the following:

   ```
   ...
   ENTRYPOINT ping
   CMD 8.8.8.8 -c 5
   ```

5. The following are some best practices recommended when authoring Dockerfiles:

   a. Use multi-stage Dockerfiles to keep the size of the final image as small as possible by including only the minimal needed executables, libraries, and other artifacts.

   b. Ordering matters in a Dockerfile since the Docker CLI uses caching aggressively. Try to have the instructions that change the least frequent at the top of the Dockerfile and those that change frequently toward the bottom

   c. Keep the number of layers small by combining (Linux) commands in a single RUN statement.

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 start-up that develops a highly acclaimed new mobile app for iOS and Android. During the interview, you are confronted with the following questions:

1. Why is it so important that we write a Dockerfile used to build Docker images?
2. We have heard of multi-stage Dockerfiles but have never used them so far. Can you give us a couple of reasons why we should start using multi-stage Dockerfiles?
3. We have heard that one should strive for to keep a Docker image as small as possible. What does this mean? How can we achieve that? And why is it so important?
4. Our developers always get confused about when to use the CMD or the ENTRYPOINT keyword in their Dockerfiles. What can you tell us about this?
5. Can you explain to an interested layperson how the Docker builder builds an image from a Docker file? Just give a very high-level overview, please.

Job interview sample answers

Here are possible answers to the job interview questions:

1. As mentioned in the sample answers, a Dockerfile is a basis for automation, and once automation is used, the process of building a Docker image becomes repeatable and auditable. Even as a developer, during the development process, it is preferable to build an image from a Dockerfile.
2. With the help of multi-stage Dockerfiles, we can keep the size of the resulting Docker images small since, in the last stage, we can use a minimal base image and copy the artifacts generated in the preceding steps into this final image. Thus, we avoid having unnecessary elements in the final image, such as compilers or linkers. Furthermore, when using multi-stage Dockerfiles, the builder can try to parallelize tasks and, ideally, as a result, massively decrease the time needed to build an image.
3. We should try to have only elements in a Docker image that are absolutely needed during runtime. Any obsolete library, program, or other artifacts only increases the attack surface of the image for malicious actors. Also, small images need less resources to store, transport, or run. One popular method we can use to achieve small sizes is using multi-stage Dockerfiles.
4. It is not usually a problem to either only use CMD or ENTRYPOINT when defining the start command that shall be executed when a container is started. But the correct way of using the two is as follows:
    a. Use ENTRYPOINT to define the process or command that shall be executed

1. Why is it so important that we write a Dockerfile used to build Docker images?
2. We have heard of multi-stage Dockerfiles but have never used them so far. Can you give us a couple of reasons why we should start using multi-stage Dockerfiles?
3. We have heard that one should strive for to keep a Docker image as small as possible. What does this mean? How can we achieve that? And why is it so important?
4. Our developers always get confused about when to use the CMD or the ENTRYPOINT keyword in their Dockerfiles. What can you tell us about this?
5. Can you explain to an interested layperson how the Docker builder builds an image from a Docker file? Just give a very high-level overview, please.

Job interview sample answers

Here are possible answers to the job interview questions:

1. As mentioned in the sample answers, a Dockerfile is a basis for automation, and once automation is used, the process of building a Docker image becomes repeatable and auditable. Even as a developer, during the development process, it is preferable to build an image from a Dockerfile.
2. With the help of multi-stage Dockerfiles, we can keep the size of the resulting Docker images small since, in the last stage, we can use a minimal base image and copy the artifacts generated in the preceding steps into this final image. Thus, we avoid having unnecessary elements in the final image, such as compilers or linkers. Furthermore, when using multi-stage Dockerfiles, the builder can try to parallelize tasks and, ideally, as a result, massively decrease the time needed to build an image.
3. We should try to have only elements in a Docker image that are absolutely needed during runtime. Any obsolete library, program, or other artifacts only increases the attack surface of the image for malicious actors. Also, small images need less resources to store, transport, or run. One popular method we can use to achieve small sizes is using multi-stage Dockerfiles.
4. It is not usually a problem to either only use CMD or ENTRYPOINT when defining the start command that shall be executed when a container is started. But the correct way of using the two is as follows:
    a. Use ENTRYPOINT to define the process or command that shall be executed

b. Use CMD to define the parameters that shall be passed to the process defined in a).

For a command like

```
wget -O - https://acme.com/install/scripts.sh
```

one would define the following fragment in the Dockerfile:

```
...
ENTRYPOINT wget
CMD -O - https://acme.com/install/script.sh
```

5. The builder first downloads and caches the base image that is defined in the Dockerfile. Then the builder creates a temporary container from this image. Inside the read/write container layer, the next statement of the Dockerfile is executed. The changes that this operation causes in the container layer are then committed into a new image layer by the builder. The now obsolete temporary container is then deleted. Next, the builder creates another temporary container from the combined layers of the base image and the previously committed layer. The third statement of the Dockerfile is then executed inside this container. The changes are once again committed to a new image layer, and the temporary container is deleted. This process continues until all statements in the Dockerfile have been processed.

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



b. Use CMD to define the parameters that shall be passed to the process defined in a).

For a command like

```
wget -O - https://acme.com/install/scripts.sh
```

one would define the following fragment in the Dockerfile:

```
...
ENTRYPOINT wget
CMD -O - https://acme.com/install/script.sh
```

5. The builder first downloads and caches the base image that is defined in the Dockerfile. Then the builder creates a temporary container from this image. Inside the read/write container layer, the next statement of the Dockerfile is executed. The changes that this operation causes in the container layer are then committed into a new image layer by the builder. The now obsolete temporary container is then deleted. Next, the builder creates another temporary container from the combined layers of the base image and the previously committed layer. The third statement of the Dockerfile is then executed inside this container. The changes are once again committed to a new image layer, and the temporary container is deleted. This process continues until all statements in the Dockerfile have been processed.

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 6

Demystifying Container Networking

Introduction

In the previous chapter, we have learned three possible ways of how one can create a custom Docker image. The first way is by manually and interactively building up a container and then committing its current state as a new image. This is useful when one is in an exploratory phase of a project. The second and most popular way is to author a Dockerfile, which will contain a human-readable manifest of what the desired custom image shall be made of. This is the way you would build your custom images in most cases. We got to know the most important elements of a Dockerfile and also how the fully qualified name of a Docker image is built such that it can be identified, versioned and shared through an image repository such as Docker Hub.

Finally, we learned that one can also create a new image in the system by importing it as a tar file. This is helpful if you need to share images with systems that have no direct internet access.

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 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, the reader is introduced to network types that are able

Chapter 6

Demystifying Container Networking

Introduction

In the previous chapter, we have learned three possible ways of how one can create a custom Docker image. The first way is by manually and interactively building up a container and then committing its current state as a new image. This is useful when one is in an exploratory phase of a project. The second and most popular way is to author a Dockerfile, which will contain a human-readable manifest of what the desired custom image shall be made of. This is the way you would build your custom images in most cases. We got to know the most important elements of a Dockerfile and also how the fully qualified name of a Docker image is built such that it can be identified, versioned and shared through an image repository such as Docker Hub.

Finally, we learned that one can also create a new image in the system by importing it as a tar file. This is helpful if you need to share images with systems that have no direct internet access.

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 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, the reader is introduced to network types that are able

to 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. In this chapter, we will mainly use the terminal and some simple Linux tools.

Structure

In this chapter, we will cover the following topics:

- Explaining the container network model
- Discussing the bridge network
- Elucidating container ports
- Discussing the null and host networks and multi-node networks
- Presenting tips on how to answer possible job interview questions related to container networking

Objective

After concluding this chapter, you will be able to explain to an interested layperson, in your own words, what the Docker container network model is. You will be able to use the Docker bridge network to connect containers running on your laptop and open ports in a container through which you can access the application running inside the container.

You will learn to describe in a few short sentences what the none and the host network are and when you would use each of them.

Without further ado, let us start.

Explaining the container network model

Docker has defined the core concepts or boundary conditions that a software defined network must fulfil to be used to connect Docker containers. Let us provide a simple graphic that illustrates this networking model:

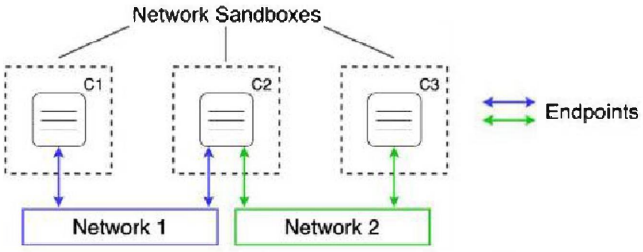


Figure 6.1: Docker container network model

to 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. In this chapter, we will mainly use the terminal and some simple Linux tools.

Structure

In this chapter, we will cover the following topics:

- Explaining the container network model
- Discussing the bridge network
- Elucidating container ports
- Discussing the null and host networks and multi-node networks
- Presenting tips on how to answer possible job interview questions related to container networking

Objective

After concluding this chapter, you will be able to explain to an interested layperson, in your own words, what the Docker container network model is. You will be able to use the Docker bridge network to connect containers running on your laptop and open ports in a container through which you can access the application running inside the container.

You will learn to describe in a few short sentences what the none and the host network are and when you would use each of them.

Without further ado, let us start.

Explaining the container network model

Docker has defined the core concepts or boundary conditions that a software defined network must fulfil to be used to connect Docker containers. Let us provide a simple graphic that illustrates this networking model:

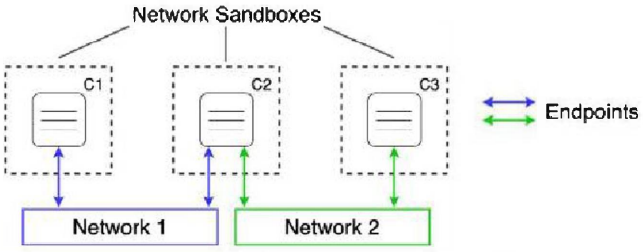


Figure 6.1: Docker container network model

In the preceding graphic, we have three containers, C1, C2, and C3. Each one of those containers is firewalled by a network sandbox that makes it impossible for any external entity to access the application or process running inside the respective container. But when an application running inside a container is completely isolated from the environment, it is, in most cases, not very useful. Different applications or services need to communicate with each other.

For applications running inside containers to be able to communicate with each other, we need some kind of pathways over which this communication can flow. We call these pathway networks. We have shown two such networks; Network 1 in blue and Network 2 in green.

Furthermore, we have shown various endpoints on the image. An endpoint can be looked at as a controlled gateway from a network into a network sandbox that shields a container. One endpoint connects container C1 with Network 1. Container C3 also is connected by a single endpoint with Network 2. Finally, container C2 is connected by one endpoint with Network 1 and by a second endpoint with Network 2.

It is important to know that each container sitting inside a networking sandbox can be attached to no network at all and remain isolated, or it can be attached to one or many networks allowing it to communicate with other containers also being connected to those networks. In our case, container C1 can communicate (only) with container C2. Container C3 can (only) communicate with container C2, whereas container C2 can communicate with both containers C1 and C2.

To use multiple software-defined networks makes a lot of sense if you want to maximize the security of your distributed application. Take as an example an application shown as follows:

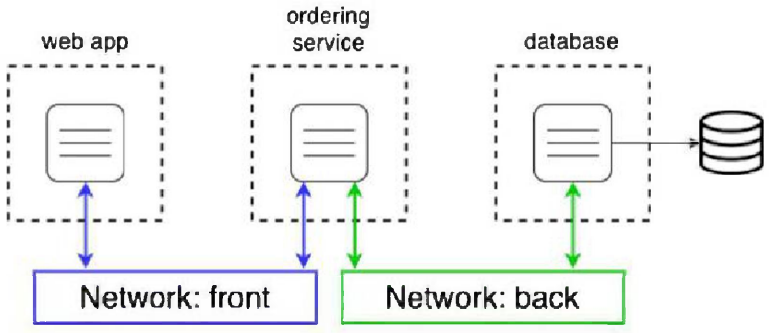


Figure 6.2: Distributed application secured with multiple SDNs

In the preceding graphic, we have three containers, C1, C2, and C3. Each one of those containers is firewalled by a network sandbox that makes it impossible for any external entity to access the application or process running inside the respective container. But when an application running inside a container is completely isolated from the environment, it is, in most cases, not very useful. Different applications or services need to communicate with each other.

For applications running inside containers to be able to communicate with each other, we need some kind of pathways over which this communication can flow. We call these pathway networks. We have shown two such networks; Network 1 in blue and Network 2 in green.

Furthermore, we have shown various endpoints on the image. An endpoint can be looked at as a controlled gateway from a network into a network sandbox that shields a container. One endpoint connects container C1 with Network 1. Container C3 also is connected by a single endpoint with Network 2. Finally, container C2 is connected by one endpoint with Network 1 and by a second endpoint with Network 2.

It is important to know that each container sitting inside a networking sandbox can be attached to no network at all and remain isolated, or it can be attached to one or many networks allowing it to communicate with other containers also being connected to those networks. In our case, container C1 can communicate (only) with container C2. Container C3 can (only) communicate with container C2, whereas container C2 can communicate with both containers C1 and C2.

To use multiple software-defined networks makes a lot of sense if you want to maximize the security of your distributed application. Take as an example an application shown as follows:

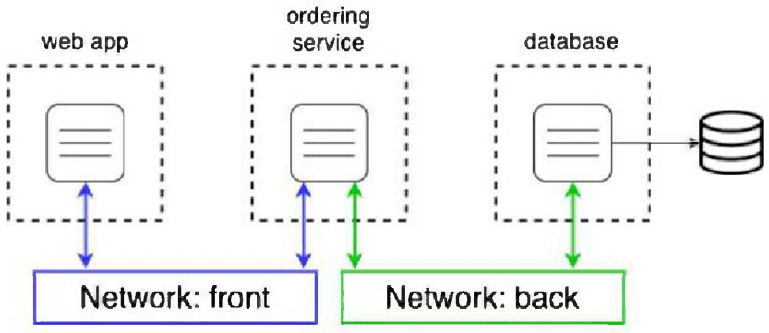


Figure 6.2: Distributed application secured with multiple SDNs

We have a Web UI called `web app` that is attached to the network called `front`. The Web application should only ever communicate with the ordering service. Thus, this service is attached to the same network `front`. The ordering service, on the other hand, needs to store data in a database—assume a PostgreSQL DB as an example. If we run the database in a container, then this container and the ordering service communicate with each other via a network called `back`.

If we have such a topology, then it is impossible for code inside the Web app container to access the database. Thus, if a hacker is able to hack the Web app container, they still have no chance to access or tinker with the database. To do so, they would also have to hack the ordering service, which seems quite improbable.

Software-defined networks (SDNs) are cheap. Each network that you add adds an additional layer of security atop your resources and protects them from malicious actors, such as injected or breached code.

The **container network model** (CNM) is just that, a model, and we need a concrete implementation to work with. Docker comes with a few implementations of the CNM out of the box. Other providers have created additional implementations that one can leverage. We will investigate the list of network implementations that are part of the Docker installation in the remainder of this chapter.

By default, Docker creates several networks when it is installed. These networks include the following:

1. **bridge**: This is the default network that is created when Docker is installed. Containers connected to this network are able to communicate with each other but are isolated from the host machine and from other networks.
2. **host**: Containers connected to this network have the same network stack as the host machine. This means that containers can access the host machine's network directly without being isolated.
3. **none**: Containers connected to this network do not have any network stack, which means that they cannot communicate with other containers or with the host machine. This network is usually used for containers that are run with the **--net=none** option.
4. **overlay**: This network allows to create of multi-host networks. This allows connecting containers running on different hosts together.

Please note that these are the default networks that are created by Docker, and we can create our own custom networks if we need to configure more advanced networking options.

First, we will start with the default network called the bridge network.

We have a Web UI called `web app` that is attached to the network called `front`. The Web application should only ever communicate with the ordering service. Thus, this service is attached to the same network `front`. The ordering service, on the other hand, needs to store data in a database—assume a PostgreSQL DB as an example. If we run the database in a container, then this container and the ordering service communicate with each other via a network called `back`.

If we have such a topology, then it is impossible for code inside the Web app container to access the database. Thus, if a hacker is able to hack the Web app container, they still have no chance to access or tinker with the database. To do so, they would also have to hack the ordering service, which seems quite improbable.

Software-defined networks (SDNs) are cheap. Each network that you add adds an additional layer of security atop your resources and protects them from malicious actors, such as injected or breached code.

The **container network model** (CNM) is just that, a model, and we need a concrete implementation to work with. Docker comes with a few implementations of the CNM out of the box. Other providers have created additional implementations that one can leverage. We will investigate the list of network implementations that are part of the Docker installation in the remainder of this chapter.

By default, Docker creates several networks when it is installed. These networks include the following:

1. **bridge**: This is the default network that is created when Docker is installed. Containers connected to this network are able to communicate with each other but are isolated from the host machine and from other networks.
2. **host**: Containers connected to this network have the same network stack as the host machine. This means that containers can access the host machine's network directly without being isolated.
3. **none**: Containers connected to this network do not have any network stack, which means that they cannot communicate with other containers or with the host machine. This network is usually used for containers that are run with the **--net=none** option.
4. **overlay**: This network allows to create of multi-host networks. This allows connecting containers running on different hosts together.

Please note that these are the default networks that are created by Docker, and we can create our own custom networks if we need to configure more advanced networking options.

First, we will start with the default network called the bridge network.

Discussing the bridge network

The Docker CLI has a network context with a few commands that help us to list installed networks, create new networks, inspect a given network, and delete a network. Let us start with the former one.

If we list the networks on a clean Docker environment with the following command:

```
docker network ls
```

Then, we should see this:

```
gabriel@Solaris ch06 % docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
3feb5c42ab4f   bridge    bridge    local
77b3da317100   host      host      local
93085f159974   none      null      local
```

Figure 6.3: List of networks in a clean Docker environment

As you can see from the preceding output, we have three networks available on the system that are created by Docker by default. One is a **`bridge`** network, the other one is a **`host`** network, and the last one is a **`null`** network. Each one of them we will explore each in detail and also provide typical usage scenarios for each. Note that we also speak of a **network driver** when we talk of a specific implementation of the container network model.

Each network in the preceding list has a unique ID and a name that must also be unique on the system. Furthermore, each network has a scope. In our list, the scope is local for all of them. This means that the network is limited to the current system on which it is defined. Only containers that run on this system can be part of that network.

There are network drivers that have **`global`** scope. That is, they are spanning multiple nodes. Global networks are used when you operate containers in a cluster such as a Kubernetes cluster or a Docker swarm. In this chapter, we will concentrate on local drivers exclusively. We will talk about global drivers in *Chapter 10, Orchestrating Containers*.

Note that in your case, the IDs of the network drivers may differ, but the remaining properties should be the same as in the preceding list.

Let us now concentrate on the first entry in the list preceding, the bridge driver. As stated before, Docker creates an instance of the bridge driver by default upon installation. It calls this network **`bridge`**.

Discussing the bridge network

The Docker CLI has a network context with a few commands that help us to list installed networks, create new networks, inspect a given network, and delete a network. Let us start with the former one.

If we list the networks on a clean Docker environment with the following command:

```
docker network ls
```

Then, we should see this:

```
gabriel@Solaris ch06 % docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
3feb5c42ab4f   bridge    bridge    local
77b3da317100   host      host      local
93085f159974   none      null      local
```

Figure 6.3: List of networks in a clean Docker environment

As you can see from the preceding output, we have three networks available on the system that are created by Docker by default. One is a **`bridge`** network, the other one is a **`host`** network, and the last one is a **`null`** network. Each one of them we will explore each in detail and also provide typical usage scenarios for each. Note that we also speak of a **network driver** when we talk of a specific implementation of the container network model.

Each network in the preceding list has a unique ID and a name that must also be unique on the system. Furthermore, each network has a scope. In our list, the scope is local for all of them. This means that the network is limited to the current system on which it is defined. Only containers that run on this system can be part of that network.

There are network drivers that have **`global`** scope. That is, they are spanning multiple nodes. Global networks are used when you operate containers in a cluster such as a Kubernetes cluster or a Docker swarm. In this chapter, we will concentrate on local drivers exclusively. We will talk about global drivers in *Chapter 10, Orchestrating Containers*.

Note that in your case, the IDs of the network drivers may differ, but the remaining properties should be the same as in the preceding list.

Let us now concentrate on the first entry in the list preceding, the bridge driver. As stated before, Docker creates an instance of the bridge driver by default upon installation. It calls this network **`bridge`**.

Inspecting a network

Let us inspect the details of this bridge network a bit more using the following command:

```
docker network inspect bridge
```

This is the output produced on the authors' system. It is fairly long but contains a lot of interesting information:



Figure 6.4: Inspecting the bridge network

The output starts with the ID, name and type of the network, as well as its creation date and whether or not the network enables IPv6 type of traffic. In the case of the bridge network, this is apparently not the case.

The next block is about the so-called IPAM settings, where IPAM stands for IP Address Management. The interesting part there is the definition of the subnet and the gateway. Apparently, Docker is using the subnet `172.17.0.0/16` to provide a valid IP address to all containers that will be attached to this network. The gateway

Inspecting a network

Let us inspect the details of this bridge network a bit more using the following command:

```
docker network inspect bridge
```

This is the output produced on the authors' system. It is fairly long but contains a lot of interesting information:



Figure 6.4: Inspecting the bridge network

The output starts with the ID, name and type of the network, as well as its creation date and whether or not the network enables IPv6 type of traffic. In the case of the bridge network, this is apparently not the case.

The next block is about the so-called IPAM settings, where IPAM stands for IP Address Management. The interesting part there is the definition of the subnet and the gateway. Apparently, Docker is using the subnet `172.17.0.0/16` to provide a valid IP address to all containers that will be attached to this network. The gateway

for this subnet has the first valid IP, which is `172.17.0.1`. Thus, the first container that will be attached to this network will get the IP `172.17.0.2` and so forth.

When one creates a new network—as we will do in a bit—one can either let Docker define the subnet automatically or one can define it explicitly.

By default, and if not specified otherwise, a container is automatically attached to this bridge network when it is created and run. How can we know that? The answer is by inspecting a container that we run. Let us do this as follows:

1. In a new terminal window, start an instance of the Ubuntu container in detach mode and call it `demo`:

   ```
   docker container run --detach --name demo ubuntu:20.04
   ```

2. Inspect this container using the following command:

   ```
   docker container inspect demo
   ```

   notice the output generated in your terminal window. The output is fairly long and detailed. Toward the end of the output, you find a node called `NetworkSettings` containing all information about the networks the container is attached to. In the following screenshot, you can see the interesting part of it:



Figure 6.5: Inspecting the network settings of a container

for this subnet has the first valid IP, which is `172.17.0.1`. Thus, the first container that will be attached to this network will get the IP `172.17.0.2` and so forth.

When one creates a new network—as we will do in a bit—one can either let Docker define the subnet automatically or one can define it explicitly.

By default, and if not specified otherwise, a container is automatically attached to this bridge network when it is created and run. How can we know that? The answer is by inspecting a container that we run. Let us do this as follows:

1. In a new terminal window, start an instance of the Ubuntu container in detach mode and call it `demo`:

   ```
   docker container run --detach --name demo ubuntu:20.04
   ```

2. Inspect this container using the following command:

   ```
   docker container inspect demo
   ```

   notice the output generated in your terminal window. The output is fairly long and detailed. Toward the end of the output, you find a node called `NetworkSettings` containing all information about the networks the container is attached to. In the following screenshot, you can see the interesting part of it:



Figure 6.5: Inspecting the network settings of a container

3. In the preceding output under the node **Networks**, we can see that the bride network is mentioned, and its **NetworkID** corresponds to the ID of the bridge network as it was listed previously.
4. When done, remove the container with the following:

```
docker container rm --force demo
```

Creating a new bridge network

Now, let us create our own bridge-type network with this command:

```
docker network create my-bridge
```

This will create a new bridge network called **my-bridge**. Docker automatically creates a network of type bridge if the driver is not specified in the create command. Docker also automatically assigns this network a subnet if we do not specify one. The subnet, in our case, will be **172.XX.0.0/16** where **XX** is a number equal to 18 or higher, as we can easily find out by using the network inspect command:

```
docker network inspect my-bridge
```

On the author's system, having played with networks, a lot of Dockers selected **172.27.0.0/16** as the subnet for the **my-bridge** network.

Analyzing the network stack of a container

If we attach two or more containers to the same network, are they able to communicate with each other, or are they? Let us make a simple test:

1. Open a new terminal window
2. Make sure no other container is running on the host by using this command:

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

3. Run a first Alpine container in interactive mode using a shell and name it one using the following command:

```
docker container run -it --name one alpine:latest ash
```

4. Open a second terminal window and inspect the container just created with the following:

```
docker container inspect one
```

3. In the preceding output under the node **Networks**, we can see that the bride network is mentioned, and its **NetworkID** corresponds to the ID of the bridge network as it was listed previously.
4. When done, remove the container with the following:

```
docker container rm --force demo
```

Creating a new bridge network

Now, let us create our own bridge-type network with this command:

```
docker network create my-bridge
```

This will create a new bridge network called **my-bridge**. Docker automatically creates a network of type bridge if the driver is not specified in the create command. Docker also automatically assigns this network a subnet if we do not specify one. The subnet, in our case, will be **172.XX.0.0/16** where **XX** is a number equal to 18 or higher, as we can easily find out by using the network inspect command:

```
docker network inspect my-bridge
```

On the author's system, having played with networks, a lot of Dockers selected **172.27.0.0/16** as the subnet for the **my-bridge** network.

Analyzing the network stack of a container

If we attach two or more containers to the same network, are they able to communicate with each other, or are they? Let us make a simple test:

1. Open a new terminal window
2. Make sure no other container is running on the host by using this command:

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

3. Run a first Alpine container in interactive mode using a shell and name it one using the following command:

```
docker container run -it --name one alpine:latest ash
```

4. Open a second terminal window and inspect the container just created with the following:

```
docker container inspect one
```

You should see this:

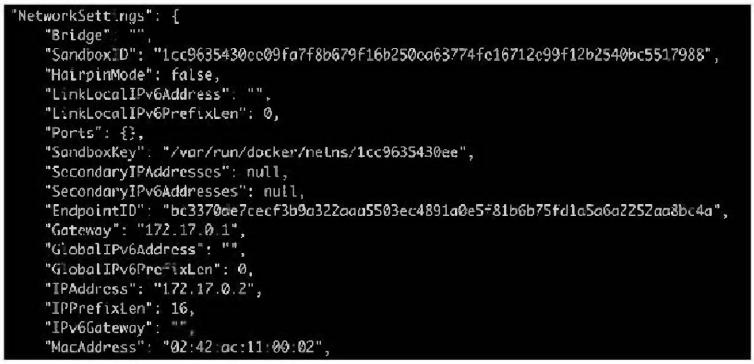


Figure 6.6: Network settings of container one

The preceding image shows the network settings part of the output. Please note that this container got the ip address 172.17.0.2 assigned and that the gateway of the subnet is at 172.17.0.1. Also note the MacAddress of the container, as the bridge network uses mac addresses to communicate.

5. Now back to the shell of container one. Let us use the **`ip`** Linux tool and run the following command to get information about the container's network stack from within the container:

   ```
   ip addr
   ```

   This command results in this output:



Figure 6.7: Network stack from inside container one

As we can see, marked by the red rectangle in the preceding output and part of the endpoint **`eth0`**, the IP address of the container is **`172.17.0.2`**, and the

You should see this:

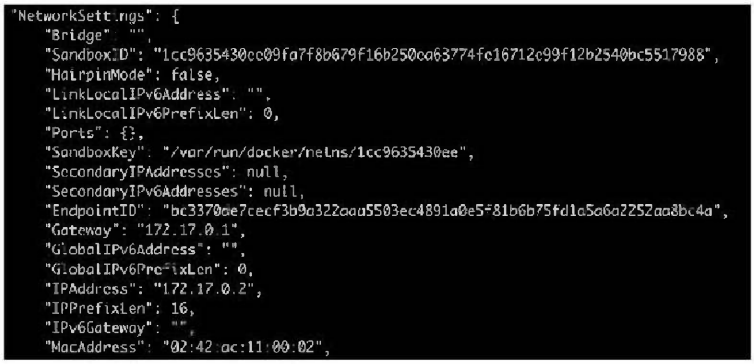


Figure 6.6: Network settings of container one

The preceding image shows the network settings part of the output. Please note that this container got the ip address 172.17.0.2 assigned and that the gateway of the subnet is at 172.17.0.1. Also note the MacAddress of the container, as the bridge network uses mac addresses to communicate.

5. Now back to the shell of container one. Let us use the **`ip`** Linux tool and run the following command to get information about the container's network stack from within the container:

   ```
   ip addr
   ```

   This command results in this output:



Figure 6.7: Network stack from inside container one

As we can see, marked by the red rectangle in the preceding output and part of the endpoint **`eth0`**, the IP address of the container is **`172.17.0.2`**, and the

Mac address is **02:42:ac:11:00:02**. These are the same values that we got while inspecting the container, which was to be expected.

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.

6. Since we are not interested at this time in any other endpoint inside the container than **eth0**, we can also run the following command to limit the output to the relevant information:

   ```
   ip addr show eth0
   ```

   This yields us the following output:

```
/ # ip addr show eth0
456: eth0@if457: <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 6.8: Details of the eth0 endpoint as seen from within the container

7. If we want to get more information about how traffic is routed inside the container, we can use the route command: **ip route**

   We should get this output:

```
/ # ip route
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0 scope link  src 172.17.0.2
```

Figure 6.9: Routing inside the container one

   This output tells us that all the traffic to the gateway at **172.17.0.1** is routed through the **eth0** device.

8. Open yet another terminal window and run a second Alpine container interactively and call it **two**:

9. **docker container run -it --name two alpine:latest ash**

10. Use the **ip** tool in the second container to inspect its network stack as you did for container **one**. Make sure that its IP address is **172.17.0.3** and the Mac address is the one you expect from inspecting container **two**.

11. Now, let us inspect the bridge network and confirm that both containers are attached to it. In your second terminal window, execute the following:

    ```
    docker network inspect bridge
    ```

Mac address is **02:42:ac:11:00:02**. These are the same values that we got while inspecting the container, which was to be expected.

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.

6. Since we are not interested at this time in any other endpoint inside the container than **eth0**, we can also run the following command to limit the output to the relevant information:

   ```
   ip addr show eth0
   ```

   This yields us the following output:

```
/ # ip addr show eth0
456: eth0@if457: <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 6.8: Details of the eth0 endpoint as seen from within the container

7. If we want to get more information about how traffic is routed inside the container, we can use the route command: **ip route**

   We should get this output:

```
/ # ip route
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0 scope link  src 172.17.0.2
```

Figure 6.9: Routing inside the container one

   This output tells us that all the traffic to the gateway at **172.17.0.1** is routed through the **eth0** device.

8. Open yet another terminal window and run a second Alpine container interactively and call it **two**:

9. **docker container run -it --name two alpine:latest ash**

10. Use the **ip** tool in the second container to inspect its network stack as you did for container **one**. Make sure that its IP address is **172.17.0.3** and the Mac address is the one you expect from inspecting container **two**.

11. Now, let us inspect the bridge network and confirm that both containers are attached to it. In your second terminal window, execute the following:

    ```
    docker network inspect bridge
    ```

and you should see this:

```
"Containers": {
    "8b9d4266b9acc05e8c763c04a31dc556549c6e25ba211ef92bb28a07d0f3d97e": {
        "Name": "two",
        "EndpointID": "5117d228c7570ab51b6d941f2c19d2ae07c5cd11a3b54e857391bb4e25538162",
        "MacAddress": "02:42:ac:11:00:03",
        "IPv4Address": "172.17.0.3/16",
        "IPv6Address": ""
    },
    "a22543519beef1b6d0ad3dac5061c71f3279a2318ff55839973556afe0fdb1c7": {
        "Name": "one",
        "EndpointID": "f74849f1b749d1b4d821393205be036b981c20ea417e39cfc851bf3a705fe769",
        "MacAddress": "02:42:ac:11:00:02",
        "IPv4Address": "172.17.0.2/16",
        "IPv6Address": ""
    }
},
```

Figure 6.10: Containers attached to the default bridge network

Here, we are only showing the content of the Containers node since the output of the inspect command is pretty long. We can indeed see that containers **one** and **two** are attached to the network.

In the next section, we are going to discuss name resolution when using the bridge network.

DNS name resolution on a bridge network

Now, let us see how we can reach a second container on the same network from within the first container:

1. From within container one, try to ping container two:

   ```
   ping two
   ```

   Unfortunately, this does not seem to work. The resulting output on your screen should be:

   ```
   ping: bad address 'two'
   ```

2. Now, try to ping container one from within container two and use the following:

   ```
   ping one
   ```

   Same result here:

   ```
   ping: bad address 'one'
   ```

   Evidently, DNS name resolution does not work for the default bridge network. Let us try to fix that by creating a custom bridge network.

and you should see this:

```
"Containers": {
    "8b9d4266b9acc05e8c763c04a31dc556549c6e25ba211ef92bb28a07d0f3d97e": {
        "Name": "two",
        "EndpointID": "5117d228c7570ab51b6d941f2c19d2ae07c5cd11a3b54e857391bb4e25538162",
        "MacAddress": "02:42:ac:11:00:03",
        "IPv4Address": "172.17.0.3/16",
        "IPv6Address": ""
    },
    "a22543519beef1b6d0ad3dac5061c71f3279a2318ff55839973556afe0fdb1c7": {
        "Name": "one",
        "EndpointID": "f74849f1b749d1b4d821393205be036b981c20ea417e39cfc851bf3a705fe769",
        "MacAddress": "02:42:ac:11:00:02",
        "IPv4Address": "172.17.0.2/16",
        "IPv6Address": ""
    }
},
```

Figure 6.10: Containers attached to the default bridge network

Here, we are only showing the content of the Containers node since the output of the inspect command is pretty long. We can indeed see that containers **one** and **two** are attached to the network.

In the next section, we are going to discuss name resolution when using the bridge network.

DNS name resolution on a bridge network

Now, let us see how we can reach a second container on the same network from within the first container:

1. From within container one, try to ping container two:

   ```
   ping two
   ```

   Unfortunately, this does not seem to work. The resulting output on your screen should be:

   ```
   ping: bad address 'two'
   ```

2. Now, try to ping container one from within container two and use the following:

   ```
   ping one
   ```

   Same result here:

   ```
   ping: bad address 'one'
   ```

   Evidently, DNS name resolution does not work for the default bridge network. Let us try to fix that by creating a custom bridge network.

3. In the second terminal, use the following command to create a new instance of type bridge network called `test-network` using the following command:

   ```
   docker network create test-network
   ```

   This will create a new bridge network (which is the default type if not specified explicitly)

4. Attach container one to this (additional) network with the following:

   ```
   docker network connect test-network one
   ```

5. Do the same for container two:

   ```
   docker network connect test-network two
   ```

6. Now, from withing the container `one` runs the ping command again to test if DNS name resolution works. And indeed, we should see something like the following:

   ```
   / # ping two -c3
   PING two (172.28.0.3): 56 data bytes
   64 bytes from 172.28.0.3: seq=0 ttl=64 time=0.252 ms
   64 bytes from 172.28.0.3: seq=1 ttl=64 time=0.386 ms
   64 bytes from 172.28.0.3: seq=2 ttl=64 time=0.352 ms

   --- two ping statistics ---
   3 packets transmitted, 3 packets received, 0% packet loss
   round-trip min/avg/max = 0.252/0.330/0.386 ms
   / #
   ```

   Figure 6.11: DNS name resolution working with custom bridge network

7. Verify that you can also ping container `one` from withing container `two`.
8. Exit containers one and two by pressing CTRL-d.
9. Remove the custom network and the containers with the following:

   ```
   docker container rm one two
   ```

   ```
   docker network rm test-network
   ```

Now, let us talk about container ports and how they can be mapped to host ports.

Elucidating container ports

Often, we want to containerize applications that are listening for incoming requests on one or even multiple ports. Typical examples are Web applications. A Web server hosting a Web application usually listens at port 80 for (unencrypted) HTTP requests and on port 443 for HTTPS requests. But we can also have other types of applications, such as an SSH server that typically listens on port 20.

3. In the second terminal, use the following command to create a new instance of type bridge network called `test-network` using the following command:

   ```
   docker network create test-network
   ```

   This will create a new bridge network (which is the default type if not specified explicitly)

4. Attach container one to this (additional) network with the following:

   ```
   docker network connect test-network one
   ```

5. Do the same for container two:

   ```
   docker network connect test-network two
   ```

6. Now, from withing the container `one` runs the ping command again to test if DNS name resolution works. And indeed, we should see something like the following:

   ```
   / # ping two -c3
   PING two (172.28.0.3): 56 data bytes
   64 bytes from 172.28.0.3: seq=0 ttl=64 time=0.252 ms
   64 bytes from 172.28.0.3: seq=1 ttl=64 time=0.386 ms
   64 bytes from 172.28.0.3: seq=2 ttl=64 time=0.352 ms

   --- two ping statistics ---
   3 packets transmitted, 3 packets received, 0% packet loss
   round-trip min/avg/max = 0.252/0.330/0.386 ms
   / #
   ```

   Figure 6.11: DNS name resolution working with custom bridge network

7. Verify that you can also ping container `one` from withing container `two`.
8. Exit containers one and two by pressing CTRL-d.
9. Remove the custom network and the containers with the following:

   ```
   docker container rm one two
   ```

   ```
   docker network rm test-network
   ```

Now, let us talk about container ports and how they can be mapped to host ports.

Elucidating container ports

Often, we want to containerize applications that are listening for incoming requests on one or even multiple ports. Typical examples are Web applications. A Web server hosting a Web application usually listens at port 80 for (unencrypted) HTTP requests and on port 443 for HTTPS requests. But we can also have other types of applications, such as an SSH server that typically listens on port 20.

If we author such an application and package it and run it inside a container, then we have a problem. We have learned that a container completely shields any process running within it is completely shielded from the outside world. The container presents an impermeable firewall to the containerized application. What can we do? Well, we need the means to get controlled access from the outside world to the port on which the containerized application is listening on. Let us try to illustrate that:

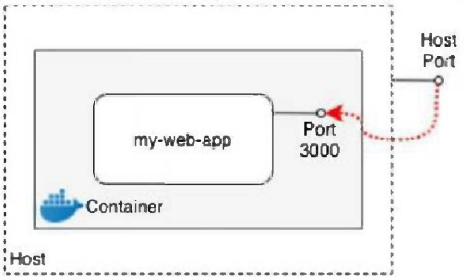


***Figure 6.12:** Opening the door to a defined port*

The preceding graphic illustrates how we have to open a door in the impermeable wall that the container puts up around the hosted application called `my-web-app` that is listening on port 3000. We need to somehow provide access from the host port to port 3000 inside the container. This is indicated by the red dashed line. The application running inside the container and listening on port 3000 could be a .NET WebApi written in C#, such as the one we are going to create in a bit.

Auto mapping ports

The Docker run command provides a means to open such a controlled gateway from a host port to a container port. Let us assume we have built a custom Docker image `my-web-app:1.0` containing our Web application as shown in the preceding picture; then, we can run a container from this image and, at the same time, open the door to the container port 3000 with the following command:

```
docker container run --detach -P my-web-app:1.0
```

This will run the container in detach mode and create a mapping to a free host port, and thus, avoid mapping conflicts. Docker will map the exposed ports in the container to a random port between `49153` and `65535` on the host machine. Which exact port Docker will choose depends on the circumstances and whether or not some of the ports in that range are already taken on the particular host.

Let us test that by using Nginx, a popular Web server and reverse proxy. By default, Nginx listens at port 80. Let us first run an instance of the latest Alpine-based version of Nginx. Open a new terminal window and run the following command:

```
docker container run --detach --name nginx1 nginx:alpine
```

If we author such an application and package it and run it inside a container, then we have a problem. We have learned that a container completely shields any process running within it is completely shielded from the outside world. The container presents an impermeable firewall to the containerized application. What can we do? Well, we need the means to get controlled access from the outside world to the port on which the containerized application is listening on. Let us try to illustrate that:

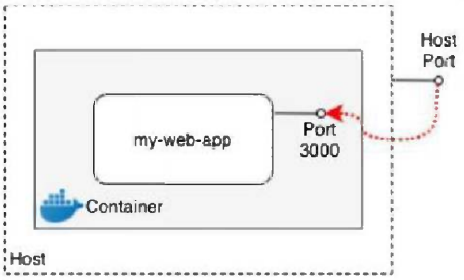


***Figure 6.12:** Opening the door to a defined port*

The preceding graphic illustrates how we have to open a door in the impermeable wall that the container puts up around the hosted application called `my-web-app` that is listening on port 3000. We need to somehow provide access from the host port to port 3000 inside the container. This is indicated by the red dashed line. The application running inside the container and listening on port 3000 could be a .NET WebApi written in C#, such as the one we are going to create in a bit.

Auto mapping ports

The Docker run command provides a means to open such a controlled gateway from a host port to a container port. Let us assume we have built a custom Docker image `my-web-app:1.0` containing our Web application as shown in the preceding picture; then, we can run a container from this image and, at the same time, open the door to the container port 3000 with the following command:

```
docker container run --detach -P my-web-app:1.0
```

This will run the container in detach mode and create a mapping to a free host port, and thus, avoid mapping conflicts. Docker will map the exposed ports in the container to a random port between `49153` and `65535` on the host machine. Which exact port Docker will choose depends on the circumstances and whether or not some of the ports in that range are already taken on the particular host.

Let us test that by using Nginx, a popular Web server and reverse proxy. By default, Nginx listens at port 80. Let us first run an instance of the latest Alpine-based version of Nginx. Open a new terminal window and run the following command:

```
docker container run --detach --name nginx1 nginx:alpine
```

Now try to access Nginx using the curl tool:

```
curl -X GET http://localhost:80
```

You should get the following response:

```
gabriel@Solaris ~ % curl -X GET http://localhost:80
curl: (7) Failed to connect to localhost port 80: Connection refused
```

Figure 6.13: Fail to access Nginx running in the container from outside

OK, that was expected since we know that the container represents an impermeable firewall to all calls from outside of the container. But let us prove that Nginx is indeed listening at port 80. We can use exec into the running container named nginx1, install the curl tool and execute the preceding curl command from within the container:

1. To run an interactive shell inside the Nginx container, use the following command:

   ```
   docker container exec -it nginx1 /bin/sh
   ```

2. Then use the following command to install the curl tool inside the container:

   ```
   apk update && apk add curl
   ```

3. Execute the `curl` command inside the container:

   ```
   curl -X GET http://localhost:80
   ```

   and observe the output in your terminal:

```
/ # curl -X GET http://localhost:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
```

Figure 6.14: Welcome page of Nginx

4. Exit the Nginx container by pressing CTRL-d.

This is proof that Nginx is indeed running and listening on port 80.

Now, let us use what we have learned about mapping a host port to the container port. We use the following command to do so:

Now try to access Nginx using the curl tool:

```
curl -X GET http://localhost:80
```

You should get the following response:

```
gabriel@Solaris ~ % curl -X GET http://localhost:80
curl: (7) Failed to connect to localhost port 80: Connection refused
```

Figure 6.13: Fail to access Nginx running in the container from outside

OK, that was expected since we know that the container represents an impermeable firewall to all calls from outside of the container. But let us prove that Nginx is indeed listening at port 80. We can use exec into the running container named nginx1, install the curl tool and execute the preceding curl command from within the container:

1. To run an interactive shell inside the Nginx container, use the following command:

   ```
   docker container exec -it nginx1 /bin/sh
   ```

2. Then use the following command to install the curl tool inside the container:

   ```
   apk update && apk add curl
   ```

3. Execute the `curl` command inside the container:

   ```
   curl -X GET http://localhost:80
   ```

   and observe the output in your terminal:

```
/ # curl -X GET http://localhost:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
```

Figure 6.14: Welcome page of Nginx

4. Exit the Nginx container by pressing CTRL-d.

This is proof that Nginx is indeed running and listening on port 80.

Now, let us use what we have learned about mapping a host port to the container port. We use the following command to do so:

```
docker container run --detach --name nginx2 -P nginx:alpine
```

Notice the `-P` in the command and also the fact that we name this container **`nginx2`**. Now, how can we find out which host port has been mapped to container port 80? Well, luckily, Docker offers us the ports command. Try to execute the following command:

```
docker container port nginx2
```

and you should see something like this:

```
gabriel@Solaris ~ % docker container port nginx2
80/tcp -> 0.0.0.0:55000
80/tcp -> :::55000
```

Figure 6.15: *Displaying the port mappings of the Nginx container*

As we can see, Docker mapped container port 80 to host port 55000. We can see two mappings, one for the IP4 and one for the IP6 protocol. You may now ask, what does the **`/tcp`** in the preceding mean. Well, the mapping is limited to TCP communication. UDP-style communication would not be allowed by these mappings.

If you run another container and then execute the port command, you will notice that the host port selected will be 55001. Docker is just taking random free ports between 49153 and 65535.

The range of 49153 to 65535 is a commonly used range for dynamic or private ports on most platforms, including Linux and Windows. This range is used to avoid conflicts with well-known ports, which are ports numbered from 0 to 1023 and are typically reserved for system or privileged services.

However, it is important to note that the specific range used for dynamic or private ports can vary depending on the platform. Some platforms might use different ranges; for example, Windows uses the range 49152 to 65535 as dynamic ports.

Additionally, some platforms might have different policies for allocating ports; for example, some Linux distributions use the range 32768 to 61000.

It is also important to note that the range of 49153 and 65535 is a default range for Docker, but you can configure it to use a different range if needed by editing the **/usr/lib/systemd/system/docker.service** file and adding the options **--userland-proxy-path** and **--userland-proxy-bind-address** to the **ExecStart** directive.

Also, it is worth mentioning that the range of 49153 to 65535 is just a convention; any range can be used as long as it is not in conflict with the well-known ports or in use by another service.

```
docker container run --detach --name nginx2 -P nginx:alpine
```

Notice the `-P` in the command and also the fact that we name this container **`nginx2`**. Now, how can we find out which host port has been mapped to container port 80? Well, luckily, Docker offers us the ports command. Try to execute the following command:

```
docker container port nginx2
```

and you should see something like this:

```
gabriel@Solaris ~ % docker container port nginx2
80/tcp -> 0.0.0.0:55000
80/tcp -> :::55000
```

Figure 6.15: *Displaying the port mappings of the Nginx container*

As we can see, Docker mapped container port 80 to host port 55000. We can see two mappings, one for the IP4 and one for the IP6 protocol. You may now ask, what does the **`/tcp`** in the preceding mean. Well, the mapping is limited to TCP communication. UDP-style communication would not be allowed by these mappings.

If you run another container and then execute the port command, you will notice that the host port selected will be 55001. Docker is just taking random free ports between 49153 and 65535.

The range of 49153 to 65535 is a commonly used range for dynamic or private ports on most platforms, including Linux and Windows. This range is used to avoid conflicts with well-known ports, which are ports numbered from 0 to 1023 and are typically reserved for system or privileged services.

However, it is important to note that the specific range used for dynamic or private ports can vary depending on the platform. Some platforms might use different ranges; for example, Windows uses the range 49152 to 65535 as dynamic ports.

Additionally, some platforms might have different policies for allocating ports; for example, some Linux distributions use the range 32768 to 61000.

It is also important to note that the range of 49153 and 65535 is a default range for Docker, but you can configure it to use a different range if needed by editing the **/usr/lib/systemd/system/docker.service** file and adding the options **--userland-proxy-path** and **--userland-proxy-bind-address** to the **ExecStart** directive.

Also, it is worth mentioning that the range of 49153 to 65535 is just a convention; any range can be used as long as it is not in conflict with the well-known ports or in use by another service.

Let us try to access Nginx now from the host and see if we are successful:

```
curl -X GET http://localhost:55000
```

Note how we have to use port 55000 (and not port 80) in the preceding URL since Docker mapped container port 80 to host port 55000.

This time you should be successful, and the HTML of the Nginx welcome page should be returned. You can also test this with your browser. Open a new browser tab and navigate to the URL **`http://localhost:55000`**, and you should see this:



Figure 6.16: Welcome page of Nginx shown in the browser

Explicitly mapping ports

So far, we have let Docker decide how to map a container port to a free host port. At times we want to have more control over this process and map a container port to a fixed host port. The Docker CLI is flexible enough to allow us to do so. Instead of using **`-P`** as we did before, we will use the lowercase p, that is **`-p`** (or **`--port`**), to explicitly map a port. If we want to map container port 80 used by Nginx inside the container to host port 3000, then the Docker run command has to look like this:

```
docker container run --detach --name nginx3 -p 3000:80 nginx:alpine
```

If we then execute the port command, we will see this:

```
gabriel@Solaris ~ % docker container port nginx3
80/tcp -> 0.0.0.0:3000
80/tcp -> :::3000
```

Figure 6.17: Explicit mapping of a container port to a host port

And now, we can test the mapping with this curl command:

```
curl http:localhost:3000
```

and we should receive the HTML of the Nginx welcome page as an answer.

Let us try to access Nginx now from the host and see if we are successful:

```
curl -X GET http://localhost:55000
```

Note how we have to use port 55000 (and not port 80) in the preceding URL since Docker mapped container port 80 to host port 55000.

This time you should be successful, and the HTML of the Nginx welcome page should be returned. You can also test this with your browser. Open a new browser tab and navigate to the URL **`http://localhost:55000`**, and you should see this:



Figure 6.16: Welcome page of Nginx shown in the browser

Explicitly mapping ports

So far, we have let Docker decide how to map a container port to a free host port. At times we want to have more control over this process and map a container port to a fixed host port. The Docker CLI is flexible enough to allow us to do so. Instead of using **`-P`** as we did before, we will use the lowercase p, that is **`-p`** (or **`--port`**), to explicitly map a port. If we want to map container port 80 used by Nginx inside the container to host port 3000, then the Docker run command has to look like this:

```
docker container run --detach --name nginx3 -p 3000:80 nginx:alpine
```

If we then execute the port command, we will see this:

```
gabriel@Solaris ~ % docker container port nginx3
80/tcp -> 0.0.0.0:3000
80/tcp -> :::3000
```

Figure 6.17: Explicit mapping of a container port to a host port

And now, we can test the mapping with this curl command:

```
curl http:localhost:3000
```

and we should receive the HTML of the Nginx welcome page as an answer.

Creating and running a containerized REST API

Let us now build our own little REST API and containerize it. For this, we will use .NET.

1. Make sure you have .NET installed on your local computer:
   a. On your Mac, use the following command to install the latest .NET version:

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

   b. Alternatively, on your Windows 10 or 11 computer, use Chocolatey to install the .NET SDK version 5.0.x with the following:

      ```
      choco install dotnet-5.0-sdk
      ```

   c. Once the .NET SDK has been successfully installed on either system, double-check it with

      ```
      dotnet -version
      ```

      which in our case results in an output of:

      ```
      5.0.302
      ```

2. Create a new folder for the source code and navigate to it:

   ```
   mkdir -p src/ch06/ordering && cd src/ch06/ordering
   ```

3. Use the .NET command line tool to create the skeleton for a new WEB API as follows:

   ```
   dotnet new webapi -o ordering
   ```

4. Run the application with:

   ```
   dotnet run
   ```

   In a new browser tab, navigate to the following URL:

   https://localhost:5001/weatherforecast

   you should see something like this:

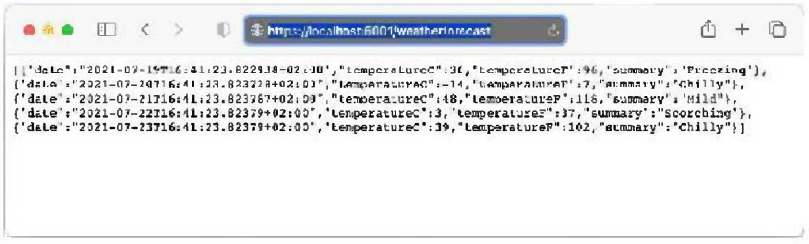


Figure 6.18: Calling the weather forecast endpoint

Creating and running a containerized REST API

Let us now build our own little REST API and containerize it. For this, we will use .NET.

1. Make sure you have .NET installed on your local computer:
   a. On your Mac, use the following command to install the latest .NET version:

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

   b. Alternatively, on your Windows 10 or 11 computer, use Chocolatey to install the .NET SDK version 5.0.x with the following:

      ```
      choco install dotnet-5.0-sdk
      ```

   c. Once the .NET SDK has been successfully installed on either system, double-check it with

      ```
      dotnet --version
      ```

      which in our case results in an output of:

      ```
      5.0.302
      ```

2. Create a new folder for the source code and navigate to it:

   ```
   mkdir -p src/ch06/ordering && cd src/ch06/ordering
   ```

3. Use the .NET command line tool to create the skeleton for a new WEB API as follows:

   ```
   dotnet new webapi -o ordering
   ```

4. Run the application with:

   ```
   dotnet run
   ```

   In a new browser tab, navigate to the following URL:

   https://localhost:5001/weatherforecast

   you should see something like this:



Figure 6.18: Calling the weather forecast endpoint

5. Open the ordering folder in Visual Studio Code with
   `code ordering/`

   you should see something like this:

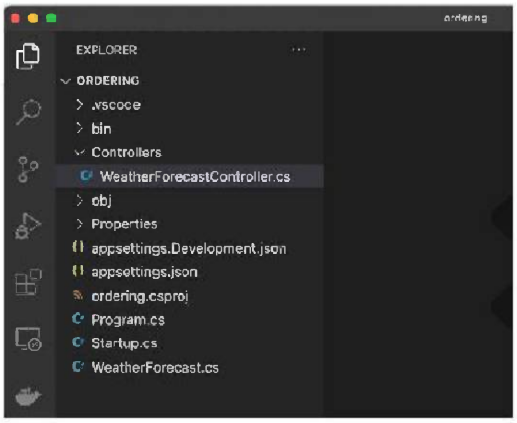


Figure 6.19: The skeleton REST API in VS Code

6. To the root of the project, add a new file called **`Dockerfile`** and add this content:

```
Dockerfile ×
Dockerfile > ...
1   FROM mcr.microsoft.com/dotnet/sdk:5.0 AS builder
2   WORKDIR /app
3   COPY . .
4   RUN dotnet build -c Release -o output
5
6   FROM mcr.microsoft.com/dotnet/aspnet:5.0
7   COPY --from=builder /app/output /app
8   CMD ["dotnet", "/app/ordering.dll"]
9
```

Figure 6.20: Multi-stage Dockerfile to build a Docker image for the REST API

This will be our manifest to build a custom Docker image. In Line 1, we use the .NET SDK image version 5 as our builder stage. In Line 3, we copy all source files into the **`/app`** folder in the container, ready to be used for compiling by the SDK. In line 4 of the Dockerfile, you see the command used to build a release version of the service into the target folder **`output`**. In Line 6, we declare that we want to use the ASP.NET runtime image for the final stage, which will then be our custom image. In Line 7, we copy the build results from the builder stage into our final stage, and in line 8, we define the start command to use when running a container from this image.

5. Open the ordering folder in Visual Studio Code with
   `code ordering/`

   you should see something like this:

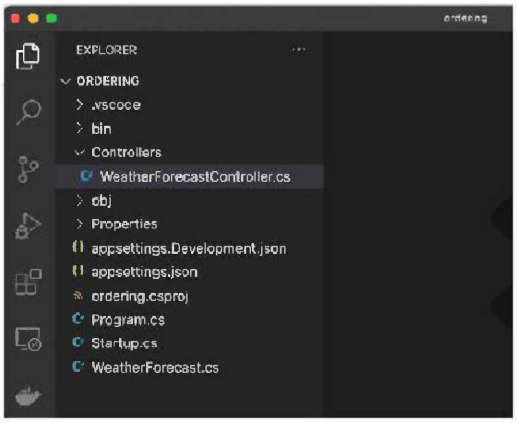


Figure 6.19: The skeleton REST API in VS Code

6. To the root of the project, add a new file called **`Dockerfile`** and add this content:

```
Dockerfile ×
Dockerfile > ...
1   FROM mcr.microsoft.com/dotnet/sdk:5.0 AS builder
2   WORKDIR /app
3   COPY . .
4   RUN dotnet build -c Release -o output
5
6   FROM mcr.microsoft.com/dotnet/aspnet:5.0
7   COPY --from=builder /app/output /app
8   CMD ["dotnet", "/app/ordering.dll"]
9
```

Figure 6.20: Multi-stage Dockerfile to build a Docker image for the REST API

This will be our manifest to build a custom Docker image. In Line 1, we use the .NET SDK image version 5 as our builder stage. In Line 3, we copy all source files into the **`/app`** folder in the container, ready to be used for compiling by the SDK. In line 4 of the Dockerfile, you see the command used to build a release version of the service into the target folder **`output`**. In Line 6, we declare that we want to use the ASP.NET runtime image for the final stage, which will then be our custom image. In Line 7, we copy the build results from the builder stage into our final stage, and in line 8, we define the start command to use when running a container from this image.

7. Build the Docker image with the following:
   ```
   docker image build -t ordering .
   ```
   This will build an image called ordering with tag `latest`.

8. Run a container from that image with the following:
   ```
   docker container run --rm -p 5000:80 ordering
   ```
   you should see this in the output:

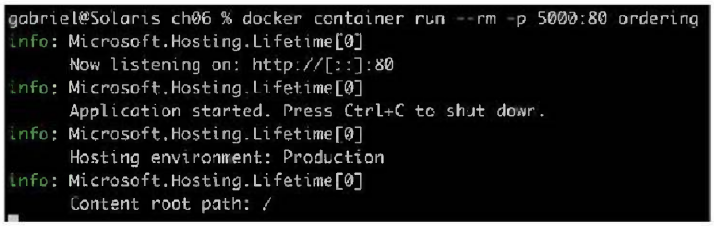


Figure 6.21: Running the REST API in a container

   As you can see in the preceding output, the service is listening at port 80 (indicated by the `http://[::]:80`). But this is the port used by the application inside the container. In our Docker run the preceding command, we map this container port 80 to the host port 5000 using the parameter `-p 5000:80`.

9. Open a new browser tab and navigate to `http://localhost:5000/weatherforecast`

   You should get the same kind of output as in preceding Point 5.

Now that we have a good overview of what the bridge network is and what its capabilities are let us concentrate on the remaining network types.

Discussing the none and host networks

Other than the bridge network Docker provides two more implementations of the CNM that can come in handy for very specific use cases. The implementations packaged with Docker are the `host` and the `none` network. Let us start with the former.

The host network

There are occasions where we want to run a container in the network namespace of the host. This can be necessary when we need to run some software in a container that is used to analyze or debug the host networks' traffic.

7. Build the Docker image with the following:
   ```
   docker image build -t ordering .
   ```
   This will build an image called ordering with tag `latest`.

8. Run a container from that image with the following:
   ```
   docker container run --rm -p 5000:80 ordering
   ```
   you should see this in the output:

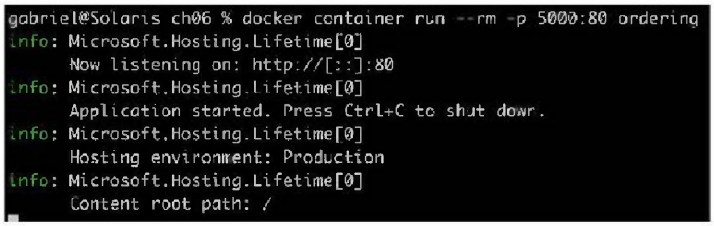


Figure 6.21: Running the REST API in a container

   As you can see in the preceding output, the service is listening at port 80 (indicated by the `http://[::]:80`). But this is the port used by the application inside the container. In our Docker run the preceding command, we map this container port 80 to the host port 5000 using the parameter `-p 5000:80`.

9. Open a new browser tab and navigate to `http://localhost:5000/weatherforecast`

   You should get the same kind of output as in preceding Point 5.

Now that we have a good overview of what the bridge network is and what its capabilities are let us concentrate on the remaining network types.

Discussing the none and host networks

Other than the bridge network Docker provides two more implementations of the CNM that can come in handy for very specific use cases. The implementations packaged with Docker are the `host` and the `none` network. Let us start with the former.

The host network

There are occasions where we want to run a container in the network namespace of the host. This can be necessary when we need to run some software in a container that is used to analyze or debug the host networks' traffic.

But please keep in mind that this is a dangerous thing and should only be used in a very limited way. There is no good reason why business software that runs in a container should ever have access to the host's network stack. It has to be limited to special purposes.

As always, let us demonstrate how we can run a container with access to the host's network namespace and discuss what this means:

1. Open a new terminal window
2. Run an Alpine container named `my-tool` interactively and attach it to the host network using the following command:

```
docker container run -it \
    --name my-tool \
    --network host \
    alpine:latest sh
```

3. Analyze the network namespace from within the container using our beloved ip tool:

```
ip addr show eth0
```

and notice the output such as the following:

```
/ # ip addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 02:50:00:00:00:01 brd ff:ff:ff:ff:ff:ff
    inet 192.168.65.6/24 brd 192.168.65.255 scope global dynamic noprefixroute eth0
       valid_lft 3495sec preferred_lft 2055sec
    inet6 fe80::50:ff:fe00:1/64 scope link
       valid_lft forever preferred_lft forever
/ #
```

Figure 6.22: Network setting from within the container when using the host network

We can see the MacAddress and the IP address `192.168.65.6` that have been assigned to this endpoint, and they correspond to the ones of the host

4. We can also analyze the routing that has been defined for the container with

```
ip route
```

and we see something like the following:

But please keep in mind that this is a dangerous thing and should only be used in a very limited way. There is no good reason why business software that runs in a container should ever have access to the host's network stack. It has to be limited to special purposes.

As always, let us demonstrate how we can run a container with access to the host's network namespace and discuss what this means:

1. Open a new terminal window
2. Run an Alpine container named `my-tool` interactively and attach it to the host network using the following command:

```
docker container run -it \
    --name my-tool \
    --network host \
    alpine:latest sh
```

3. Analyze the network namespace from within the container using our beloved ip tool:

```
ip addr show eth0
```

and notice the output such as the following:

```
/ # ip addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 02:50:00:00:00:01 brd ff:ff:ff:ff:ff:ff
    inet 192.168.65.6/24 brd 192.168.65.255 scope global dynamic noprefixroute eth0
       valid_lft 3495sec preferred_lft 2055sec
    inet6 fe80::50:ff:fe00:1/64 scope link
       valid_lft forever preferred_lft forever
/ #
```

Figure 6.22: Network setting from within the container when using the host network

We can see the MacAddress and the IP address `192.168.65.6` that have been assigned to this endpoint, and they correspond to the ones of the host

4. We can also analyze the routing that has been defined for the container with

```
ip route
```

and we see something like the following:

```
/ # ip route
default via 192.168.65.1 dev eth0  src 192.168.65.6  metric 202
127.0.0.0/8 dev lo scope host
172.17.0.0/16 dev docker0 scope link  src 172.17.0.1
172.27.0.0/16 dev br-8b1ee9c16bcb scope link  src 172.27.0.1
192.168.65.0/24 dev eth0 scope link  src 192.168.65.6  metric 202
192.168.65.5 dev services1 scope link  src 192.168.65.4
```

Figure 6.23: Routing inside a container attached to the host network

We can see that any traffic is routed inside the container the same way as on the host, which is expected since the container uses the same network namespace as the host.

Before we continue, it should be pointed out once again that it is dangerous to use the host network, and it should never be used for line of business applications.

The none network

Sometimes one wants to run a job inside a container that has absolutely no other dependencies and does not need any network access at all. In such situations, it is best practice to attach this container to the null network. This way, all processes running inside the container are perfectly shielded from external access. Let us proceed with an example:

1. To run a container such as an Alpine container attached to this network, execute the following command:

   ```
   docker container run -it \
      --name test \
      --network none \
      alpine:latest sh
   ```

2. From withing the started container, try to use the ip tool we already know from earlier experiments and run the following command:

   ```
   ip addr show eth0
   ```

   The resulting output should be the following:

   ```
   ip: can't find device 'eth0'
   ```

   confirming that, indeed, there is no network available in this container.

3. Exit the container with CTRL-d. The container should still remain in memory, though in status exited.

```
/ # ip route
default via 192.168.65.1 dev eth0  src 192.168.65.6  metric 202
127.0.0.0/8 dev lo scope host
172.17.0.0/16 dev docker0 scope link  src 172.17.0.1
172.27.0.0/16 dev br-8b1ee9c16bcb scope link  src 172.27.0.1
192.168.65.0/24 dev eth0 scope link  src 192.168.65.6  metric 202
192.168.65.5 dev services1 scope link  src 192.168.65.4
```

Figure 6.23: Routing inside a container attached to the host network

We can see that any traffic is routed inside the container the same way as on the host, which is expected since the container uses the same network namespace as the host.

Before we continue, it should be pointed out once again that it is dangerous to use the host network, and it should never be used for line of business applications.

The none network

Sometimes one wants to run a job inside a container that has absolutely no other dependencies and does not need any network access at all. In such situations, it is best practice to attach this container to the null network. This way, all processes running inside the container are perfectly shielded from external access. Let us proceed with an example:

1. To run a container such as an Alpine container attached to this network, execute the following command:

   ```
   docker container run -it \
      --name test \
      --network none \
      alpine:latest sh
   ```

2. From withing the started container, try to use the ip tool we already know from earlier experiments and run the following command:

   ```
   ip addr show eth0
   ```

   The resulting output should be the following:

   ```
   ip: can't find device 'eth0'
   ```

   confirming that, indeed, there is no network available in this container.

3. Exit the container with CTRL-d. The container should still remain in memory, though in status exited.

4. Inspect the container's network settings with this command:

   ```
   docker container inspect test
   ```

   And you should see something similar to this:



Figure 6.24: Inspecting network settings of a container attached to none network

In the preceding image, we see the network settings part of the overall output of the container inspect command. And the container is indeed attached to the **none** network. Consequently, it has no MacAddress and no IP address defined.

5. Clean up your environment with the following command:

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

   What do the preceding commands exactly do? If you cannot decipher them, please use online help or Google for it.

With this, we have reached the end of this chapter about Docker container networking.

Conclusion

In this chapter, we have been introduced to the Docker **container network model (CNM)**. A CNM describes how a **software-defined network (SDN)** connecting multiple Docker containers has to look like and what requirements it has to fulfil. We then discussed the three implementations of the CNM in detail that are part of the default installation of Docker on our machines. We put a lot of emphasis on describing the bridge network, which is used to connect multiple containers running on a single Docker host and shielding them from the outside world.

4. Inspect the container's network settings with this command:

   ```
   docker container inspect test
   ```

   And you should see something similar to this:



Figure 6.24: Inspecting network settings of a container attached to none network

In the preceding image, we see the network settings part of the overall output of the container inspect command. And the container is indeed attached to the **none** network. Consequently, it has no MacAddress and no IP address defined.

5. Clean up your environment with the following command:

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

   What do the preceding commands exactly do? If you cannot decipher them, please use online help or Google for it.

With this, we have reached the end of this chapter about Docker container networking.

Conclusion

In this chapter, we have been introduced to the Docker **container network model (CNM)**. A CNM describes how a **software-defined network (SDN)** connecting multiple Docker containers has to look like and what requirements it has to fulfil. We then discussed the three implementations of the CNM in detail that are part of the default installation of Docker on our machines. We put a lot of emphasis on describing the bridge network, which is used to connect multiple containers running on a single Docker host and shielding them from the outside world.

Questions

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

1. Name the three main elements described or named in Docker's container network model (CNM).
2. What is the primary reason to use networks with containers?
3. Which three types of networks are supported out of the box by Docker?
4. What is an important difference between the default bridge network on a Docker host and an explicitly created instance of the bridge network?
5. What can you do to provide the outside world access to a Web application running in a container and listening on container port 3000?
6. How can I run a service in a container such that the service has full access to the network stack of the host machine?

Answers

Here is the list of correct answers:

1. The CNM describes or names the network sandbox, the networks and the endpoints as fundamental constituent elements of the model.
2. When attaching containers to networks, they are able to communicate with each other. Containers are otherwise perfectly shielded from the outside world. In that sense, networks present a perfect firewall shielding the processes or applications running inside containers from external access.
3. The three network drivers that come out of the box with Docker are the **`null`** network, the **`host`** network, and the **`bridge`** network.
4. The default instance of the bridge network does not support name resolution. That is, containers attached to this network cannot use DNS to localize each other. An instance of type bridge explicitly created, on the other hand, does support DNS name resolution.
5. In the **`docker container run`** command, we can use the **`--port`** (or short **`-p`**) parameter to map container port 3000 to any free host port such as, e.g., port 8080:

```
docker container run \
    --detach \
    --port 8080:3000 \
    dockerforjobseekers/web:1.0
```

Questions

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

1. Name the three main elements described or named in Docker's container network model (CNM).
2. What is the primary reason to use networks with containers?
3. Which three types of networks are supported out of the box by Docker?
4. What is an important difference between the default bridge network on a Docker host and an explicitly created instance of the bridge network?
5. What can you do to provide the outside world access to a Web application running in a container and listening on container port 3000?
6. How can I run a service in a container such that the service has full access to the network stack of the host machine?

Answers

Here is the list of correct answers:

1. The CNM describes or names the network sandbox, the networks and the endpoints as fundamental constituent elements of the model.
2. When attaching containers to networks, they are able to communicate with each other. Containers are otherwise perfectly shielded from the outside world. In that sense, networks present a perfect firewall shielding the processes or applications running inside containers from external access.
3. The three network drivers that come out of the box with Docker are the **`null`** network, the **`host`** network, and the **`bridge`** network.
4. The default instance of the bridge network does not support name resolution. That is, containers attached to this network cannot use DNS to localize each other. An instance of type bridge explicitly created, on the other hand, does support DNS name resolution.
5. In the **`docker container run`** command, we can use the **`--port`** (or short **`-p`**) parameter to map container port 3000 to any free host port such as, e.g., port 8080:

```
docker container run \
    --detach \
    --port 8080:3000 \
    dockerforjobseekers/web:1.0
```

6. If a service that is running inside a container needs access to the host's network stack, then one should attach this container to the host network:

   ```
   docker container run --network host ...
   ```

   Note that this technique is dangerous and should only be used in highly controlled circumstances.

Mastering job interview question

In this section, we will be presenting tips on how to answer possible job interview questions related to container networking. Questions asked during a good job interview are often open-ended to give the candidate an opportunity to show their knowledge and make them talk. This is in contrast to multiple choice questions that you are often presented with during exams.

So, you finally got a chance for an onsite interview for a job as a DevOps engineer. The job is offered by a company that is just starting to work with containers and is planning to host all their software services in the cloud. During the interview, you are confronted with the following questions:

1. We have heard that when working with containers, one can use so-called **software-defined networks (SDNs)**? Can you tell us a bit more about what those SDNs are and how they relate to containers?
2. Docker has defined the **Container Networking Model (CNM)**. What is this definition good for, and what are the key elements of that model?
3. Our developers often run multiple containers on their machines simultaneously. From the documentation, we know that those containers are automatically attached to the (default) bridge network. Technically these containers—or more precisely, the applications running inside the containers—should now be able to communicate with each other, right? In reality, we find that this is not the case. What are we doing wrong?
4. We want to run a special service in a container. This service should intercept and analyze the network traffic happening on the host. Is that even possible? And if yes, how could we do such a thing? Are there any considerations to be taken into account if doing so?
5. We have heard that the containers of a distributed application should be attached to different networks. Why should we not just use a single network and attach all containers to it?

6. If a service that is running inside a container needs access to the host's network stack, then one should attach this container to the host network:

   ```
   docker container run --network host ...
   ```

   Note that this technique is dangerous and should only be used in highly controlled circumstances.

Mastering job interview question

In this section, we will be presenting tips on how to answer possible job interview questions related to container networking. Questions asked during a good job interview are often open-ended to give the candidate an opportunity to show their knowledge and make them talk. This is in contrast to multiple choice questions that you are often presented with during exams.

So, you finally got a chance for an onsite interview for a job as a DevOps engineer. The job is offered by a company that is just starting to work with containers and is planning to host all their software services in the cloud. During the interview, you are confronted with the following questions:

1. We have heard that when working with containers, one can use so-called **software-defined networks (SDNs)**? Can you tell us a bit more about what those SDNs are and how they relate to containers?
2. Docker has defined the **Container Networking Model (CNM)**. What is this definition good for, and what are the key elements of that model?
3. Our developers often run multiple containers on their machines simultaneously. From the documentation, we know that those containers are automatically attached to the (default) bridge network. Technically these containers—or more precisely, the applications running inside the containers—should now be able to communicate with each other, right? In reality, we find that this is not the case. What are we doing wrong?
4. We want to run a special service in a container. This service should intercept and analyze the network traffic happening on the host. Is that even possible? And if yes, how could we do such a thing? Are there any considerations to be taken into account if doing so?
5. We have heard that the containers of a distributed application should be attached to different networks. Why should we not just use a single network and attach all containers to it?

Sample answers to the job interview questions

Here are potential answers to the job interview questions:

1. SDNs are instances of a network driver installed with Docker on a Docker host. There are different types of network types, also called network drivers. Each of those network drivers has to follow the rules of the Docker **Container Network Model (CNM)**. SDNs are like firewalls protecting containers from external access yet allowing services running inside containers that are attached to the same network to communicate with each other.

2. The CNM defines three key elements:

    a. **The network sandbox**: This shields the processes running inside a container from being accessed by any external party. It is like a firewall protecting the container and its content.

    b. **The network**: The network corresponds to the lane or route through which communication can flow.

    c. **The endpoint**: Endpoints are like controlled gateways from a container to a network. A container can be attached to one or many networks through endpoints.

3. The default bridge network on a Docker host does not support DNS name resolution. Thus, applications in different containers running on this network cannot freely communicate. One can mitigate that by explicitly creating a bridge network and attaching the containers to it.

4. Yes, it is indeed possible to provide containerized service access to the host's network stack; for this, one has to attach the container to the so-called *host* network that is defined by default on a Docker host. The problem with this is that it is a double edge sword. This technique should only be used in very controlled environments since if the service is hacked, the hacker gets access to the network stack of the host, which can do a lot of harm.

5. Software-defined networks are cheap to create, and each additional network provides another layer of security to the application. If we take the sample of a simple Web application consisting of a frontend part (*Web*), a business service (called *order-service*) and a database, then one could have two networks called *front* and *back* where the *Web* and the *order-service* are attached to the *front* network. The *database* and the *order-service* are attached to the *back* network. Thus, the *Web* service is not able to communicate with the *database* service, and only the middle tier *order-service* can bridge the traffic between the two networks. Thus, if the *Web* gets hacked, the malicious code has no way to access the *database* service.

Sample answers to the job interview questions

Here are potential answers to the job interview questions:

1. SDNs are instances of a network driver installed with Docker on a Docker host. There are different types of network types, also called network drivers. Each of those network drivers has to follow the rules of the Docker **Container Network Model (CNM)**. SDNs are like firewalls protecting containers from external access yet allowing services running inside containers that are attached to the same network to communicate with each other.
2. The CNM defines three key elements:
    a. **The network sandbox**: This shields the processes running inside a container from being accessed by any external party. It is like a firewall protecting the container and its content.
    b. **The network**: The network corresponds to the lane or route through which communication can flow.
    c. **The endpoint**: Endpoints are like controlled gateways from a container to a network. A container can be attached to one or many networks through endpoints.
3. The default bridge network on a Docker host does not support DNS name resolution. Thus, applications in different containers running on this network cannot freely communicate. One can mitigate that by explicitly creating a bridge network and attaching the containers to it.
4. Yes, it is indeed possible to provide containerized service access to the host's network stack; for this, one has to attach the container to the so-called *host* network that is defined by default on a Docker host. The problem with this is that it is a double edge sword. This technique should only be used in very controlled environments since if the service is hacked, the hacker gets access to the network stack of the host, which can do a lot of harm.
5. Software-defined networks are cheap to create, and each additional network provides another layer of security to the application. If we take the sample of a simple Web application consisting of a frontend part (*Web*), a business service (called *order-service*) and a database, then one could have two networks called *front* and *back* where the *Web* and the *order-service* are attached to the *front* network. The *database* and the *order-service* are attached to the *back* network. Thus, the *Web* service is not able to communicate with the *database* service, and only the middle tier *order-service* can bridge the traffic between the two networks. Thus, if the *Web* gets hacked, the malicious code has no way to access the *database* service.

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



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 7

Managing Complex Apps with Docker Compose

Introduction

In the previous chapter, we introduced the Docker **container network model (CNM)**, and we discussed the default implementations of this model coming included with Docker Desktop. We specifically concentrated on the bridge network driver, which can be used for single-host networking. We also briefly discussed drivers that can span multiple nodes in a cluster of Docker hosts.

In this chapter, we introduce 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 services and the Docker Compose file.

Structure

In this chapter, we will discuss the following topics:

- Explaining the Docker-compose syntax
- Authoring your own Docker-compose file
- Building and running all or individual services
- Discussing advanced usages of Docker compose

CHAPTER 7

Managing Complex Apps with Docker Compose

Introduction

In the previous chapter, we introduced the Docker **container network model** (CNM), and we discussed the default implementations of this model coming included with Docker Desktop. We specifically concentrated on the bridge network driver, which can be used for single-host networking. We also briefly discussed drivers that can span multiple nodes in a cluster of Docker hosts.

In this chapter, we introduce 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 services and the Docker Compose file.

Structure

In this chapter, we will discuss the following topics:

- Explaining the Docker-compose syntax
- Authoring your own Docker-compose file
- Building and running all or individual services
- Discussing advanced usages of Docker compose

Objective

After finishing this chapter, you will learn the structure of a Docker-compose file for an interested layperson. You will be able to write your own Docker-compose file for a multi-container application and build, run, and scale services using Docker compose

Let us right dive into the topics and start by explaining the syntax used in a Docker compose file.

Explaining the Docker-compose syntax

In this section, we will get to know the syntax used for Docker compose files. A compose file can go from being very simple to pretty complex and powerful. We can use compose files to orchestrate single services to many services. Docker compose files are written in a declarative way. What does that mean?

Imperative versus declarative syntax

Before we dive into the innards of the Docker compose syntax, we need to clarify what imperative and what declarative syntax actually means. A Docker compose file, as we will see, is written using the latter, which is declarative syntax.

When we are writing a typical application, say in one of the popular languages such as Python, Java, Go, or C#, then we usually use the imperative style of coding. That is, we write a bunch of code where we tell the runtime step-by-step what we expect it to do. We use commands, conditional statements or loops to determine the exact sequence of what has to happen and how it has to happen. We are totally in control of the path of execution of the application, and that is good.

Sometimes we do not want to specify in detail how a certain task shall be done, but we are only interested in the outcome. We delegate the decision on how to achieve a desired outcome to the algorithm or service at hand, such as a database. There we use SQL to formulate the desired outcome and then hand this SQL statement over to the database to execute in a way that is optimized for the problem we state. We do not tell the database how to calculate the resulting output but only tell it what we want to see. The database then uses some sophisticated algorithms to come up with a good solution.

Similarly, we can write a Docker compose file in declarative syntax and let the Docker engine figure out the details of how to achieve what we have declared that we want, namely, running a bunch of containers that each contain a service that is part of an overall application. We do not have to specify in detail how Docker shall achieve this goal, and we just declare what services or containers we want it to run and how they

Objective

After finishing this chapter, you will learn the structure of a Docker-compose file for an interested layperson. You will be able to write your own Docker-compose file for a multi-container application and build, run, and scale services using Docker compose

Let us right dive into the topics and start by explaining the syntax used in a Docker compose file.

Explaining the Docker-compose syntax

In this section, we will get to know the syntax used for Docker compose files. A compose file can go from being very simple to pretty complex and powerful. We can use compose files to orchestrate single services to many services. Docker compose files are written in a declarative way. What does that mean?

Imperative versus declarative syntax

Before we dive into the innards of the Docker compose syntax, we need to clarify what imperative and what declarative syntax actually means. A Docker compose file, as we will see, is written using the latter, which is declarative syntax.

When we are writing a typical application, say in one of the popular languages such as Python, Java, Go, or C#, then we usually use the imperative style of coding. That is, we write a bunch of code where we tell the runtime step-by-step what we expect it to do. We use commands, conditional statements or loops to determine the exact sequence of what has to happen and how it has to happen. We are totally in control of the path of execution of the application, and that is good.

Sometimes we do not want to specify in detail how a certain task shall be done, but we are only interested in the outcome. We delegate the decision on how to achieve a desired outcome to the algorithm or service at hand, such as a database. There we use SQL to formulate the desired outcome and then hand this SQL statement over to the database to execute in a way that is optimized for the problem we state. We do not tell the database how to calculate the resulting output but only tell it what we want to see. The database then uses some sophisticated algorithms to come up with a good solution.

Similarly, we can write a Docker compose file in declarative syntax and let the Docker engine figure out the details of how to achieve what we have declared that we want, namely, running a bunch of containers that each contain a service that is part of an overall application. We do not have to specify in detail how Docker shall achieve this goal, and we just declare what services or containers we want it to run and how they

are related to each other. We also may specify some configuration values to be used in the process of creating and running those containers.

Using a declarative approach is very important when dealing with containerized applications. Docker compose is just one example where this style is used. The other place where it is used and which you know very well by now is when authoring a Dockerfile. The syntax there is also declarative. In *Chapter 10, Orchestrating Containers*, we will see that when working with a Kubernetes cluster or a Docker swarm, we also use a declarative syntax.

Now, we are ready to discuss the syntax of a simple yet concrete sample Docker compose file.

Inspecting a simple Docker compose file

A Docker compose file is just a simple text document similar to what we have learned with the Dockerfile. The difference here, compared to Dockerfiles, is that in a Docker compose file, we use the YAML syntax.

Please have a look at the following sample compose file:

cocker-compose.yml ×

docker > docker-compose.yml

```
1   version: '2'
2   services:
3     web:
4       image: my-web:1.0
5       ports:
6         - 3000:80
7       environment:
8         connection_string: postgresql://demo-user:Top-Secret@database:5432/demo-db
9     database:
10      image: postgres:alpine
11      environment:
12        POSTGRES_PASSWORD: Top-Secret
13        POSTGRES_USER: demo-user
14        POSTGRES_DATABASE: demo-db
```

Figure 7.1: Simple Docker compose file

By default, a Docker compose file is called `docker-compose.yml`. But you are free to name the file differently. The advantage of using the default name is that you do not have to specify the file name when using the Docker compose tool. Let us now analyze the preceding file in detail:

- In Line 1, we have to declare the version of the Docker compose syntax we want to use. There are currently two major versions supported, 2.x and

are related to each other. We also may specify some configuration values to be used in the process of creating and running those containers.

Using a declarative approach is very important when dealing with containerized applications. Docker compose is just one example where this style is used. The other place where it is used and which you know very well by now is when authoring a Dockerfile. The syntax there is also declarative. In *Chapter 10, Orchestrating Containers*, we will see that when working with a Kubernetes cluster or a Docker swarm, we also use a declarative syntax.

Now, we are ready to discuss the syntax of a simple yet concrete sample Docker compose file.

Inspecting a simple Docker compose file

A Docker compose file is just a simple text document similar to what we have learned with the Dockerfile. The difference here, compared to Dockerfiles, is that in a Docker compose file, we use the YAML syntax.

Please have a look at the following sample compose file:

cocker-compose.yml ×

docker > docker-compose.yml

```
1  version: '2'
2  services:
3    web:
4      image: my-web:1.0
5      ports:
6        - 3000:80
7      environment:
8        connection_string: postgresql://demo-user:Top-Secret@database:5432/demo-db
9    database:
10     image: postgres:alpine
11     environment:
12       POSTGRES_PASSWORD: Top-Secret
13       POSTGRES_USER: demo-user
14       POSTGRES_DATABASE: demo-db
```

Figure 7.1: Simple Docker compose file

By default, a Docker compose file is called `docker-compose.yml`. But you are free to name the file differently. The advantage of using the default name is that you do not have to specify the file name when using the Docker compose tool. Let us now analyze the preceding file in detail:

- In Line 1, we have to declare the version of the Docker compose syntax we want to use. There are currently two major versions supported, 2.x and

3.x. The former is used whenever working on a single node, such as in development when using Docker Desktop or on a build server in a CI/CD pipeline. Version 3.x, on the other hand, is used when working with clusters, as we will see in *Chapter 10, Orchestrating Containers*.

- In Line 2, the `services` keyword declares that now what follows in this section is the list of services making up the application.
- In Lines 3 and 9, we have the names of two `services`, Web and `database`. These two services make up our sample application. The two names are indented to comply with the requirements of the YAML standard. Please be careful with the indentations; if you get this wrong, then you will get errors when using the Docker compose file later on.
- Each service needs information on which Docker image to use. In Line 4, we have the keyword `image:` followed by the name of the image to use. In our case, we use an image called `my-web:1.0` for the Web service.

  In Line 10, we see that the `database` service uses the image `postgres:alpine`.
- Now to Lines 5 and 6: The keyword `ports:` starts a section where we can declare which container ports we want to expose to the host computer. It is the equivalent of using the `-p` parameter in the `docker container run` command. In this example, we open container port 3000 and map it to host port 80.

  It is possible to map more than just one port. Just add another line starting with a hyphen in the `ports` section.

  Note that for the `database` service, we do not need to map a port, and thus, there is no `ports` section defined for this service.
- In Line 7, we have the section `environment:`. It allows us to define environment variables and their respective values for the my-web service. In this particular case, we only define a single variable called `connection_string` and assign it a value.

  In Lines 11–14, we do the same for the `database` service. This time we have three environment variables we define.

  Defining an environment variable like this is the equivalent of using the `-e` parameter in the `docker container run` command.

The preceding snippet represents a complete and valid Docker compose file. It can be used to manage an application consisting of a Web service and a database easily and declaratively. To start the application as an example, we can simply execute the following command from within the folder in which the `docker-compose.yml` file is located:

3.x. The former is used whenever working on a single node, such as in development when using Docker Desktop or on a build server in a CI/CD pipeline. Version 3.x, on the other hand, is used when working with clusters, as we will see in *Chapter 10, Orchestrating Containers*.

- In Line 2, the `services` keyword declares that now what follows in this section is the list of services making up the application.
- In Lines 3 and 9, we have the names of two `services`, Web and `database`. These two services make up our sample application. The two names are indented to comply with the requirements of the YAML standard. Please be careful with the indentations; if you get this wrong, then you will get errors when using the Docker compose file later on.
- Each service needs information on which Docker image to use. In Line 4, we have the keyword `image:` followed by the name of the image to use. In our case, we use an image called `my-web:1.0` for the Web service.

  In Line 10, we see that the `database` service uses the image `postgres:alpine`.
- Now to Lines 5 and 6: The keyword `ports:` starts a section where we can declare which container ports we want to expose to the host computer. It is the equivalent of using the `-p` parameter in the `docker container run` command. In this example, we open container port 3000 and map it to host port 80.

  It is possible to map more than just one port. Just add another line starting with a hyphen in the `ports` section.

  Note that for the `database` service, we do not need to map a port, and thus, there is no `ports` section defined for this service.
- In Line 7, we have the section `environment:`. It allows us to define environment variables and their respective values for the my-web service. In this particular case, we only define a single variable called `connection_string` and assign it a value.

  In Lines 11–14, we do the same for the `database` service. This time we have three environment variables we define.

  Defining an environment variable like this is the equivalent of using the `-e` parameter in the `docker container run` command.

The preceding snippet represents a complete and valid Docker compose file. It can be used to manage an application consisting of a Web service and a database easily and declaratively. To start the application as an example, we can simply execute the following command from within the folder in which the `docker-compose.yml` file is located:

```
docker compose up
```

And Docker will locate the compose file and parse it. It will then start the two services by inspecting which image they use and running a container from each of those images, and configuring them according to the definitions we have also added to the compose file. Furthermore, Docker compose automatically creates a new bridge network for our application and attaches the two services to it. We do not have to specify how this shall happen; we are only declaring what we want as an end result.

Using build instructions

Docker compose files cannot only be used to orchestrate services using already built Docker images; no, we can also add instructions to the file on how to build a Docker image first from the source.

In its simplest form, the build command consists of just one line, such as the following:

```
build: web
```

This assumes that the `docker-compose.yml` file containing the preceding statement is sitting in a folder, which has a subfolder `web`, which, in turn, contains a `Dockerfile`.

If we want to use a different context for the Docker compose tool or use a Dockerfile that has a different name, then we can use the slightly more detailed syntax like in the following example:

```
build:
  context: web
  Dockerfile: docker/Dockerfile-dev
```

This snippet would instruct the tool to use the subfolder `web` as the execution context for the Docker build command and use a file called `Dockerfile-dev` as the manifest to build the image. Note that the Dockerfile is expected to be in the subfolder `docker` of the `web` folder.

Next, we will discuss how one can define what networks to use in a Docker compose application.

Using networks

By default, Docker compose automatically creates a new bridge network called the same as the folder inside which the `docker-compose.yml` file is located with a postfix `_default`. Thus, if the compose file is located in a folder `my-app` then the name of the network will be `my-app_default`. Docker compose then attaches all containers of the application to this network.

```
docker compose up
```

And Docker will locate the compose file and parse it. It will then start the two services by inspecting which image they use and running a container from each of those images, and configuring them according to the definitions we have also added to the compose file. Furthermore, Docker compose automatically creates a new bridge network for our application and attaches the two services to it. We do not have to specify how this shall happen; we are only declaring what we want as an end result.

Using build instructions

Docker compose files cannot only be used to orchestrate services using already built Docker images; no, we can also add instructions to the file on how to build a Docker image first from the source.

In its simplest form, the build command consists of just one line, such as the following:

```
build: web
```

This assumes that the `docker-compose.yml` file containing the preceding statement is sitting in a folder, which has a subfolder `web`, which, in turn, contains a `Dockerfile`.

If we want to use a different context for the Docker compose tool or use a Dockerfile that has a different name, then we can use the slightly more detailed syntax like in the following example:

```
build:
  context: web
  Dockerfile: docker/Dockerfile-dev
```

This snippet would instruct the tool to use the subfolder `web` as the execution context for the Docker build command and use a file called `Dockerfile-dev` as the manifest to build the image. Note that the Dockerfile is expected to be in the subfolder `docker` of the `web` folder.

Next, we will discuss how one can define what networks to use in a Docker compose application.

Using networks

By default, Docker compose automatically creates a new bridge network called the same as the folder inside which the `docker-compose.yml` file is located with a postfix `_default`. Thus, if the compose file is located in a folder `my-app` then the name of the network will be `my-app_default`. Docker compose then attaches all containers of the application to this network.

At times we want to define our own network or even define and use multiple networks to which some of the services are attached to. For this, we need to introduce a new section in the compose file called **networks**. This section is a top-level section similar to the **services** or the **version** section. Take this as an example:

```
1  version: '2'
2  services:
3      web:
4          image: my-web:1.0
5          networks:
6              - front-network
7      api:
8          image: my-api:1.0
9          networks:
10             - front-network
11             - back-network
12     database:
13         image: postgres:alpine
14         networks:
15             - back-network
16
17 networks:
18     front-network:
19     back-network:
```

Figure 7.2: Custom networks in a Docker compose file

Let us analyze this compose file a bit:

- In Line 17, we define this network section under which we can define the list of networks we want to use.
- In Lines 18 and 19, we create two custom networks called **front-network** and **back-network**, respectively. Since we do not define any other parameters there, Docker compose automatically creates these networks as bridge networks.
- In Lines 5 and 6, we define that we want the Web service to use and attached to the **front-network**.
- In Lines 9–11, we declare that we want the API service to use and attached to the **front-network** as well as to the **back-network**.
- Finally, in Lines 14 and 15, we instruct Docker compose to attach the database service to the **back-network**.

At times we want to define our own network or even define and use multiple networks to which some of the services are attached to. For this, we need to introduce a new section in the compose file called **networks**. This section is a top-level section similar to the **services** or the **version** section. Take this as an example:

```
1  version: '2'
2  services:
3      web:
4          image: my-web:1.0
5          networks:
6              - front-network
7      api:
8          image: my-api:1.0
9          networks:
10             - front-network
11             - back-network
12     database:
13         image: postgres:alpine
14         networks:
15             - back-network
16
17 networks:
18     front-network:
19     back-network:
```

Figure 7.2: Custom networks in a Docker compose file

Let us analyze this compose file a bit:

- In Line 17, we define this network section under which we can define the list of networks we want to use.
- In Lines 18 and 19, we create two custom networks called **front-network** and **back-network**, respectively. Since we do not define any other parameters there, Docker compose automatically creates these networks as bridge networks.
- In Lines 5 and 6, we define that we want the Web service to use and attached to the **front-network**.
- In Lines 9–11, we declare that we want the API service to use and attached to the **front-network** as well as to the **back-network**.
- Finally, in Lines 14 and 15, we instruct Docker compose to attach the database service to the **back-network**.

As you can see, we can define between 0 and many custom networks in a compose file, and we can assign each service to one or several of those networks. It is a popular best practice to use more than one network to run a multi-service application on. Networks serve as firewalls and thus increase the security of the application as a whole. For example, in the preceding configuration, the `web` service cannot access the `database` service directly since the two services sit on different networks. On the other hand, the `api` service is sitting in between and is attached to both networks and can communicate with the `web` and with the `database` service.

Authoring your own Docker-compose file

Now that you have a good grasp of the syntax of a Docker compose file, it is time that you try to author one yourself. To make this an interesting exercise, we want to first create a sample application that consists of a Web service containing the frontend code, an API service containing some backend code and a database that will store the data produced and managed by the application. Let us right dig in.

Preparing a multi-service application

In this section, we will create the application that we will use to demonstrate the capabilities of Docker compose. It shall be an application consisting of a Web frontend built using Node Express, a RESTful API built using Java and the Spring Boot framework, and finally, a PostgreSQL database used to manage the data produced and consumed by the applications.

The API service in Java

Let us start with the API service:

1. Open a new terminal window and navigate to the `~/src` folder where we have all our source code.
2. Create a folder `ch07/demo-app` in your `~/src` folder and navigate to it:

```
mkdir -p ch07/demo-app && ch07/demo-app
```

As you can see, we can define between 0 and many custom networks in a compose file, and we can assign each service to one or several of those networks. It is a popular best practice to use more than one network to run a multi-service application on. Networks serve as firewalls and thus increase the security of the application as a whole. For example, in the preceding configuration, the `web` service cannot access the `database` service directly since the two services sit on different networks. On the other hand, the `api` service is sitting in between and is attached to both networks and can communicate with the `web` and with the `database` service.

Authoring your own Docker-compose file

Now that you have a good grasp of the syntax of a Docker compose file, it is time that you try to author one yourself. To make this an interesting exercise, we want to first create a sample application that consists of a Web service containing the frontend code, an API service containing some backend code and a database that will store the data produced and managed by the application. Let us right dig in.

Preparing a multi-service application

In this section, we will create the application that we will use to demonstrate the capabilities of Docker compose. It shall be an application consisting of a Web frontend built using Node Express, a RESTful API built using Java and the Spring Boot framework, and finally, a PostgreSQL database used to manage the data produced and consumed by the applications.

The API service in Java

Let us start with the API service:

1. Open a new terminal window and navigate to the `~/src` folder where we have all our source code.
2. Create a folder `ch07/demo-app` in your `~/src` folder and navigate to it:

```
mkdir -p ch07/demo-app && ch07/demo-app
```

3. Open a new browser window and navigate to **https://start.spring.io**. Fill in the options as follows:

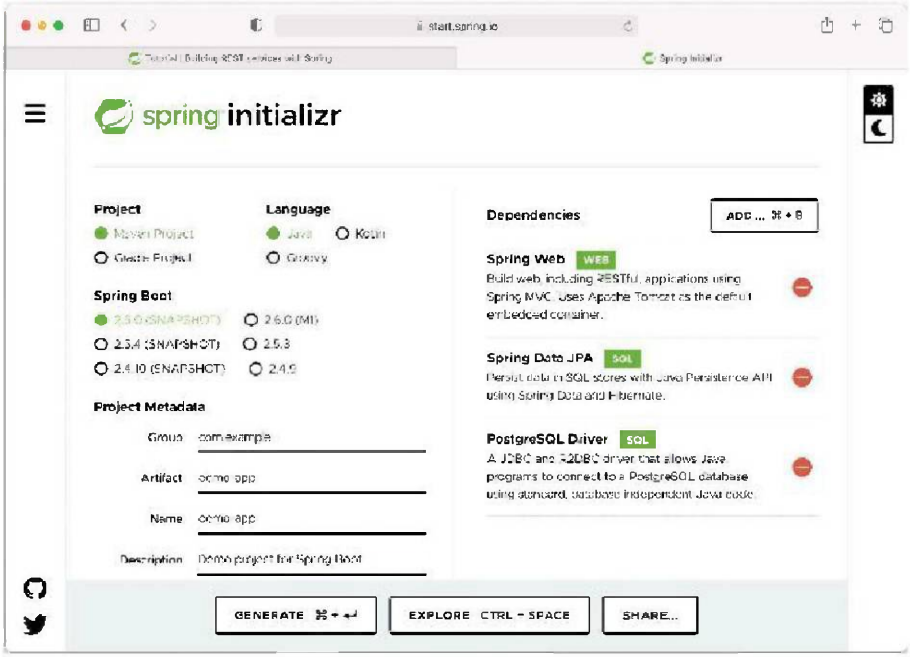


Figure 7.3: Spring boot initializer

   a. Project: **Maven**
   b. Language: **Java**
   c. Spring Boot: **2.5.3** (leave the selected default)
   d. Metadata: leave the defaults, except name the artifact **demo-api-java** (instead of **demo**)
   e. Dependencies: click **ADD** and then search for the **Spring Web**, the **Spring Data JPA** and the **PostgreSQL Driver** components and add them to the project

4. In the dialog box, click **GENERATE**. A file **demo-api-java.zip** containing a skeleton Java Spring Boot application will be generated and downloaded to your computer.
5. Extract the content of this zip file to a subfolder called **api-java** of the **demo-app** folder we created earlier.

3. Open a new browser window and navigate to **https://start.spring.io**. Fill in the options as follows:

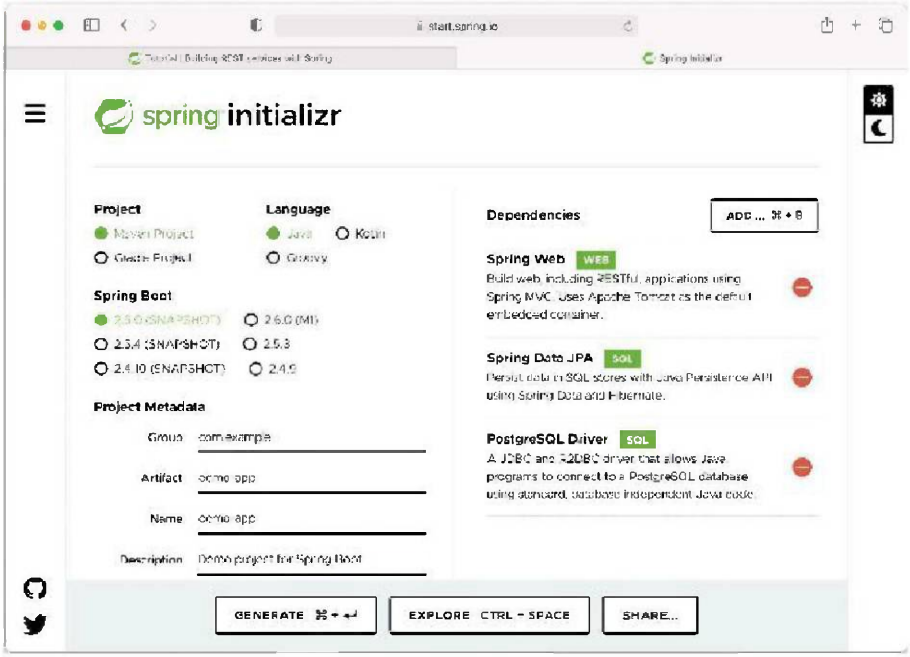


Figure 7.3: Spring boot initializer

   a. Project: **Maven**
   b. Language: **Java**
   c. Spring Boot: **2.5.3** (leave the selected default)
   d. Metadata: leave the defaults, except name the artifact **demo-api-java** (instead of **demo**)
   e. Dependencies: click **ADD** and then search for the **Spring Web**, the **Spring Data JPA** and the **PostgreSQL Driver** components and add them to the project

4. In the dialog box, click **GENERATE**. A file **demo-api-java.zip** containing a skeleton Java Spring Boot application will be generated and downloaded to your computer.
5. Extract the content of this zip file to a subfolder called **api-java** of the **demo-app** folder we created earlier.

6. From within the **demo-app** folder, open Visual Studio code with:

   ```
   code.
   ```

7. Explore the content of the Spring Boot app. You should see a structure similar to this:

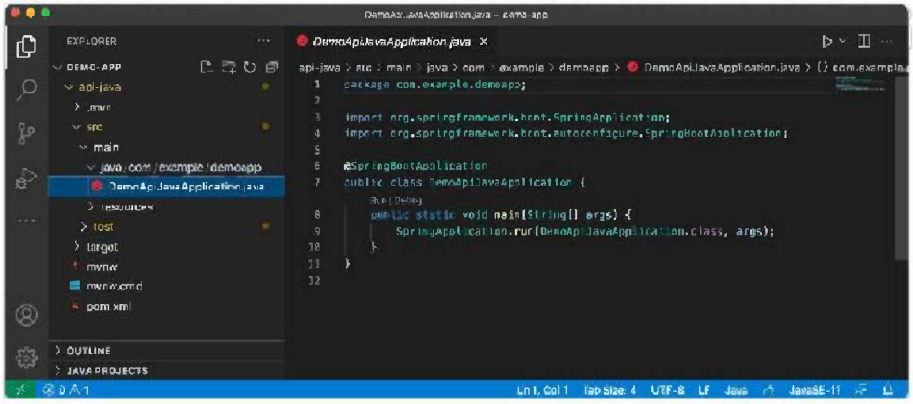


***Figure 7.4:** API service consisting of a Java Spring Boot application*

We can identify all files that are needed for a typical Java Spring Boot application using Maven as a lifecycle tool.

8. Try to debug the preceding application as follows:

   a. Put a breakpoint on Line 10.

   b. Click the Debug link right preceding the main method of the `DemoApiJavaApplication` class.

   c. Notice how the application is started and the execution stops at the breakpoint.

   d. Press *F5* to continue the execution of the application. Notice how the application is running. Of course, we have not defined any meaningful logic yet; thus, the application ends immediately.

   e. Observe the output produced into the terminal window of VS Code.

9. When done experimenting, stop the application.

10. We now want to create a simple API that accepts some rest calls to store, update and retrieve order entries from a PostgreSQL database. For this, we need to add a few Java classes, an order class, an order status enumeration, a repository to access the database and a REST controller.

6. From within the **demo-app** folder, open Visual Studio code with:

   ```
   code.
   ```

7. Explore the content of the Spring Boot app. You should see a structure similar to this:

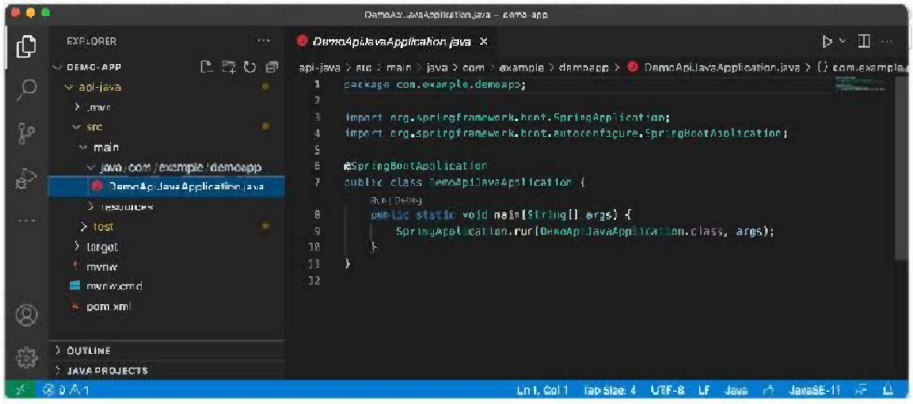


***Figure 7.4:** API service consisting of a Java Spring Boot application*

We can identify all files that are needed for a typical Java Spring Boot application using Maven as a lifecycle tool.

8. Try to debug the preceding application as follows:

   a. Put a breakpoint on Line 10.

   b. Click the Debug link right preceding the main method of the `DemoApiJavaApplication` class.

   c. Notice how the application is started and the execution stops at the breakpoint.

   d. Press *F5* to continue the execution of the application. Notice how the application is running. Of course, we have not defined any meaningful logic yet; thus, the application ends immediately.

   e. Observe the output produced into the terminal window of VS Code.

9. When done experimenting, stop the application.

10. We now want to create a simple API that accepts some rest calls to store, update and retrieve order entries from a PostgreSQL database. For this, we need to add a few Java classes, an order class, an order status enumeration, a repository to access the database and a REST controller.

11. First, let us create an `Order` class that we will use to store and retrieve orders from the database. Add a new file called `Order.java` as a sibling to the application class and add this content to it:

```
package com.example.demoapp;
import java.time.LocalDate;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "orders")
class Order {
  @Id
  @GeneratedValue
  private Long id;
  @Column(name = "order_date", nullable = false)
  private LocalDate orderDate;
  @Column(name = "status", nullable = false)
  private OrderStatus status;

  Order() {}

  public Order(LocalDate orderDate, OrderStatus status) {
    this.orderDate = orderDate;
    this.status = status;
  }

  public Long getId() {
    return this.id;
  }

  public LocalDate getOrderDate() {
    return this.orderDate;
  }
```

11. First, let us create an **Order** class that we will use to store and retrieve orders from the database. Add a new file called **Order.java** as a sibling to the application class and add this content to it:

```
package com.example.demoapp;
import java.time.LocalDate;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "orders")
class Order {
  @Id
  @GeneratedValue
  private Long id;
  @Column(name = "order_date", nullable = false)
  private LocalDate orderDate;
  @Column(name = "status", nullable = false)
  private OrderStatus status;

  Order() {}

  public Order(LocalDate orderDate, OrderStatus status) {
    this.orderDate = orderDate;
    this.status = status;
  }

  public Long getId() {
    return this.id;
  }

  public LocalDate getOrderDate() {
    return this.orderDate;
  }
```

```
    public OrderStatus getStatus() {
      return this.status;
    }

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

    public void setOrderDate(LocalDate orderDate) {
      this.orderDate = orderDate;
    }

    public void setStatus(OrderStatus status) {
      this.status = status;
    }

    @Override
    public String toString() {
      return "Order{" + "id=" + this.id + ", orderDate='" + this.
orderDate + '\'' + ", status='" + this.status + '\'' + '}';
    }
}
```

12. For the preceding class, we also need to define the `OrderStatus` enum. Thus, add another file next to the `Order` class called `OrderStatus.java` and give it this content:



Figure 7.5: OrderStatus Java enum

```
    public OrderStatus getStatus() {
      return this.status;
    }

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

    public void setOrderDate(LocalDate orderDate) {
      this.orderDate = orderDate;
    }

    public void setStatus(OrderStatus status) {
      this.status = status;
    }

    @Override
    public String toString() {
      return "Order{" + "id=" + this.id + ", orderDate='" + this.
orderDate + '\'' + ", status='" + this.status + '\'' + '}';
    }
}
```

12. For the preceding class, we also need to define the `OrderStatus` enum. Thus, add another file next to the `Order` class called `OrderStatus.java` and give it this content:



Figure 7.5: OrderStatus Java enum

13. Then, we need to define a repository. Add a new file next to the order class and call it `OrderRepository.java`. Add the following content to it. We are simply inheriting from the `JpaRepository` base interface, which does the heavy lifting for us:

```
OrderRepository.java ×
api-java > src > main > java > com > example > demoapp > OrderRepository.java >
1  package com.example.demoapp;
2
3  import org.springframework.data.jpa.repository.JpaRepository;
4
5  interface OrderRepository extends JpaRepository<Order, Long> {
6  }
```

Figure 7.6: OrderRepository Java class

14. Finally, let us add a controller and define an endpoint. Next to the file `DemoApiJavaApplication.java` create a file called `OrdersController.java` and add the following content:

```
OrderController.java ×
api-java > src > main > java > com > example > demoapp > OrderController.java
1  package com.example.demoapp;
2
3  import java.util.List;
4
5  import org.springframework.web.bind.annotation.GetMapping;
6  import org.springframework.web.bind.annotation.RestController;
7
8  @RestController
9  public class OrderController {
10     private OrderRepository repository;
11
12     public OrderController(OrderRepository repository) {
13         this.repository = repository;
14     }
15
16     @GetMapping("/orders")
17     List<Order> all() {
18       return repository.findAll();
19     }
20 }
```

Figure 7.7: OrdersController Java class

15. We also want to define a Java bean that will be used to initialize the database with some initial data we can then play with. Thus, add a file called `LoadDatabase.java` and give it this content:

13. Then, we need to define a repository. Add a new file next to the order class and call it `OrderRepository.java`. Add the following content to it. We are simply inheriting from the `JpaRepository` base interface, which does the heavy lifting for us:

```
OrderRepository.java ×
api-java > src > main > java > com > example > demoapp > OrderRepository.java >
1  package com.example.demoapp;
2
3  import org.springframework.data.jpa.repository.JpaRepository;
4
5  interface OrderRepository extends JpaRepository<Order, Long> {
6  }
```

Figure 7.6: OrderRepository Java class

14. Finally, let us add a controller and define an endpoint. Next to the file `DemoApiJavaApplication.java` create a file called `OrdersController.java` and add the following content:

```
OrderController.java ×
api-java > src > main > java > com > example > demoapp > OrderController.java
1  package com.example.demoapp;
2
3  import java.util.List;
4
5  import org.springframework.web.bind.annotation.GetMapping;
6  import org.springframework.web.bind.annotation.RestController;
7
8  @RestController
9  public class OrderController {
10     private OrderRepository repository;
11
12     public OrderController(OrderRepository repository) {
13         this.repository = repository;
14     }
15
16     @GetMapping("/orders")
17     List<Order> all() {
18       return repository.findAll();
19     }
20 }
```

Figure 7.7: OrdersController Java class

15. We also want to define a Java bean that will be used to initialize the database with some initial data we can then play with. Thus, add a file called `LoadDatabase.java` and give it this content:

LoadDatabase.java

api-java > src > main > java > com > example > demoapp > LoadDatabase.java > ...

```
1  package com.example.demoapp;
2
3  import java.time.LocalDate;
4  import org.springframework.boot.CommandLineRunner;
5  import org.springframework.context.annotation.Bean;
6  import org.springframework.context.annotation.Configuration;
7
8  @Configuration
9  class LoadDatabase {
10   @Bean
11   CommandLineRunner initDatabase(OrderRepository repository) {
12
13     return args -> {
14       repository.save(new Order(LocalDate.parse("2021-01-10"), OrderStatus.FULFILLED));
15       repository.save(new Order(LocalDate.parse("2021-01-11"), OrderStatus.FULFILLED));
16       repository.save(new Order(LocalDate.parse("2021-02-12"), OrderStatus.DRAFT));
17       repository.save(new Order(LocalDate.parse("2021-02-12"), OrderStatus.APPROVED));
18       repository.save(new Order(LocalDate.parse("2021-03-13"), OrderStatus.DRAFT));
19       repository.save(new Order(LocalDate.parse("2021-03-13"), OrderStatus.DRAFT));
20     };
21   }
22 }
```

Figure 7.8: LoadDatabase Java bean

16. We are done with the Java-specific part. Now comes the Docker-relevant part. To the **api-java** folder, add a new file called **Dockerfile** and add this content to it:

Dockerfile

api-java > Dockerfile > ...

```
1  FROM maven as builder
2  WORKDIR /app
3  COPY pom.xml ./
4  COPY src ./src
5  RUN mvn clean package -DskipTests
6
7  FROM openjdk:jre-slim
8  WORKDIR /app
9  COPY --from=builder /app/target/*.jar ./application.jar
10 ENTRYPOINT ["java", "-jar", "application.jar"]
```

Figure 7.9: Dockerfile for the Java-based API service

To follow best practices, we are using a multi-stage Dockerfile to keep the resulting image small. Also, note that we are skipping any tests on Line 5 when building the Java application since, in the current state, they would fail.

LoadDatabase.java ×

api-java > src > main > java > com > example > demoapp > LoadDatabase.java > ...

```
1   package com.example.demoapp;
2
3   import java.time.LocalDate;
4   import org.springframework.boot.CommandLineRunner;
5   import org.springframework.context.annotation.Bean;
6   import org.springframework.context.annotation.Configuration;
7
8   @Configuration
9   class LoadDatabase {
10    @Bean
11    CommandLineRunner initDatabase(OrderRepository repository) {
12
13      return args -> {
14        repository.save(new Order(LocalDate.parse("2021-01-10"), OrderStatus.FULFILLED));
15        repository.save(new Order(LocalDate.parse("2021-01-11"), OrderStatus.FULFILLED));
16        repository.save(new Order(LocalDate.parse("2021-02-12"), OrderStatus.DRAFT));
17        repository.save(new Order(LocalDate.parse("2021-02-12"), OrderStatus.APPROVED));
18        repository.save(new Order(LocalDate.parse("2021-03-13"), OrderStatus.DRAFT));
19        repository.save(new Order(LocalDate.parse("2021-03-13"), OrderStatus.DRAFT));
20      };
21    }
22  }
```

Figure 7.8: LoadDatabase Java bean

16. We are done with the Java-specific part. Now comes the Docker-relevant part. To the **`api-java`** folder, add a new file called **`Dockerfile`** and add this content to it:

Dockerfile ×

api-java > Dockerfile > ...

```
1   FROM maven as builder
2   WORKDIR /app
3   COPY pom.xml ./
4   COPY src ./src
5   RUN mvn clean package -DskipTests
6
7   FROM openjdk:jre-slim
8   WORKDIR /app
9   COPY --from=builder /app/target/*.jar ./application.jar
10  ENTRYPOINT ["java", "-jar", "application.jar"]
```

Figure 7.9: Dockerfile for the Java-based API service

To follow best practices, we are using a multi-stage Dockerfile to keep the resulting image small. Also, note that we are skipping any tests on Line 5 when building the Java application since, in the current state, they would fail.

For the moment, this is all we need to do. Our Java-based REST API using PostgreSQL is complete. Now that we have defined an API service using Java, we can do the same using .NET 5.x. With this, we want to show you that there is no limit to what languages you can use to write applications that can then be run in containers.

The API service in .NET 5.x

To create a .NET-based API, please follow these step-by-step instructions:

1. In a new terminal window, navigate to the application folder `ch07/demo-app`
2. Create a new Web API project using the dotnet command line tool

   ```
   dotnet new webapi -o api-dotnet
   ```

3. Navigate into the project folder `api-dotnet` that the previous command created:

   ```
   cd api-dotnet
   ```

4. We will use the `EntityFramework` and PostgreSQL; thus, add the necessary NuGet packages with the following command:

   ```
   dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL
   ```

   Note that by adding the Postgres driver (NuGet package) for the EntityFramework, the framework itself is also added.

5. Open the project in VS Code with the following:

   ```
   code .
   ```

6. Locate and remove the two files `WeatherForecast.cs` and `WeatherForecastController.cs`. We do not need them.
7. In the project folder, add a new folder called `Models`.
8. Add a file called `Order.cs` to the models folder and give it the following content:

For the moment, this is all we need to do. Our Java-based REST API using PostgreSQL is complete. Now that we have defined an API service using Java, we can do the same using .NET 5.x. With this, we want to show you that there is no limit to what languages you can use to write applications that can then be run in containers.

The API service in .NET 5.x

To create a .NET-based API, please follow these step-by-step instructions:

1. In a new terminal window, navigate to the application folder `ch07/demo-app`
2. Create a new Web API project using the dotnet command line tool

   ```
   dotnet new webapi -o api-dotnet
   ```

3. Navigate into the project folder `api-dotnet` that the previous command created:

   ```
   cd api-dotnet
   ```

4. We will use the `EntityFramework` and PostgreSQL; thus, add the necessary NuGet packages with the following command:

   ```
   dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL
   ```

   Note that by adding the Postgres driver (NuGet package) for the EntityFramework, the framework itself is also added.

5. Open the project in VS Code with the following:

   ```
   code .
   ```

6. Locate and remove the two files `WeatherForecast.cs` and `WeatherForecastController.cs`. We do not need them.
7. In the project folder, add a new folder called `Models`.
8. Add a file called `Order.cs` to the models folder and give it the following content:

```
using System;

namespace api_dotnet.Models
{
    1 references
    public class Order
    {
        7 references
        public long Id { get; set; }
        7 references
        public DateTime OrderDate { get; set; }
        8 references
        public OrderStatus Status { get; set; }

        public override string ToString() =>
            $"Id={Id}, OrderDate={OrderDate}, Status={Status}";
    }
}
```

Figure 7.10: Order class of the .NET API

9. To the models folder, add another class called **`DataContext.cs`** and add this content as follows. This will be our gateway to the database using **`EntityFramework`**:

```
using Microsoft.EntityFrameworkCore;

namespace api_dotnet.Models
{
    4 references
    public class DataContext: DbContext
    {
        0 references
        public DataContext(DbContextOptions<DataContext> options): base(options)
        { }

        3 references
        public DbSet<Order> Orders { get; set; }
    }
}
```

Figure 7.11: DataContext class of the .NET API

As you can see, we only need access to the orders table through the property **`Orders`**.

10. We want to initialize the database upon the start of the API service. This is a shortcut convenient for this demo application. In a production-like application, you would probably solve this with DB migrations that run in a standalone application.

```
using System;

namespace api_dotnet.Models
{
    1 references
    public class Order
    {
        7 references
        public long Id { get; set; }
        7 references
        public DateTime OrderDate { get; set; }
        8 references
        public OrderStatus Status { get; set; }

        public override string ToString() =>
            $"Id={Id}, OrderDate={OrderDate}, Status={Status}";
    }
}
```

Figure 7.10: Order class of the .NET API

9. To the models folder, add another class called **`DataContext.cs`** and add this content as follows. This will be our gateway to the database using **`EntityFramework`**:

```
using Microsoft.EntityFrameworkCore;

namespace api_dotnet.Models
{
    4 references
    public class DataContext: DbContext
    {
        0 references
        public DataContext(DbContextOptions<DataContext> options): base(options)
        { }

        3 references
        public DbSet<Order> Orders { get; set; }
    }
}
```

Figure 7.11: DataContext class of the .NET API

As you can see, we only need access to the orders table through the property **`Orders`**.

10. We want to initialize the database upon the start of the API service. This is a shortcut convenient for this demo application. In a production-like application, you would probably solve this with DB migrations that run in a standalone application.

In the **Startup** class, modify the **Configure** method as follows:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env, DataContext context)
{
    if(context.Database.EnsureCreated())
        SeedDatabase(context);

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
```

Figure 7.12: Modified Configure method in Startup class

The red-marked parts have been added.

11. Now, we need to define the method **SeeDatabase** called in the added snippet. Add this code snippet to the **Startup** class:

```
62          private void SeedDatabase(DataContext context)
63          {
64              Console.WriteLine("Trying to access DB... waiting 2 sec.");
65              System.Threading.Thread.Sleep(2000);
66              Console.WriteLine("Now trying to access DB...");
67              if(context.Orders.Count() > 0)
68                  return; // only add seed if empty...
69              context.Orders.AddRange(new []{
70                  new Order{ OrderDate = new DateTime(2021,8,1), Status = OrderStatus.Fulfilled },
71                  new Order{ OrderDate = new DateTime(2021,8,1), Status = OrderStatus.Accepted },
72                  new Order{ OrderDate = new DateTime(2021,8,2), Status = OrderStatus.Fulfilled },
73                  new Order{ OrderDate = new DateTime(2021,8,3), Status = OrderStatus.Accepted },
74                  new Order{ OrderDate = new DateTime(2021,8,3), Status = OrderStatus.Draft },
75              });
76              context.SaveChanges();
77          }
```

Figure 7.13: Method to seed the database

We are creating five new orders, but only if the orders table is empty. Please note that lines 65 and 67 are a hack to make sure the DB exists when we try to access it. In a real application, this code should be replaced with more robust code.

In the **Startup** class, modify the **Configure** method as follows:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env, DataContext context)
{
    if(context.Database.EnsureCreated())
        SeedDatabase(context);

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
```

Figure 7.12: Modified Configure method in Startup class

The red-marked parts have been added.

11. Now, we need to define the method **SeeDatabase** called in the added snippet. Add this code snippet to the **Startup** class:

```
62          private void SeedDatabase(DataContext context)
63          {
64              Console.WriteLine("Trying to access DB... waiting 2 sec.");
65              System.Threading.Thread.Sleep(2000);
66              Console.WriteLine("Now trying to access DB...");
67              if(context.Orders.Count() > 0)
68                  return; // only add seed if empty...
69              context.Orders.AddRange(new []{
70                  new Order{ OrderDate = new DateTime(2021,8,1), Status = OrderStatus.Fulfilled },
71                  new Order{ OrderDate = new DateTime(2021,8,1), Status = OrderStatus.Accepted },
72                  new Order{ OrderDate = new DateTime(2021,8,2), Status = OrderStatus.Fulfilled },
73                  new Order{ OrderDate = new DateTime(2021,8,3), Status = OrderStatus.Accepted },
74                  new Order{ OrderDate = new DateTime(2021,8,3), Status = OrderStatus.Draft },
75              });
76              context.SaveChanges();
77          }
```

Figure 7.13: Method to seed the database

We are creating five new orders, but only if the orders table is empty. Please note that lines 65 and 67 are a hack to make sure the DB exists when we try to access it. In a real application, this code should be replaced with more robust code.

12. Now, we have to register the **DataContext** class in the **Startup** class. Locate the **Startup.cs** file in the project folder and modify the method **ConfigureServices** by adding Lines 24–26 to it as shown as follows:

```
        0 references
22      public void ConfigureServices(IServiceCollection services)
23      {
24          var connString = Configuration.GetConnectionString("DemoDB");
25          services.AddDbContext<DataContext>(
26              options => options.UseNpgsql(connString));
27
28          services.AddControllers();
```

Figure 7.14: Configuring the data context in the Startup class

Please note how we retrieve the value for the connection string from the configuration property **ConnectionStrings:DemoDB**.

13. Now, we need to add a configuration setting called DemoDB in the respective configuration file. Locate the file **appsettings.Development.json** in the project folder and add Lines 2–4 to it as shown here:

```
1   {
2     "ConnectionStrings": {
3       "DemoDB": "Host=localhost;Port=5432;Database=demo-database;Username=demo-user;Password=Top-Secret"
4     },
5     "Logging": {
6       "LogLevel": {
7         "Default": "Information",
```

Figure 7.15: Defining the connection string in the application settings

The value for the connection string property should be the following:

```
Host=localhost;Port=5432;Database=demo-database;Username=demo-
user;Password=Top-Secret
```

12. Now, we have to register the **DataContext** class in the **Startup** class. Locate the **Startup.cs** file in the project folder and modify the method **ConfigureServices** by adding Lines 24–26 to it as shown as follows:

```
        0 references
22      public void ConfigureServices(IServiceCollection services)
23      {
24          var connString = Configuration.GetConnectionString("DemoDB");
25          services.AddDbContext<DataContext>(
26              options => options.UseNpgsql(connString));
27
28          services.AddControllers();
```

Figure 7.14: Configuring the data context in the Startup class

Please note how we retrieve the value for the connection string from the configuration property **ConnectionStrings:DemoDB**.

13. Now, we need to add a configuration setting called DemoDB in the respective configuration file. Locate the file **appsettings.Development.json** in the project folder and add Lines 2–4 to it as shown here:

```
1  {
2    "ConnectionStrings": {
3      "DemoDB": "Host=localhost;Port=5432;Database=demo-database;Username=demo-user;Password=Top-Secret"
4    },
5    "Logging": {
6      "LogLevel": {
7        "Default": "Information",
```

Figure 7.15: Defining the connection string in the application settings

The value for the connection string property should be the following:

```
Host=localhost;Port=5432;Database=demo-database;Username=demo-
user;Password=Top-Secret
```

14. To the `Controllers` folder, add a new class called `OrdersController.cs` and add the following code to it:

```
1  using System.Collections.Generic;
2  using System.Linq;
3  using Microsoft.AspNetCore.Mvc;
4  using api_dotnet.Models;
5
6  namespace api_dotnet.Controllers
7  {
8      [ApiController]
9      [Route("[controller]")]
       2 references
10     public class OrdersController : ControllerBase
11     {
           1 reference
12         private readonly DataContext dataContext;
13
           0 references
14         public OrdersController(DataContext dataContext)
15         {
16             this.dataContext = dataContext;
17         }
18
19         [HttpGet]
           0 references
20         public IEnumerable<Order> Get([FromQuery]OrderStatus? status)
21         { //TODO: add code here }
22
23         [HttpPost]
24         public void AddOrder(Order order)
25         { //TODO: add code here }
26     }
27 }
```

Figure 7.16: OrdersController class of the .NET API

Note that this is only the skeleton of the class, and the code for the two methods for retrieving a list of orders by status and adding a new order has not yet been filled in.

15. Now, we add the code to the `Get` method to retrieve the list of orders from the database. We will use the data context for this that we have injected in the constructor of the class on Line 14. Modify the `Get` method so that it looks like the following:

```
22         [HttpGet]
           0 references
23         public IEnumerable<Order> Get([FromQuery]OrderStatus? status)
24         {
25             return status == null
26                 ? dataContext.Orders.ToList()
27                 : dataContext.Orders
28                     .Where(o => o.Status == status)
29                     .ToList();
30         }
```

Figure 7.17: HTTP Get method to retrieve all orders

14. To the `Controllers` folder, add a new class called `OrdersController.cs` and add the following code to it:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using api_dotnet.Models;

namespace api_dotnet.Controllers
{
    [ApiController]
    [Route("[controller]")]
    2 references
    public class OrdersController : ControllerBase
    {
        1 reference
        private readonly DataContext dataContext;

        0 references
        public OrdersController(DataContext dataContext)
        {
            this.dataContext = dataContext;
        }

        [HttpGet]
        0 references
        public IEnumerable<Order> Get([FromQuery]OrderStatus? status)
        { //TODO: add code here }

        [HttpPost]
        public void AddOrder(Order order)
        { //TODO: add code here }
    }
}
```

Figure 7.16: OrdersController class of the .NET API

Note that this is only the skeleton of the class, and the code for the two methods for retrieving a list of orders by status and adding a new order has not yet been filled in.

15. Now, we add the code to the `Get` method to retrieve the list of orders from the database. We will use the data context for this that we have injected in the constructor of the class on Line 14. Modify the `Get` method so that it looks like the following:

```
22      [HttpGet]
        0 references
23      public IEnumerable<Order> Get([FromQuery]OrderStatus? status)
24      {
25          return status == null
26              ? dataContext.Orders.ToList()
27              : dataContext.Orders
28                  .Where(o => o.Status == status)
29                  .ToList();
30      }
```

Figure 7.17: HTTP Get method to retrieve all orders

16. Finally, modify the `AddOrder` method to look like the following:

```
32          [HttpPost]
            0 references
33          public void AddOrder(Order order)
34          {
35              dataContext.Orders.Add(order);
36              dataContext.SaveChanges();
37          }
```

Figure 7.18: HTTP POST method to add a new order

17. Make sure the project can be compiled and run. In your terminal window, execute the following command:

    ```
    dotnet run
    ```

    If everything goes well, you should see something like this in your terminal:

```
gabriel@Solaris api-dotnet % dotnet run
Buildvorgang wird ausgeführt...
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
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:
```

Figure 7.19: Running the .NET based API

    As we can see, the API can be accessed at `https://localhost:5001`

16. Finally, modify the `AddOrder` method to look like the following:

```
32          [HttpPost]
            0 references
33          public void AddOrder(Order order)
34          {
35              dataContext.Orders.Add(order);
36              dataContext.SaveChanges();
37          }
```

Figure 7.18: HTTP POST method to add a new order

17. Make sure the project can be compiled and run. In your terminal window, execute the following command:

    ```
    dotnet run
    ```

    If everything goes well, you should see something like this in your terminal:

```
gabriel@Solaris api-dotnet % dotnet run
Buildvorgang wird ausgeführt...
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
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:
```

Figure 7.19: Running the .NET based API

    As we can see, the API can be accessed at `https://localhost:5001`

18. Open a new browser tab and navigate to

    `https://localhost:5001/swagger/index.html`, and you should see the Swagger documentation of the API:

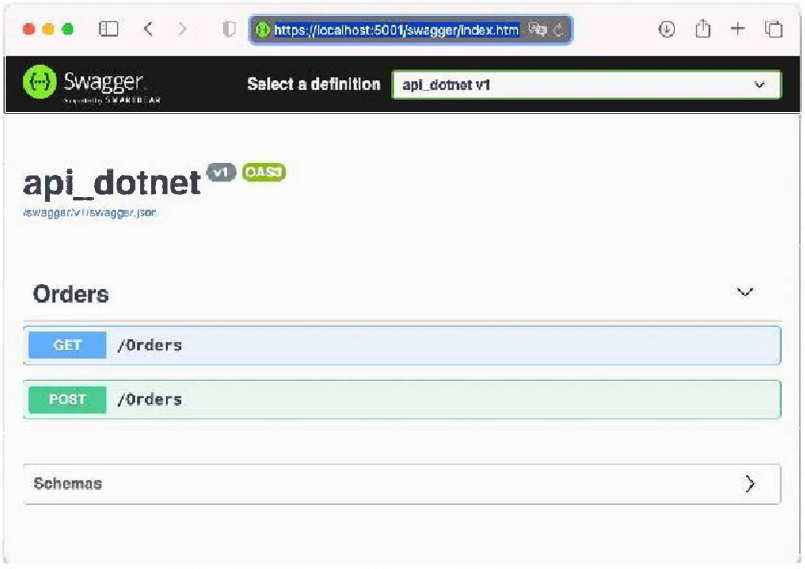


Figure 7.20: Swagger documentation of the .NET API

19. Add a new file called `Dockerfile` to the project folder of the .NET API with the following content. We are using a multi-stage Dockerfile to generate a minimal Docker image according to best practices:

```
1   FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
2   WORKDIR /src
3   COPY ["api-dotnet.csproj", "./"]
4   RUN dotnet restore
5   COPY . .
6   RUN dotnet publish -c Release -o /app/publish
7
8   FROM mcr.microsoft.com/dotnet/aspnet:5.0
9   WORKDIR /app
10  COPY --from=build /app/publish .
11  EXPOSE 80
12  ENTRYPOINT ["dotnet", "api-dotnet.dll"]
```

Figure 7.21: Dockerfile for the .NET-based API

With this, our .NET-based REST API is complete, and we can proceed with the next microservice, the Web frontend.

18. Open a new browser tab and navigate to

    `https://localhost:5001/swagger/index.html`, and you should see the Swagger documentation of the API:

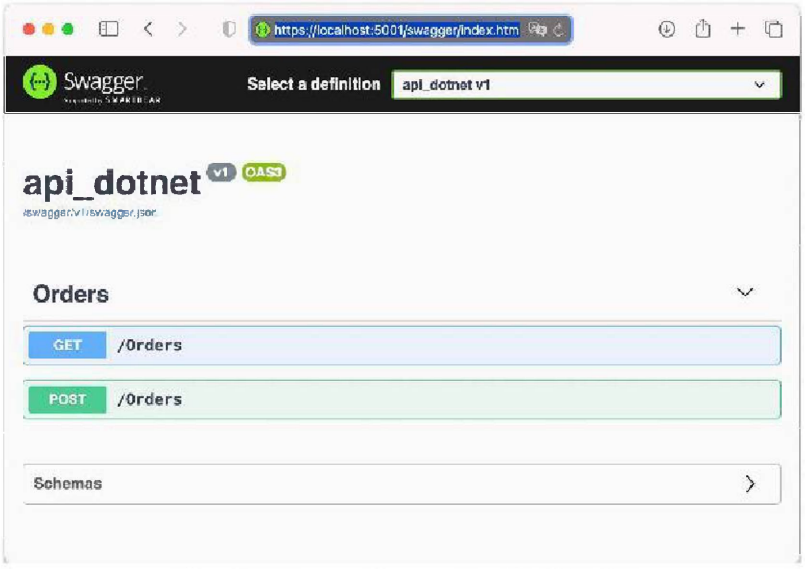


Figure 7.20: Swagger documentation of the .NET API

19. Add a new file called `Dockerfile` to the project folder of the .NET API with the following content. We are using a multi-stage Dockerfile to generate a minimal Docker image according to best practices:

```
1   FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
2   WORKDIR /src
3   COPY ["api-dotnet.csproj", "./"]
4   RUN dotnet restore
5   COPY . .
6   RUN dotnet publish -c Release -o /app/publish
7
8   FROM mcr.microsoft.com/dotnet/aspnet:5.0
9   WORKDIR /app
10  COPY --from=build /app/publish .
11  EXPOSE 80
12  ENTRYPOINT ["dotnet", "api-dotnet.dll"]
```

Figure 7.21: Dockerfile for the .NET-based API

With this, our .NET-based REST API is complete, and we can proceed with the next microservice, the Web frontend.

The Web service

Next in the list is the Web service that we will implement using Node Express JS. This Web service is our UI and will access the **api** service we built previously. Follow the following steps:

1. In a terminal window, navigate to the **demo-app** folder previously created.
2. Create a subfolder called Web and navigate to it:

   ```
   mkdir web && cd web
   ```

3. Use **npm** to initialize a new Node JS application:

   ```
   npm init
   ```

   when asked by the tool to accept all default values. At the end of this process, you should have a **packages.json** file created in your Web folder looking similar to the following:

```
{
  "name": "web",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  Debuggen. ▷
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Figure 7.22: The packages.json file of the Web project

4. Add Express JS to the project with the following command:

   ```
   npm install express -s
   ```

   This will install Express locally and add a **dependencies** node with a subnode **express** to your **packages.json** file. In my case, it installed version **^4.17.1** of the library.

5. Also, install the two libraries **got** (a popular Web client library) and **pug** (a popular template engine for ExpressJS) with the following:

   ```
   npm install got pug -s
   ```

The Web service

Next in the list is the Web service that we will implement using Node Express JS. This Web service is our UI and will access the **api** service we built previously. Follow the following steps:

1. In a terminal window, navigate to the **demo-app** folder previously created.
2. Create a subfolder called Web and navigate to it:

   ```
   mkdir web && cd web
   ```

3. Use **npm** to initialize a new Node JS application:

   ```
   npm init
   ```

   when asked by the tool to accept all default values. At the end of this process, you should have a **packages.json** file created in your Web folder looking similar to the following:

```
{
  "name": "web",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  Debuggen. ▷
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Figure 7.22: The packages.json file of the Web project

4. Add Express JS to the project with the following command:

   ```
   npm install express -s
   ```

   This will install Express locally and add a **dependencies** node with a subnode **express** to your **packages.json** file. In my case, it installed version **^4.17.1** of the library.

5. Also, install the two libraries **got** (a popular Web client library) and **pug** (a popular template engine for ExpressJS) with the following:

   ```
   npm install got pug -s
   ```

6. Add a file called **index.js** to the project folder (Web) and add this content:

```
1  const express = require('express')
2  const bodyParser = require("body-parser");
3  const got = require('got');
4  const path = require('path');
5
6  const app = express()
7  const port = 3000
8
9  app.use(express.static('public'));
10 app.set("view engine", "pug");
11 app.set("views", path.join(__dirname, "views"));
12 app.use(bodyParser.json());
13
14 app.listen(port, () => {
15   console.log('Demo Web application listening at http://localhost:${port}')
16 })
17
18 app.get('/', (req, res) => {
19   res.render('index');
20 })
```

Figure 7.23: The index.js file of the Web application

The code on Lines 1–16 is initializing our Web application. In Lines 18–20, we define an HTTP GET endpoint "/" and render the index page when hit. We will define the template for this page next.

7. Create a folder called **views** in the project folder and add a file called **index.pug** to it. The extension **pug** is due to the fact that we use the **pug** template library. The file should have the following content:

```
1 html
2   head
3     title Demo Application
4   body
5     h1 Demo Web application!
```

Figure 7.24: Content of the index template

8. Try to run the Web application by executing the following command from within your terminal window:

```
node index.js
```

You should see this output in your terminal:

```
Demo Web application listening at http://localhost:3000
```

6. Add a file called **index.js** to the project folder (Web) and add this content:

```
1  const express = require('express')
2  const bodyParser = require("body-parser");
3  const got = require('got');
4  const path = require('path');
5
6  const app = express()
7  const port = 3000
8
9  app.use(express.static('public'));
10 app.set("view engine", "pug");
11 app.set("views", path.join(__dirname, "views"));
12 app.use(bodyParser.json());
13
14 app.listen(port, () => {
15   console.log(`Demo Web application listening at http://localhost:${port}`)
16 })
17
18 app.get('/', (req, res) => {
19   res.render('index');
20 })
```

Figure 7.23: The index.js file of the Web application

The code on Lines 1–16 is initializing our Web application. In Lines 18–20, we define an HTTP GET endpoint "/" and render the index page when hit. We will define the template for this page next.

7. Create a folder called **views** in the project folder and add a file called **index.pug** to it. The extension **pug** is due to the fact that we use the **pug** template library. The file should have the following content:

```
1 html
2   head
3     title Demo Application
4   body
5     h1 Demo Web application!
```

Figure 7.24: Content of the index template

8. Try to run the Web application by executing the following command from within your terminal window:

```
node index.js
```

You should see this output in your terminal:

```
Demo Web application listening at http://localhost:3000
```

9. Open a browser tab and navigate to **`http://localhost:3000`**. You should be greeted by view with the message Hello Demo Web application!

10. Next, we are going to implement the HTTP GET endpoint **`/orders`**. Add this code snippet to the **`index.js`** file:

```
22  app.get('/orders', async (req, res) => {
23      const status = req.query.status || 'draft';
24      console.log("Getting order for status: " + status);
25      const baseUrl = process.env.API_ENDPOINT || 'http://localhost:5000';
26      const url = `${baseUrl}/orders?status=${status}` ;
27      const orders = await got(url).json();
28      res.render('orders', {title: 'Orders', message: 'List of Orders', orders: orders});
29  })
```

Figure 7.25: HTTP GET method in the index.js file

Please note Line 25. We are reading the base URL to the API from the environment variable called **`API_ENDPOINT`**. If no such variable is defined, then we are assuming **`http://localhost:5000`**. As you can see on Line 28, we are rendering a template called **`orders`** and pass some data, such as the list or orders retrieved from the API service, to the template engine.

11. Add a file called **`orders.pug`** to the **`views`** folder and add this content to it:

```
1   html
2     head
3       title= title
4       link(href='css/main.css' rel='stylesheet')
5     body
6       h1= message
7       p= 'Number of orders:' + orders.length
8       div(class='container')
9         div(class='cell') ID
10        div(class='cell') Order Date
11        div(class='cell') Status
12        each order in orders
13          div(calss='cell')=order.id
14          div(calss='cell')=order.orderDate
15          div(calss='cell')=order.status
16      hr
17
```

Figure 7.26: Orders PUG template

9. Open a browser tab and navigate to **`http://localhost:3000`**. You should be greeted by view with the message Hello Demo Web application!

10. Next, we are going to implement the HTTP GET endpoint **`/orders`**. Add this code snippet to the **`index.js`** file:

```
22  app.get('/orders', async (req, res) => {
23      const status = req.query.status || 'draft';
24      console.log("Getting order for status: " + status);
25      const baseUrl = process.env.API_ENDPOINT || 'http://localhost:5000';
26      const url = `${baseUrl}/orders?status=${status}`;
27      const orders = await got(url).json();
28      res.render('orders', {title: 'Orders', message: 'List of Orders', orders: orders});
29  })
```

Figure 7.25: HTTP GET method in the index.js file

Please note Line 25. We are reading the base URL to the API from the environment variable called **`API_ENDPOINT`**. If no such variable is defined, then we are assuming **`http://localhost:5000`**. As you can see on Line 28, we are rendering a template called **`orders`** and pass some data, such as the list or orders retrieved from the API service, to the template engine.

11. Add a file called **`orders.pug`** to the **`views`** folder and add this content to it:

```
1   html
2     head
3       title= title
4       link(href='css/main.css' rel='stylesheet')
5     body
6       h1= message
7       p= 'Number of orders:' + orders.length
8       div(class='container')
9         div(class='cell') ID
10        div(class='cell') Order Date
11        div(class='cell') Status
12        each order in orders
13          div(calss='cell')=order.id
14          div(calss='cell')=order.orderDate
15          div(calss='cell')=order.status
16      hr
17
```

Figure 7.26: Orders PUG template

12. Create a folder **public/css** in the project folder and add a file called **main.css** to this new folder. The file should have this content:

```
body {
    font-family: Arial, Helvetica, sans-serif;
}

.container {
    display: grid;
    grid-template-columns: auto auto auto;
}
```

Figure 7.27: Content of the main.css file

13. Add a file called **Dockerfile** to the project folder **web** with the following content:

```
FROM node:slim
WORKDIR /app
COPY package.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD node index.js
```

Figure 7.28: Dockerfile for the Web service

To keep the image small, we use the slim version of the Node base image. As you can see, we are using Dockerfile best practices in the preceding example. We first only copy the **package.json** file into the container and then run the **npm install**. Only after that we copy all other files of the project into the container.

Now that we have completed our Web service, we are ready to author the Docker compose file, which we will then use to manage our multi-service demo application.

Writing your first Docker-compose file

Now that we have created our application consisting of a simple Node Express Web frontend and .NET RESTful API, we will define a **docker-compose** file that we will then use to build Docker images for the services and then run them. Let us start with the following:

1. In the root of our project **demo-app**, create a file called **docker-compose.yml**
2. We will start by defining a service for the PostgreSQL database we are going to use in our application. Add the following code snippet to the Docker compose file:

12. Create a folder **public/css** in the project folder and add a file called **main.css** to this new folder. The file should have this content:

```
body {
    font-family: Arial, Helvetica, sans-serif;
}

.container {
    display: grid;
    grid-template-columns: auto auto auto;
}
```

Figure 7.27: Content of the main.css file

13. Add a file called **Dockerfile** to the project folder **web** with the following content:

```
FROM node:slim
WORKDIR /app
COPY package.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD node index.js
```

Figure 7.28: Dockerfile for the Web service

To keep the image small, we use the slim version of the Node base image. As you can see, we are using Dockerfile best practices in the preceding example. We first only copy the **package.json** file into the container and then run the **npm install**. Only after that we copy all other files of the project into the container.

Now that we have completed our Web service, we are ready to author the Docker compose file, which we will then use to manage our multi-service demo application.

Writing your first Docker-compose file

Now that we have created our application consisting of a simple Node Express Web frontend and .NET RESTful API, we will define a **docker-compose** file that we will then use to build Docker images for the services and then run them. Let us start with the following:

1. In the root of our project **demo-app**, create a file called **docker-compose.yml**
2. We will start by defining a service for the PostgreSQL database we are going to use in our application. Add the following code snippet to the Docker compose file:

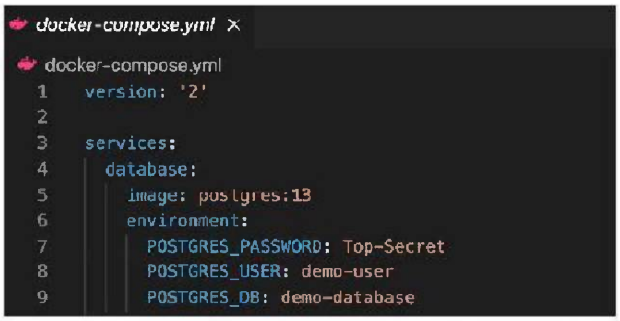


Figure 7.29: Defining the database service in the Docker compose file

As you can see on Line 4, we call the service **database**. This will also be the DNS name with which code from other containers on the same network can call the DB. We are using Postgres version 13 (Line 5). In Lines 7–9, we are defining values for the three environment variables **POSTGRES_PASSWORD**, **POSTGRES_USER** and **POSTGRES_DB**. The database service running inside the container will pick them up and create a database called **demo-database**, whose owner will be a **demo-user** with the password **Top-Secret**.

3. Next, add a service called **pgadmin**. The corresponding container will host PgAmin4, a popular management tool for PostgreSQL. Add the following snippet right after Line 10:

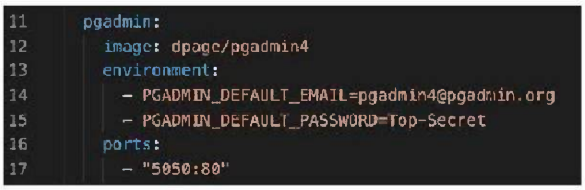


Figure 7.30: Defining the pgadmin service in the Docker compose file

We are using the latest version of the tool (Line 12). In Lines 14 and 15, we are defining values for the mandatory environment variables **PGADMIN_DEFAULT_EMAIL** and **PGADMIN_DEFAULT_PASSWORD**. We will later use those values to login to the tool. Finally, we are mapping container port 80 to host port 5050; that is, we will be able to access PgAdmin4 through this port later on.

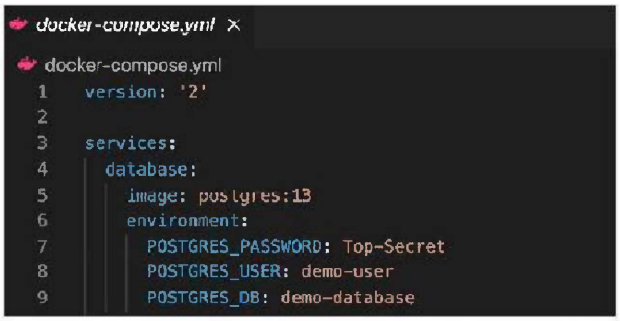


Figure 7.29: Defining the database service in the Docker compose file

As you can see on Line 4, we call the service **database**. This will also be the DNS name with which code from other containers on the same network can call the DB. We are using Postgres version 13 (Line 5). In Lines 7–9, we are defining values for the three environment variables **POSTGRES_PASSWORD**, **POSTGRES_USER** and **POSTGRES_DB**. The database service running inside the container will pick them up and create a database called **demo-database**, whose owner will be a **demo-user** with the password **Top-Secret**.

3. Next, add a service called **pgadmin**. The corresponding container will host PgAmin4, a popular management tool for PostgreSQL. Add the following snippet right after Line 10:

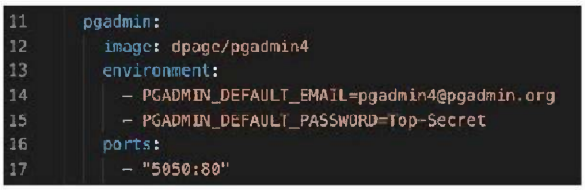


Figure 7.30: Defining the pgadmin service in the Docker compose file

We are using the latest version of the tool (Line 12). In Lines 14 and 15, we are defining values for the mandatory environment variables **PGADMIN_DEFAULT_EMAIL** and **PGADMIN_DEFAULT_PASSWORD**. We will later use those values to login to the tool. Finally, we are mapping container port 80 to host port 5050; that is, we will be able to access PgAdmin4 through this port later on.

4. We will now define a service for the Java-based API. Add the following code snippet to the Docker compose file after Line 18:

```
19    api-java:
20      image: dockerforjobseekers/api-java:0.1
21      build: api-java
22      depends_on:
23        - database
24      environment:
25        - SPRING_DATASOURCE_URL=jdbc:postgresql://database:5432/demo-database
26        - SPRING_DATASOURCE_USERNAME=demo-user
27        - SPRING_DATASOURCE_PASSWORD=Top-Secret
28        - SPRING_JPA_HIBERNATE_DDL_AUTO=update
29      ports:
30        - 8080:8080
```

Figure 7.31: Defining the service for the Java-based API in the Docker compose file

In Line 20, we call the image of the service **`dockerforjobseekers/api-java:0.1`**. We explicitly use a complete name, including organization and tag, since later on, we will want to push that image to an image registry. In Line 21, we then define that the Dockerfile for the build can be found in the subfolder **`api-java`** of our project folder. In Line 23, we declare that the **`api-java`** service is dependent on the **`database`** service. We will learn later in this chapter what that exactly means. Then, in Lines 25–28, we define environment variables for the JPA framework, such as that it knows what values to use when connecting to the database. Note that the values correspond to those selected for the database service. The last of the environment values make sure that Hibernate automatically creates the DB schema if not present.

We have mapped the container port 8080 to host port 8080 on Line 30. Technically this is not necessary. Here, we do it so that we can then easily debug the API by accessing it directly from the command line with a tool such as **`curl`**.

5. We will now define a service for the .NET API. Add this snippet after Line 31:

```
32    api-dotnet:
33      image: dockerforjobseekers/demo-api:0.1
34      build: api-dotnet
35      ports:
36        - 5000:80
37      environment:
38        ConnectionStrings__DemoDB: Host=database;Port=5432;Database=demo-databa
39      depends_on:
40        - database
```

Figure 7.32: Defining the service for the .NET-based API in the Docker compose file

4. We will now define a service for the Java-based API. Add the following code snippet to the Docker compose file after Line 18:

```
19    api-java:
20      image: dockerforjobseekers/api-java:0.1
21      build: api-java
22      depends_on:
23        - database
24      environment:
25        - SPRING_DATASOURCE_URL=jdbc:postgresql://database:5432/demo-database
26        - SPRING_DATASOURCE_USERNAME=demo-user
27        - SPRING_DATASOURCE_PASSWORD=Top-Secret
28        - SPRING_JPA_HIBERNATE_DDL_AUTO=update
29      ports:
30        - 8080:8080
```

Figure 7.31: Defining the service for the Java-based API in the Docker compose file

In Line 20, we call the image of the service **`dockerforjobseekers/api-java:0.1`**. We explicitly use a complete name, including organization and tag, since later on, we will want to push that image to an image registry. In Line 21, we then define that the Dockerfile for the build can be found in the subfolder **`api-java`** of our project folder. In Line 23, we declare that the **`api-java`** service is dependent on the **`database`** service. We will learn later in this chapter what that exactly means. Then, in Lines 25–28, we define environment variables for the JPA framework, such as that it knows what values to use when connecting to the database. Note that the values correspond to those selected for the database service. The last of the environment values make sure that Hibernate automatically creates the DB schema if not present.

We have mapped the container port 8080 to host port 8080 on Line 30. Technically this is not necessary. Here, we do it so that we can then easily debug the API by accessing it directly from the command line with a tool such as **`curl`**.

5. We will now define a service for the .NET API. Add this snippet after Line 31:

```
32    api-dotnet:
33      image: dockerforjobseekers/demo-api:0.1
34      build: api-dotnet
35      ports:
36        - 5000:80
37      environment:
38        ConnectionStrings__DemoDB: Host=database;Port=5432;Database=demo-databa
39      depends_on:
40        - database
```

Figure 7.32: Defining the service for the .NET-based API in the Docker compose file

In Line 33, we call the image of the service **`dockerforjobseekers/api-dotnet:0.1`**. We explicitly use a complete name, including organization and tag, since later on, we will want to push that image to an image registry. In Line 34, we then define that the Dockerfile for the build can be found in the subfolder **`api-dotnet`** of our project folder. We have mapped container port 80 to host port 5000 on Line 36. Technically this is not necessary. Here we do it, and we can then easily debug the API by accessing it directly from the command line with a tool such as **`curl`**. In Line 38, we define a value for the environment variable **`ConnectionStrings__DemoDB`**, which will override the value defined in the **`appsettings.Development.json`** file. Notice the double underline (__) in the name of the environment variable, which is used to define the equivalent of a nested variable in the JSON file. The value of the environment variable is truncated in the preceding image because it is too long to display. The value should be the following:

```
Host=database;Port=5432;Database=demo-database;Username=demo-
user;Password=Top-Secret
```

The only thing that changed compared to the value we have in the app settings file is the Host value, which here is the database (the DNS name of the Postgres DB).

6. Finally, the last service we have to define is the Web service. Please add this snippet after Line 41 of the Docker compose file:

```
42    web:
43      image: dockerforjobseekers/demo-web:0.1
44      build: web
45      ports:
46        - 3000:3000
47      environment:
48        API_ENDPOINT: http://api-java:8080   # Java API
49        # API_ENDPOINT: http://api-dotnet:5000   # .NET API
50      depends_on:
51        - api-java
52        #- api-dotnet
```

Figure 7.33: Defining the Web service in the Docker compose file

In Line 43, we call the Docker image of this service **`dockerforjobseekers/demo-web:1.0`**, and in Line 44 define that the Dockerfile for the image is to be found in a subfolder called **`web`** of the project folder. To be able to access the **`web`** service from a browser, we map container port 3000 to the equivalent host port (Line 46). Finally, in Line 48, we define the base URL of the API service the **`web`** service will be using. Since we called the service for the API "**`api-java`**", the base URL has to be **`http://api-java:5000`**. In Line 49, we have the (commented out) value we would use if we wanted to play with the .NET version of the API instead.

In Line 33, we call the image of the service **`dockerforjobseekers/api-dotnet:0.1`**. We explicitly use a complete name, including organization and tag, since later on, we will want to push that image to an image registry. In Line 34, we then define that the Dockerfile for the build can be found in the subfolder **`api-dotnet`** of our project folder. We have mapped container port 80 to host port 5000 on Line 36. Technically this is not necessary. Here we do it, and we can then easily debug the API by accessing it directly from the command line with a tool such as **`curl`**. In Line 38, we define a value for the environment variable **`ConnectionStrings__DemoDB`**, which will override the value defined in the **`appsettings.Development.json`** file. Notice the double underline (**`__`**) in the name of the environment variable, which is used to define the equivalent of a nested variable in the JSON file. The value of the environment variable is truncated in the preceding image because it is too long to display. The value should be the following:

```
Host=database;Port=5432;Database=demo-database;Username=demo-
user;Password=Top-Secret
```

The only thing that changed compared to the value we have in the app settings file is the Host value, which here is the database (the DNS name of the Postgres DB).

6. Finally, the last service we have to define is the Web service. Please add this snippet after Line 41 of the Docker compose file:

```
42    web:
43      image: dockerforjobseekers/demc-web:0.1
44      build: web
45      ports:
46        - 3000:3000
47      environment:
48        API_ENDPOINT: http://api-java:8080   # Java API
49        # API_ENDPOINT: http://api-dotnet:5000   # .NET API
50      depends_on:
51        - api-java
52        #- api-dotnet
```

Figure 7.33: Defining the Web service in the Docker compose file

In Line 43, we call the Docker image of this service **`dockerforjobseekers/demo-web:1.0`**, and in Line 44 define that the Dockerfile for the image is to be found in a subfolder called **`web`** of the project folder. To be able to access the **`web`** service from a browser, we map container port 3000 to the equivalent host port (Line 46). Finally, in Line 48, we define the base URL of the API service the **`web`** service will be using. Since we called the service for the API "**`api-java`**", the base URL has to be **`http://api-java:5000`**. In Line 49, we have the (commented out) value we would use if we wanted to play with the .NET version of the API instead.

In Line 51, we define that the Web service is dependent on the `api-java` service. Once again, if we want to use the .NET-based API, we should use the (commented out) value in Line 52.

Now, we have defined all services, and our Docker compose file should be complete. We are ready to run.

Building and running all or individual services

Now that we have prepared a realistic application consisting of more than one service, it is time that we leverage the Docker compose tool to its fullest. First, we will run the `database` service and use the PgAdmin4 to explore and configure it:

Preparing the database

Note we have configured our APIs such that they create the database schema and some initial data when they are started up. Thus, what we are showing next is more like a demonstration on how you can access and manage our database running in a container.

1. First, use the Docker compose command to run the database service in detach mode:

   ```
   docker compose up -d database
   ```

2. And then run the `pgadmin` service, again with the help of Docker compose:

   ```
   docker compose up -d pgadmin
   ```

   You will have to wait a few moments until `pgadmin` is ready. You may want to use the Docker Desktop dashboard to inspect the logs of `pgadmin` to find out if it is read.

3. Open a new browser window and navigate to `http://localhost:5050/login` and enter the credentials to login into PgAdmin4 that you have defined in the `docker-compose.yml` file. In our case, this should be `pgadmin@pgadmin.org` for the email and `Top-Secret` for the password.

4. Add a new database server, and in the dialog box that pops up, use the following values:

   a. On Tab **General,** fill out the name:

      i. **Name**: Demo

   b. On tab **Connection,** fill in these values:

      i. **Host name/Address**: database

In Line 51, we define that the Web service is dependent on the `api-java` service. Once again, if we want to use the .NET-based API, we should use the (commented out) value in Line 52.

Now, we have defined all services, and our Docker compose file should be complete. We are ready to run.

Building and running all or individual services

Now that we have prepared a realistic application consisting of more than one service, it is time that we leverage the Docker compose tool to its fullest. First, we will run the `database` service and use the PgAdmin4 to explore and configure it:

Preparing the database

Note we have configured our APIs such that they create the database schema and some initial data when they are started up. Thus, what we are showing next is more like a demonstration on how you can access and manage our database running in a container.

1. First, use the Docker compose command to run the database service in detach mode:

   ```
   docker compose up -d database
   ```

2. And then run the `pgadmin` service, again with the help of Docker compose:

   ```
   docker compose up -d pgadmin
   ```

   You will have to wait a few moments until `pgadmin` is ready. You may want to use the Docker Desktop dashboard to inspect the logs of `pgadmin` to find out if it is read.

3. Open a new browser window and navigate to `http://localhost:5050/login` and enter the credentials to login into PgAdmin4 that you have defined in the `docker-compose.yml` file. In our case, this should be `pgadmin@pgadmin.org` for the email and `Top-Secret` for the password.

4. Add a new database server, and in the dialog box that pops up, use the following values:

   a. On Tab **General,** fill out the name:

      i. **Name**: Demo

   b. On tab **Connection,** fill in these values:

      i. **Host name/Address**: database

ii. **Port**: 5432

iii. **Maintenance database:** demo-database

iv. **Username**: demo-user

v. **Password:** Top-Secret

In the following image, you can see the tab **Connection** of the new database server dialog:



Figure 7.34: Connection settings in PgAdmin

Note that we need to give the value **database** as hostname since that is how we called the Postgres service in our Docker compose file, and since PgAdmin4 runs on the same Docker network as the database, this is also the DNS name PgAdmin4 should use to connect to the DB.

4. Now, you will be able to use PgAdmin4 to manage the Postgres database. Expand the **Demo** node and then the **Databases** node. You should see the **demo-database** that was automatically created by Postgres upon starting the container (we specified it in our Docker compose file, remember?).

ii. **Port**: 5432

iii. **Maintenance database:** demo-database

iv. **Username**: demo-user

v. **Password:** Top-Secret

In the following image, you can see the tab **Connection** of the new database server dialog:



Figure 7.34: Connection settings in PgAdmin

Note that we need to give the value **database** as hostname since that is how we called the Postgres service in our Docker compose file, and since PgAdmin4 runs on the same Docker network as the database, this is also the DNS name PgAdmin4 should use to connect to the DB.

4. Now, you will be able to use PgAdmin4 to manage the Postgres database. Expand the **Demo** node and then the **Databases** node. You should see the **demo-database** that was automatically created by Postgres upon starting the container (we specified it in our Docker compose file, remember?).

5. Expand the **Schemas** node and then locate the **Tables** node. Right-click on the **Tables** node and select **Create->Table** in the context menu to create a new database.

6. On the **General** tab of the **Create – Table** dialog box, enter **`Orders`** as the name of the table. Add three columns **`Id`**, **`OrderDate`**, and **`Status`**, to this table, as shown in the following image:

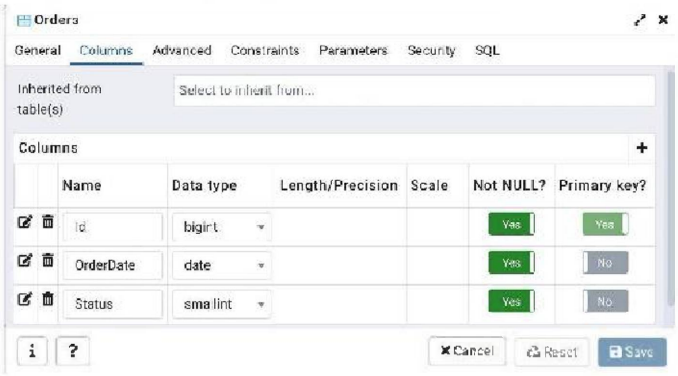


Figure 7.35: Columns of the orders table

When done, click on **Save**.

7. Use PgAdmin4 to add a few records to the **`Orders`** table. Make sure that the **`Status`** field contains a valid value between 1 and 3 (1 = Draft, 2 = Accepted, and 3 = Fulfilled). An example is given as follows:

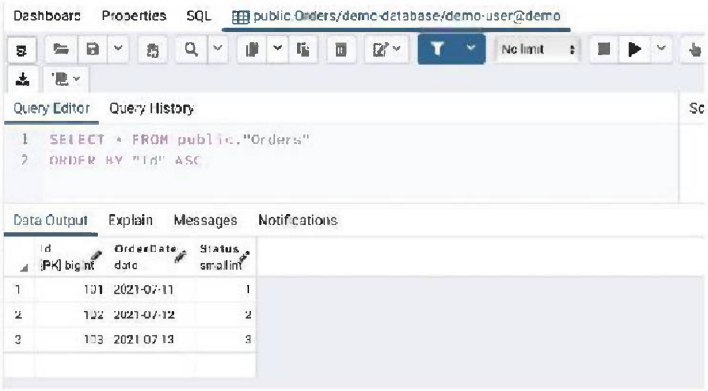


Figure 7.36: Some sample rows in the orders table

5. Expand the **Schemas** node and then locate the **Tables** node. Right-click on the **Tables** node and select **Create->Table** in the context menu to create a new database.

6. On the **General** tab of the **Create – Table** dialog box, enter **`Orders`** as the name of the table. Add three columns **`Id`**, **`OrderDate`**, and **`Status`**, to this table, as shown in the following image:

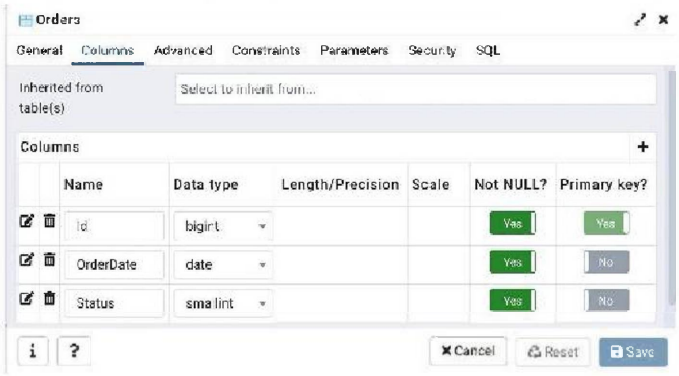


Figure 7.35: Columns of the orders table

When done, click on **Save**.

7. Use PgAdmin4 to add a few records to the **`Orders`** table. Make sure that the **`Status`** field contains a valid value between 1 and 3 (1 = Draft, 2 = Accepted, and 3 = Fulfilled). An example is given as follows:

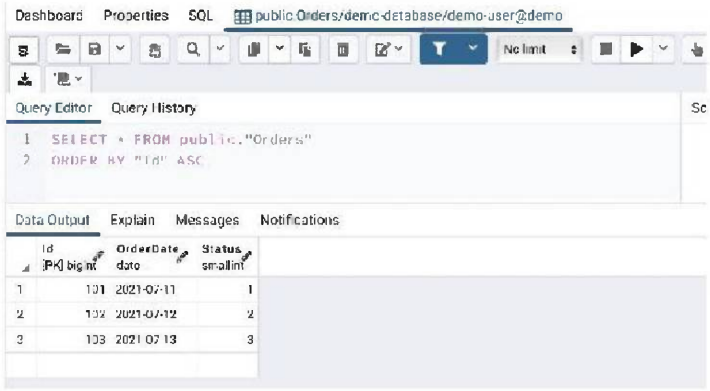


Figure 7.36: Some sample rows in the orders table

Alternatively, you can also use Docker compose to run a shell inside the PostgreSQL container and use the `psql` command line tool to prepare the database as follows:

1. In the terminal, use the following command to run a Bash shell inside the database container:

   ```
   docker compose exec database bash
   ```

2. Run the `psql` command line tool:

   ```
   psql -U demo-user --password demo-database
   ```

   Note that you will be prompted to enter the password, which in our case should be `Top-Secret`. Once you enter this password, you should be ready to prepare the `demo-database`.

3. Create the orders table with the following command:

   ```
   create table "Orders"(
     "Id" bigint not null primary key,
     "OrderDate" date not null,
     "Status" smallint not null
   );
   ```

4. Try to insert a few records into the table, such as:

   ```
   insert into "Orders"("Id", "OrderDate", "Status")
     values(101, '2021-07-11', 1);
   insert into "Orders"("Id", "OrderDate", "Status")
     values(102, '2021-07-12', 2);
   insert into "Orders"("Id", "OrderDate", "Status")
     values(103, '2021-07-13', 3);
   ...
   ```

5. And then, list all records in the table with the following:

   ```
   select * from "Orders";
   ```

   The records you just inserted should be displayed.

6. When done, exit `psql` with `CTRL-d` and then the `database` container also with CTRL-d.

Next, we will use it to build Docker images for our services from the source code.

Alternatively, you can also use Docker compose to run a shell inside the PostgreSQL container and use the `psql` command line tool to prepare the database as follows:

1. In the terminal, use the following command to run a Bash shell inside the database container:

   ```
   docker compose exec database bash
   ```

2. Run the `psql` command line tool:

   ```
   psql -U demo-user --password demo-database
   ```

   Note that you will be prompted to enter the password, which in our case should be `Top-Secret`. Once you enter this password, you should be ready to prepare the `demo-database`.

3. Create the orders table with the following command:

   ```
   create table "Orders"(
     "Id" bigint not null primary key,
     "OrderDate" date not null,
     "Status" smallint not null
   );
   ```

4. Try to insert a few records into the table, such as:

   ```
   insert into "Orders"("Id", "OrderDate", "Status")
     values(101, '2021-07-11', 1);
   insert into "Orders"("Id", "OrderDate", "Status")
     values(102, '2021-07-12', 2);
   insert into "Orders"("Id", "OrderDate", "Status")
     values(103, '2021-07-13', 3);
   ...
   ```

5. And then, list all records in the table with the following:

   ```
   select * from "Orders";
   ```

   The records you just inserted should be displayed.

6. When done, exit `psql` with `CTRL-d` and then the `database` container also with CTRL-d.

Next, we will use it to build Docker images for our services from the source code.

Building service with Docker compose

To build all services referenced in a Docker compose file and have a build node, we can just use the command:

```
docker compose build
```

Docker will then try to locate the first services, which has a `build` node and next locate its `Dockerfile` and use it to build that service. This will be repeated until all services having a `build` node are built. In our case, both the `web` and the `api` service have such a build node and thus will be built by the preceding command.

If we repeat the preceding command without that anything in the underlying source code has changed, then Docker compose will use the cached results, and the build will be fast(er). It is similar to what we have learned when discussing the Docker build command.

When building the image, Docker compose will use the value in the `image` node of the respective service to name the resulting image. In our case, for the `web` service, the resulting image name would be `demo-web:0.1`. Similarly, for the `api` service, the image name would be `demo-api:0.1`. You can double-check this by using the `docker image ls` command to list all images.

At times we might only want to build an individual service. In this case, we can use this command:

```
docker compose build <service-name>
```

where `<service-name>` is the name of the service in the Docker compose file that we want to build. In our case, this could be `web`, `api-java` or `api-dotnet`.

Now that we know how to build Docker images from the source, we may also want to push them to some registry for further use. Let us discuss this next.

Pushing images to a registry

To be able to share images built on your computer with other users or environments, we have to push them to an image registry such as Docker Hub. Let us do this now:

1. Make sure you are logged in to Docker Hub. You can do this with the following command:

   ```
   docker login
   ```

   You will be prompted to enter your username and password.

   Alternatively, you can login on to the Docker Hub Web page: **https://hub.docker.com**.

Building service with Docker compose

To build all services referenced in a Docker compose file and have a build node, we can just use the command:

```
docker compose build
```

Docker will then try to locate the first services, which has a `build` node and next locate its `Dockerfile` and use it to build that service. This will be repeated until all services having a `build` node are built. In our case, both the `web` and the `api` service have such a build node and thus will be built by the preceding command.

If we repeat the preceding command without that anything in the underlying source code has changed, then Docker compose will use the cached results, and the build will be fast(er). It is similar to what we have learned when discussing the Docker build command.

When building the image, Docker compose will use the value in the `image` node of the respective service to name the resulting image. In our case, for the `web` service, the resulting image name would be `demo-web:0.1`. Similarly, for the `api` service, the image name would be `demo-api:0.1`. You can double-check this by using the `docker image ls` command to list all images.

At times we might only want to build an individual service. In this case, we can use this command:

```
docker compose build <service-name>
```

where `<service-name>` is the name of the service in the Docker compose file that we want to build. In our case, this could be `web`, `api-java` or `api-dotnet`.

Now that we know how to build Docker images from the source, we may also want to push them to some registry for further use. Let us discuss this next.

Pushing images to a registry

To be able to share images built on your computer with other users or environments, we have to push them to an image registry such as Docker Hub. Let us do this now:

1. Make sure you are logged in to Docker Hub. You can do this with the following command:

   ```
   docker login
   ```

   You will be prompted to enter your username and password.

   Alternatively, you can login on to the Docker Hub Web page: **https://hub.docker.com**.

2. Use Docker compose to push all images at once to the registry:

   ```
   docker compose push
   ```

   The output of the preceding command will look very similar to what you saw when you were using the Docker image push command earlier on in *Chapter 5, Creating Your Own Docker Image*.

3. Alternatively, you can also push individual services to the registry with the following:

   ```
   docker compose push <service name>
   ```

   where in our case, `<service name>` would be either `api-java`, `api-dotnet`, or `web`.

Now it is time to run our application.

Running an application

Earlier, in section "Preparing the database" we have already used Docker compose to run the `database` and the `pgadmin` services. Let us now dive a bit deeper into this command and its possibilities.

Assuming we have a Docker compose file somewhere which defines declaratively our multi-service application, then we can just simply run the following command to run the whole application:

```
docker compose up -d
```

If the images of the services are not yet found in the local image cache, then Docker compose will automatically download them from the image registry or build them first if the corresponding service has a build instruction in it—as do the `web` and the `api-java` (or `api-dotnet`) service in our case.

Once all images are available, Docker compose will create the necessary network(s) and create a container instance for each of the services and attach those containers to the network(s).

In the preceding command, we used the `-d` parameter to run all services in detach (that is, non-blocking) mode.

Often people define the Docker compose file without any `depends_on` statements. In that case, there is no specific order in which the containers are created other than from the top down. But be aware that in our case, dependencies between containers have been defined, such as, for example, that service `api-java` is dependent on the service `database`. We will have a look at this option later in this chapter.

2. Use Docker compose to push all images at once to the registry:

   ```
   docker compose push
   ```

   The output of the preceding command will look very similar to what you saw when you were using the Docker image push command earlier on in *Chapter 5, Creating Your Own Docker Image*.

3. Alternatively, you can also push individual services to the registry with the following:

   ```
   docker compose push <service name>
   ```

   where in our case, `<service name>` would be either `api-java`, `api-dotnet`, or `web`.

Now it is time to run our application.

Running an application

Earlier, in section "Preparing the database" we have already used Docker compose to run the `database` and the `pgadmin` services. Let us now dive a bit deeper into this command and its possibilities.

Assuming we have a Docker compose file somewhere which defines declaratively our multi-service application, then we can just simply run the following command to run the whole application:

```
docker compose up -d
```

If the images of the services are not yet found in the local image cache, then Docker compose will automatically download them from the image registry or build them first if the corresponding service has a build instruction in it—as do the `web` and the `api-java` (or `api-dotnet`) service in our case.

Once all images are available, Docker compose will create the necessary network(s) and create a container instance for each of the services and attach those containers to the network(s).

In the preceding command, we used the `-d` parameter to run all services in detach (that is, non-blocking) mode.

Often people define the Docker compose file without any `depends_on` statements. In that case, there is no specific order in which the containers are created other than from the top down. But be aware that in our case, dependencies between containers have been defined, such as, for example, that service `api-java` is dependent on the service `database`. We will have a look at this option later in this chapter.

If we are done with our application and want to stop it, we can use the following command:

```
docker compose down
```

This will try to stop and then remove all containers of the application and delete the network(s) created for this app. If we are using volumes with our services and want to have them removed too when we tear down the application, we can use the parameter -v:

```
docker compose down -v
```

Be careful, though; after the preceding command, all data that has been generated by the application is gone for good!

Now, let us use our demo application to test these commands:

1. Open a new terminal window and navigate to the `demo-app` folder
2. Execute the following command to start (all the services of) the demo application:

   ```
   docker compose up --build
   ```

   Notice that we are forcing a rebuild of all containers using the `--build` parameter and purposefully not running in detach mode, so you can see the log output generated by each service.
3. Open a browser tab and navigate to `http://localhost:5050` and use PgAdmin4 to analyze what data is generated by the application and to be able to manipulate this data if wanted.
4. Open a browser tab and navigate to `http://localhost:3000/orders` and observe the log entries generated in the terminal window as you do so. In your browser window, you should see something like the following:

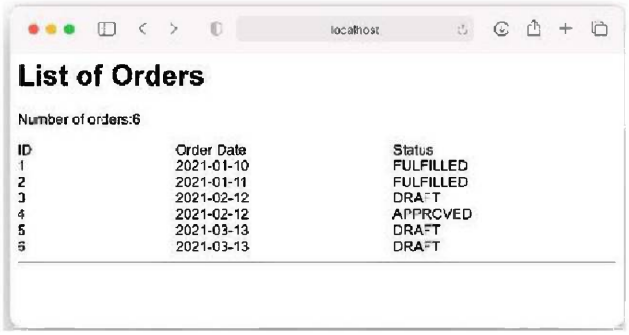


Figure 7.37: The demo application in action

If we are done with our application and want to stop it, we can use the following command:

```
docker compose down
```

This will try to stop and then remove all containers of the application and delete the network(s) created for this app. If we are using volumes with our services and want to have them removed too when we tear down the application, we can use the parameter -v:

```
docker compose down -v
```

Be careful, though; after the preceding command, all data that has been generated by the application is gone for good!

Now, let us use our demo application to test these commands:

1. Open a new terminal window and navigate to the `demo-app` folder
2. Execute the following command to start (all the services of) the demo application:

   ```
   docker compose up --build
   ```

   Notice that we are forcing a rebuild of all containers using the `--build` parameter and purposefully not running in detach mode, so you can see the log output generated by each service.
3. Open a browser tab and navigate to `http://localhost:5050` and use PgAdmin4 to analyze what data is generated by the application and to be able to manipulate this data if wanted.
4. Open a browser tab and navigate to `http://localhost:3000/orders` and observe the log entries generated in the terminal window as you do so. In your browser window, you should see something like the following:

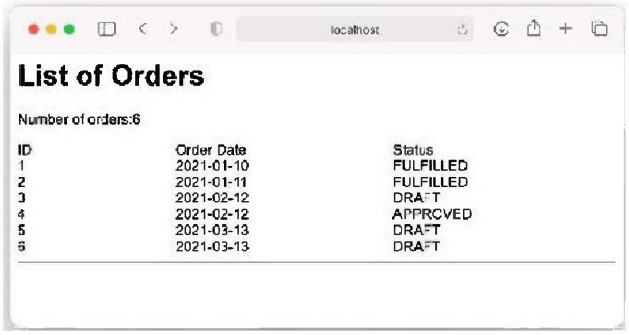


Figure 7.37: The demo application in action

With this, we have demonstrated that the application consisting of multiple services is up and running.

Optionally open another browser tab and navigate to **`http://localhost:8080/orders`** to directly access the REST API. You should see the following:



Figure 7.38: Accessing the API directly from the browser

5. Once you are done, in the terminal, hit CTRL-c to stop the application
6. Tear down the whole application with the following:

   ```
   docker compose down -v
   ```

So far, we have started and stopped our multi-service application as a whole. Sometimes we do want to have a bit more fine-grained control about what to run and what not. Let us discuss this next.

Running, restarting, stopping, and removing individual services

With Docker compose, we can also run, stop, remove, or restart individual services of our multi-service application. For this, we use the following command:

```
docker compose up -d <service-name>
```

where **`<service-name>`** is the name of the service we want to run.

As a developer or software engineer, you know very well that our code is not always behaving the way we are expecting it to do. Then, we need to debug our services. One way of debugging is to look at logging information produced by our application and hope to find some clues inside the logs. Luckily, the Docker compose tool can help us to access such logging information.

With this, we have demonstrated that the application consisting of multiple services is up and running.

Optionally open another browser tab and navigate to **`http://localhost:8080/orders`** to directly access the REST API. You should see the following:



Figure 7.38: Accessing the API directly from the browser

5. Once you are done, in the terminal, hit CTRL-c to stop the application
6. Tear down the whole application with the following:

   ```
   docker compose down -v
   ```

So far, we have started and stopped our multi-service application as a whole. Sometimes we do want to have a bit more fine-grained control about what to run and what not. Let us discuss this next.

Running, restarting, stopping, and removing individual services

With Docker compose, we can also run, stop, remove, or restart individual services of our multi-service application. For this, we use the following command:

```
docker compose up -d <service-name>
```

where **`<service-name>`** is the name of the service we want to run.

As a developer or software engineer, you know very well that our code is not always behaving the way we are expecting it to do. Then, we need to debug our services. One way of debugging is to look at logging information produced by our application and hope to find some clues inside the logs. Luckily, the Docker compose tool can help us to access such logging information.

Logging with Docker compose

If we run our application with `docker compose up -d` in detach mode, then we cannot directly see the log output generated by each service. But no worries, we have the means to retrieve the log entries at any time.

1. Run our application in detach with the following:

   ```
   docker compose up -d
   ```

2. Now retrieve the logs with

   ```
   docker compose logs
   ```

   Notice how the log output of each service is shown. The output uses a different color for each service to make identification simpler for us. Also, note that all log entries from the very beginning are shown, and the output stops with the most recent entry of each service.

3. If you want to follow the log that does not end the command but wait for new arriving entries, you can use the following command:

   ```
   docker compose logs -f
   ```

   When done, hit CTRL-c to stop following the logs.

4. We can also just list the tail of the log with the following:

   ```
   docker compose -t <num-entries>
   ```

   where `<num-entries>` is the number of log entries we want to see from the tail, e.g.,

   ```
   docker compose -t 10
   ```

   to view the last 10 generated log entries.

5. If you only want to see the last five log entries of say service `api-java` and follow the log you can use the following command:

   ```
   docker compose logs -t 5 -f api-java
   ```

We have used many of the most common commands provided by the Docker compose tool to manage our multi-service application.

Advanced usages of Docker compose

So far, we have worked with the most basic and common commands of the Docker compose tool. The tool has a few more tricks up its sleeves, though. Let us investigate some.

Logging with Docker compose

If we run our application with `docker compose up -d` in detach mode, then we cannot directly see the log output generated by each service. But no worries, we have the means to retrieve the log entries at any time.

1. Run our application in detach with the following:

   ```
   docker compose up -d
   ```

2. Now retrieve the logs with

   ```
   docker compose logs
   ```

   Notice how the log output of each service is shown. The output uses a different color for each service to make identification simpler for us. Also, note that all log entries from the very beginning are shown, and the output stops with the most recent entry of each service.

3. If you want to follow the log that does not end the command but wait for new arriving entries, you can use the following command:

   ```
   docker compose logs -f
   ```

   When done, hit CTRL-c to stop following the logs.

4. We can also just list the tail of the log with the following:

   ```
   docker compose -t <num-entries>
   ```

   where `<num-entries>` is the number of log entries we want to see from the tail, e.g.,

   ```
   docker compose -t 10
   ```

   to view the last 10 generated log entries.

5. If you only want to see the last five log entries of say service `api-java` and follow the log you can use the following command:

   ```
   docker compose logs -t 5 -f api-java
   ```

We have used many of the most common commands provided by the Docker compose tool to manage our multi-service application.

Advanced usages of Docker compose

So far, we have worked with the most basic and common commands of the Docker compose tool. The tool has a few more tricks up its sleeves, though. Let us investigate some.

Defining dependencies between services

Sometimes we want to run a service that is dependent of other services that are part of our overall application. For example, our API service is dependent on the database service. To make sure that the database is running, if we start the API service, we can declare a dependency in the Docker-compose file. Let us look at the following steps:

1. In VS Code, open the **`docker-compose.yml`** file and add the following code snippet in Lines 22 and 23 to the definition of the **`api-java`** service (if it is not already there):

```
19  api-java:
20    image: dockerforjobseekers/api-java:0.1
21    build: api-java
22    depends_on:
23      - database
24    environment:
25      - SPRING_DATASOURCE_URL=jdbc:postgresql://database:5432/demo-database
26      - SPRING_DATASOURCE_USERNAME=demo-user
27      - SPRING_DATASOURCE_PASSWORD=Top Secret
28      - SPRING_JPA_HIBERNATE_DDL_AUTO=update
29    ports:
30      - 8080:8080
```

Figure 7.39: Defining a dependency on the database for the API service

This defines that the **`api-java`** service is dependent on the **`database`** service, and Docker compose will make sure that the **`database`** service is started before the **`api-java`** service is started.

2. Add a similar snippet to the **`web`** service declaring a dependency on the **`api-java`** service:

```
42  web:
43    image: dockerforjobseekers/demo-web:0.1
44    build: web
45    ports:
46      - 3000:3000
47    environment:
48      API_ENDPOINT: http://api-java:8080   # Java API
49      # API_ENDPOINT: http://api-dotnet:5000   # .NET API
50    depends_on:
51      - api-java
52      #- api-dotnet
```

Figure 7.40: Defining a dependency to the api-java for the Web service

Defining dependencies between services

Sometimes we want to run a service that is dependent of other services that are part of our overall application. For example, our API service is dependent on the database service. To make sure that the database is running, if we start the API service, we can declare a dependency in the Docker-compose file. Let us look at the following steps:

1. In VS Code, open the **`docker-compose.yml`** file and add the following code snippet in Lines 22 and 23 to the definition of the **`api-java`** service (if it is not already there):

```
19  api-java:
20    image: dockerforjobseekers/api-java:0.1
21    build: api-java
22    depends_on:
23      - database
24    environment:
25      - SPRING_DATASOURCE_URL=jdbc:postgresql://database:5432/demo-database
26      - SPRING_DATASOURCE_USERNAME=demo-user
27      - SPRING_DATASOURCE_PASSWORD=Top Secret
28      - SPRING_JPA_HIBERNATE_DDL_AUTO=update
29    ports:
30      - 8080:8080
```

Figure 7.39: Defining a dependency on the database for the API service

This defines that the **`api-java`** service is dependent on the **`database`** service, and Docker compose will make sure that the **`database`** service is started before the **`api-java`** service is started.

2. Add a similar snippet to the **`web`** service declaring a dependency on the **`api-java`** service:

```
42  web:
43    image: dockerforjobseekers/demo-web:0.1
44    build: web
45    ports:
46      - 3000:3000
47    environment:
48      API_ENDPOINT: http://api-java:8080   # Java API
49      # API_ENDPOINT: http://api-dotnet:5000   # .NET API
50    depends_on:
51      - api-java
52      #- api-dotnet
```

Figure 7.40: Defining a dependency to the api-java for the Web service

Since we want to use the Java-based API, we have Line 51 active. If we use the .NET-based API, we need to comment Line 51 and uncomment Line 52 instead.

3. To test this, make sure you can start with a clean slate by using the following command to remove all running services:

   ```
   docker compose down -v
   ```

4. Now start the API service in detach mode with the following command:

   ```
   docker compose -d api-java
   ```

   Notice how Docker Compose automatically first starts the `database` service and then only starts the `api-java` service:

```
gabriel@Solaris demo-app % docker compose up -d api-java
[+] Running 2/2
 ⠿ Container demo-app_database_1  Started                                    2.0s
 ⠿ Container demo-app_api-java_1  Started                                    5.5s
```

Figure 7.41: Starting the api service, which is dependent on the database service

5. Double-check that the `api-java` and the `database` services are running:

   ```
   docker compose ps
   ```

   You should see something like the following:

```
gabriel@Solaris demo-app % docker compose ps
NAME                SERVICE             STATUS              PORTS
demo-app_api_1      api                 running             0.0.0.0:5000->80/tcp, :::5000->80/tcp
demo-app_database_1 database            running             5432/tcp
```

Figure 7.42: List of running services

Notice that both service show status `running`, as expected.

6. Use `curl` to access the API. You will be accessing the `/orders` endpoint to get a list of all draft orders:

   ```
   curl -X GET http://localhost:8080/orders
   ```

   You should get a list of draft orders similar to the following:

```
gabriel@Solaris demo-app % curl -X GET http://localhost:8080/orders

[{"id":1,"orderDate":"2021-01-10","status":"FULFILLED"},{"id":2,"orderDate":"2021-01-11","s
tatus":"FULFILLED"},{"id":3,"orderDate":"2021-02-12","status":"DRAFT"},{"id":4,"orderDate":
"2021-02-12","status":"APPROVED"},{"id":5,"orderDate":"2021-03-13","status":"DRAFT"},{"id":
6,"orderDate":"2021-03-13","status":"DRAFT"}]%
```

Figure 7.43: Getting the list of pending orders from the API service

The fact that you are actually getting the list is another proof for the fact that the database is running.

Since we want to use the Java-based API, we have Line 51 active. If we use the .NET-based API, we need to comment Line 51 and uncomment Line 52 instead.

3. To test this, make sure you can start with a clean slate by using the following command to remove all running services:

   ```
   docker compose down -v
   ```

4. Now start the API service in detach mode with the following command:

   ```
   docker compose -d api-java
   ```

   Notice how Docker Compose automatically first starts the `database` service and then only starts the `api-java` service:

```
gabriel@Solaris demo-app % docker compose up -d api-java
[+] Running 2/2
 ⠿ Container demo-app_database_1  Started                                    2.0s
 ⠿ Container demo-app_api-java_1  Started                                    5.5s
```

Figure 7.41: Starting the api service, which is dependent on the database service

5. Double-check that the `api-java` and the `database` services are running:

   ```
   docker compose ps
   ```

   You should see something like the following:

```
gabriel@Solaris demo-app % docker compose ps
NAME                SERVICE             STATUS              PORTS
demo-app_api_1      api                 running             0.0.0.0:5000->80/tcp, :::5000->80/tcp
demo-app_database_1 database            running             5432/tcp
```

Figure 7.42: List of running services

Notice that both service show status `running`, as expected.

6. Use `curl` to access the API. You will be accessing the `/orders` endpoint to get a list of all draft orders:

   ```
   curl -X GET http://localhost:8080/orders
   ```

   You should get a list of draft orders similar to the following:

```
gabriel@Solaris demo-app % curl -X GET http://localhost:8080/orders

[{"id":1,"orderDate":"2021-01-10","status":"FULFILLED"},{"id":2,"orderDate":"2021-01-11","s
tatus":"FULFILLED"},{"id":3,"orderDate":"2021-02-12","status":"DRAFT"},{"id":4,"orderDate":
"2021-02-12","status":"APPROVED"},{"id":5,"orderDate":"2021-03-13","status":"DRAFT"},{"id":
6,"orderDate":"2021-03-13","status":"DRAFT"}]%
```

Figure 7.43: Getting the list of pending orders from the API service

The fact that you are actually getting the list is another proof for the fact that the database is running.

7. End and remove the two services with the following:

   ```
   docker compose down -v
   ```

8. Now run the Web service in detach mode with the following:

   ```
   docker compose -d web
   ```

   Notice again how Docker Compose automatically first starts the `database` service and then the `api-java` service before it finally starts the Web service. All this happens since you defined the dependencies of each service in the Docker-compose file.

9. Open a new browser tab and navigate to `http://localhost:3000/orders` to make sure the application works as expected.

10. End the three services with the following:

    ```
    docker compose down -v
    ```

Great, you have just learned how to define dependencies between services in a Docker compose the application. Next, you will learn how you can leverage the tool to scale the individual services of the application.

Scaling a service

We all know that one technique to cope with increased traffic or demand for resources in an application is to scale out the whole application or individual parts of it. This is no different when using containers. Containerized services are very easy to scale up and down as long as they are stateless. Stateful services such as databases are a bit more involved. But this is beyond the scope of this book.

Docker compose makes it quite straightforward to scale a particular service. Let us do this with the `api` service of our application. If we want to scale the `api` service up to three instances, we can do so like the following:

1. Start our demo-app application in detach mode:

   ```
   docker compose up -d
   ```

2. List all services of the application and make sure they are all up and running:

   ```
   docker compose ps
   ```

   you should see this:

```
gabriel@Solaris demo-app % docker compose ps
NAME                  SERVICE             STATUS              PORTS
demo-app_api-dotnet_1   api-dotnet          running             0.0.0.0:5000->80/tcp, :::5000->80/tcp
demo-app_api-java_1     api-java            running             0.0.0.0:8080->8080/tcp, :::8080->8080/tcp
demo-app_database_1     database            running             5432/tcp
demo-app_pgadmin_1      pgadmin             running             443/tcp, 0.0.0.0:5050->80/tcp, :::5050->80/tcp
demo-app_web_1          web                 running             0.0.0.0:3000->3000/tcp, :::3000->3000/tcp
```

Figure 7.44: List of services of the demo-app

7. End and remove the two services with the following:

   ```
   docker compose down -v
   ```

8. Now run the Web service in detach mode with the following:

   ```
   docker compose -d web
   ```

   Notice again how Docker Compose automatically first starts the `database` service and then the `api-java` service before it finally starts the Web service. All this happens since you defined the dependencies of each service in the Docker-compose file.

9. Open a new browser tab and navigate to `http://localhost:3000/orders` to make sure the application works as expected.

10. End the three services with the following:

    ```
    docker compose down -v
    ```

Great, you have just learned how to define dependencies between services in a Docker compose the application. Next, you will learn how you can leverage the tool to scale the individual services of the application.

Scaling a service

We all know that one technique to cope with increased traffic or demand for resources in an application is to scale out the whole application or individual parts of it. This is no different when using containers. Containerized services are very easy to scale up and down as long as they are stateless. Stateful services such as databases are a bit more involved. But this is beyond the scope of this book.

Docker compose makes it quite straightforward to scale a particular service. Let us do this with the `api` service of our application. If we want to scale the `api` service up to three instances, we can do so like the following:

1. Start our demo-app application in detach mode:

   ```
   docker compose up -d
   ```

2. List all services of the application and make sure they are all up and running:

   ```
   docker compose ps
   ```

   you should see this:

```
gabriel@Solaris demo-app % docker compose ps
NAME                    SERVICE             STATUS              PORTS
demo-app_api-dotnet_1   api-dotnet          running             0.0.0.0:5000->80/tcp, :::5000->80/tcp
demo-app_api-java_1     api-java            running             0.0.0.0:8080->8080/tcp, :::8080->8080/tcp
demo-app_database_1     database            running             5432/tcp
demo-app_pgadmin_1      pgadmin             running             443/tcp, 0.0.0.0:5050->80/tcp, :::5050->80/tcp
demo-app_web_1          web                 running             0.0.0.0:3000->3000/tcp, :::3000->3000/tcp
```

Figure 7.44: List of services of the demo-app

3. Now use Docker compose to scale the `api-java` service to three instances:

```
docker-compose up --scale api-java=3
```

NOTE: In the preceding command, we use the docker-compose tool instead of the Docker compose command since, at the time of writing, the latter has an error, and the scaling does not work correctly. Maybe at the time you read this, the bug has been fixed. Thus, try both variants.

You will notice immediately that when executing the preceding command, we get an error claiming something like the following:

```
Error response from daemon: Ports are not available: listen tcp
0.0.0.0:8080: bind: address already in use
```

What happened? The reason for this is that we have (for debugging purposes) mapped the container port of the API service to the host. But we cannot bind three different container ports all to the same host port. That is the reason for this failure.

As a solution, we can just commend out the port mapping of the `api-java` service in our `docker-compose.yml` file. Do this and then repeat the scale command. This time it should work.

4. To confirm that, indeed, three instances are running, use the `ps` command again:

```
docker compose ps
```

and notice that the output has changed to:

```
gabriel@Solaris demo-app % docker compose ps
NAME                    SERVICE             STATUS              PORTS
demo-app_api-dotnet_1   api-dotnet          running             0.0.0.0:5000->80/tcp, :::5000->80/tcp
demo-app_api-java_1     api-java            running
demo-app_api-java_2     api-java            running
demo-app_api-java_3     api-java            running
demo-app_database_1     database            running             5432/tcp
demo-app_pgadmin_1      pgadmin             running             443/tcp, 0.0.0.0:5050->80/tcp, :::5050->80/tcp
demo-app_web_1          web                 running             0.0.0.0:3000->3000/tcp, :::3000->3000/tcp
```

Figure 7.45: API service scaled to three instances

5. Scale down the api-java service to two instances:

```
docker-compose up --scale api-java=2
```

Here again, we are using the `docker-compose` tool to avoid the bug mentioned in Point 3. Reconfirm with the `ps` command that the `api-java` service now runs with two instances.

6. When done, end the application with the following:

```
docker compose down -v
```

3. Now use Docker compose to scale the `api-java` service to three instances:

```
docker-compose up --scale api-java=3
```

NOTE: In the preceding command, we use the docker-compose tool instead of the Docker compose command since, at the time of writing, the latter has an error, and the scaling does not work correctly. Maybe at the time you read this, the bug has been fixed. Thus, try both variants.

You will notice immediately that when executing the preceding command, we get an error claiming something like the following:

```
Error response from daemon: Ports are not available: listen tcp
0.0.0.0:8080: bind: address already in use
```

What happened? The reason for this is that we have (for debugging purposes) mapped the container port of the API service to the host. But we cannot bind three different container ports all to the same host port. That is the reason for this failure.

As a solution, we can just commend out the port mapping of the `api-java` service in our `docker-compose.yml` file. Do this and then repeat the scale command. This time it should work.

4. To confirm that, indeed, three instances are running, use the `ps` command again:

```
docker compose ps
```

and notice that the output has changed to:

```
gabriel@Solaris demo-app % docker compose ps
NAME                    SERVICE             STATUS              PORTS
demo-app_api-dotnet_1   api-dotnet          running             0.0.0.0:5000->80/tcp, :::5000->80/tcp
demo-app_api-java_1     api-java            running
demo-app_api-java_2     api-java            running
demo-app_api-java_3     api-java            running
demo-app_database_1     database            running             5432/tcp
demo-app_pgadmin_1      pgadmin             running             443/tcp, 0.0.0.0:5050->80/tcp, :::5050->80/tcp
demo-app_web_1          web                 running             0.0.0.0:3000->3000/tcp, :::3000->3000/tcp
```

Figure 7.45: API service scaled to three instances

5. Scale down the api-java service to two instances:

```
docker-compose up --scale api-java=2
```

Here again, we are using the `docker-compose` tool to avoid the bug mentioned in Point 3. Reconfirm with the `ps` command that the `api-java` service now runs with two instances.

6. When done, end the application with the following:

```
docker compose down -v
```

Note that if you want to scale the `web` service the same way as you did with the `api-java` service, you will fail. And there is no way to circumvent the error as we did in preceding Point 3. Why? The reason is that the `web` service, as it is defined in the Docker compose file, uses some port mapping to map the container port 3000 to the host port 3000 (this is imperative; otherwise, no one can access the Web service). That works as long as we have only one instance of the Web service. As soon as we want to scale up, a port mapping conflict will occur since the second instance will also want to map its container port to host port 3000. But this is not possible. Each host port can only have one listener. This problem is due to the fact that we want to scale on a single Docker host. In a production system, you will have several Docker hosts forming a cluster. We can then spread out the instances of the Web service to the different nodes of the cluster, and on each one, the respective instance of the Web service can be mapped to the same host port. An alternative way would be to not map the container port to a fixed host port but let Docker decide which free host port to use for each instance.

Optional exercise: try to modify your Docker compose file such that Docker decides which host port the container port of the Web service is mapped to.

Next, we want to learn how to run multiple distinct instances of a Docker compose application simultaneously in multiple instances.

Defining a project name

We have learned that Docker compose, by default, uses the name of the folder in which the compose file is located as a prefix to the network names it creates. Thus, if we do not explicitly define a network in the compose file and the folder is called `my-app`, then the resulting network created by Docker compose is `my-app_default`. On the other hand, if we define a custom network and call it `backend-network`, the resulting name of the network created by Docker compose is `my-app_backend-network`. It is always a combination of the network name used in the compose file and the folder in which the file resides.

Now, this is fine as long as we want to only run one instance of the application defined in the compose file. If, for some reason, we want to run more than one instance, we will have a name clash. To avoid this, we can define a project name when using the Docker compose command as follows:

```
docker compose --project-name project-1 up -d
```

Please note the `--project-name` (or its short form `-p`) parameter. In this case, the default network created by the compose tool would be `project-1_default` (and the name of the containing folder would not matter).

We can then easily run a second instance of the application, e.g., with the following:

```
docker compose --project-name project-2 up -d
```

Note that if you want to scale the `web` service the same way as you did with the `api-java` service, you will fail. And there is no way to circumvent the error as we did in preceding Point 3. Why? The reason is that the `web` service, as it is defined in the Docker compose file, uses some port mapping to map the container port 3000 to the host port 3000 (this is imperative; otherwise, no one can access the Web service). That works as long as we have only one instance of the Web service. As soon as we want to scale up, a port mapping conflict will occur since the second instance will also want to map its container port to host port 3000. But this is not possible. Each host port can only have one listener. This problem is due to the fact that we want to scale on a single Docker host. In a production system, you will have several Docker hosts forming a cluster. We can then spread out the instances of the Web service to the different nodes of the cluster, and on each one, the respective instance of the Web service can be mapped to the same host port. An alternative way would be to not map the container port to a fixed host port but let Docker decide which free host port to use for each instance.

Optional exercise: try to modify your Docker compose file such that Docker decides which host port the container port of the Web service is mapped to.

Next, we want to learn how to run multiple distinct instances of a Docker compose application simultaneously in multiple instances.

Defining a project name

We have learned that Docker compose, by default, uses the name of the folder in which the compose file is located as a prefix to the network names it creates. Thus, if we do not explicitly define a network in the compose file and the folder is called `my-app`, then the resulting network created by Docker compose is `my-app_default`. On the other hand, if we define a custom network and call it `backend-network`, the resulting name of the network created by Docker compose is `my-app_backend-network`. It is always a combination of the network name used in the compose file and the folder in which the file resides.

Now, this is fine as long as we want to only run one instance of the application defined in the compose file. If, for some reason, we want to run more than one instance, we will have a name clash. To avoid this, we can define a project name when using the Docker compose command as follows:

```
docker compose --project-name project-1 up -d
```

Please note the `--project-name` (or its short form `-p`) parameter. In this case, the default network created by the compose tool would be `project-1_default` (and the name of the containing folder would not matter).

We can then easily run a second instance of the application, e.g., with the following:

```
docker compose --project-name project-2 up -d
```

And the tool will create a network `project-2_default` for the application and attach all containers to it. This way, the containers from the two application instances are perfectly isolated from each other, and no name clash occurs.

The next interesting use case is when dealing with multiple environments in which we want to run a Docker compose application.

Using Docker compose overrides

In a modern software development environment, teams use different environments to build, run and test their applications. Typically, the first environment a Docker compose application is supposed to run is, is the developer's personal computer or laptop. Once the developer is happy with their code, they push it to their source code repository, such as GitHub and create a merge request for it. The CI/CD pipeline then picks up the new code changes, and the build agent tries to build the application and run all automated tests against it. Thus, the build agent is a second environment where we might want to run our compose application. Later we may want to run the application on, say, a special environment where we can demo it to stakeholders or an environment where we can execute stress and load tests for the application. We will talk more about Docker and automated CI/CD pipelines in *Chapter 9, Establishing an automated build pipeline*.

Each of those environments most probably has slightly different environment settings. For example, the connection string that an application service needs differs from environment to environment. Or the network settings used by the Docker networks may also differ.

We could now just author a separate Docker compose file for each environment. But this would quickly result in code duplication and a danger that the different versions may be out of sync soon. We call this a drift. A better alternative is to author a main Docker compose file and use so-called overrides for each of the environments we want to run the application in.

Let us start with the base Docker compose file. We will take one similar to what we used in the previous sections to run our application consisting of a `web` service, an `api` service and a `database` service. We will give this file the default name `docker-compose.yml`. Here is its content:

And the tool will create a network `project-2_default` for the application and attach all containers to it. This way, the containers from the two application instances are perfectly isolated from each other, and no name clash occurs.

The next interesting use case is when dealing with multiple environments in which we want to run a Docker compose application.

Using Docker compose overrides

In a modern software development environment, teams use different environments to build, run and test their applications. Typically, the first environment a Docker compose application is supposed to run is, is the developer's personal computer or laptop. Once the developer is happy with their code, they push it to their source code repository, such as GitHub and create a merge request for it. The CI/CD pipeline then picks up the new code changes, and the build agent tries to build the application and run all automated tests against it. Thus, the build agent is a second environment where we might want to run our compose application. Later we may want to run the application on, say, a special environment where we can demo it to stakeholders or an environment where we can execute stress and load tests for the application. We will talk more about Docker and automated CI/CD pipelines in *Chapter 9, Establishing an automated build pipeline*.

Each of those environments most probably has slightly different environment settings. For example, the connection string that an application service needs differs from environment to environment. Or the network settings used by the Docker networks may also differ.

We could now just author a separate Docker compose file for each environment. But this would quickly result in code duplication and a danger that the different versions may be out of sync soon. We call this a drift. A better alternative is to author a main Docker compose file and use so-called overrides for each of the environments we want to run the application in.

Let us start with the base Docker compose file. We will take one similar to what we used in the previous sections to run our application consisting of a `web` service, an `api` service and a `database` service. We will give this file the default name `docker-compose.yml`. Here is its content:

```yaml
version: '2'
services:
    web:
        image: dockerforjobseekers/web:1.0
        ports:
            - 3000:3000

    api:
        image: dockerforjobseekers/api:1.0
        environment:
            connection_string: "postgresql://database/demo-database"
            db-username: demo-user
            db-password: Top-Secret

    database:
        image: postgres:alpine
        volumes:
            - demo-app-data:/var/lib/postgresql/data
        environment:
            POSTGRES_USER: demo-user
            POSTGRES_PASSWORD: Top-Secret

volumes:
    demo-app-data
```

Figure 7.46: Base Docker compose file

As we can see, the `web` service defines a port mapping and three environment variables `connection-string`, `db-username`, and `db-password`, with some default values. The `api` service does the same for variables `POSTGRES_USER` and `POSTGRES_PASSWORD`. Let us assume that these values are valid in local development, that is, on the developer's laptop. Furthermore, the base file defines no explicit network. All containers will be mounted to the `default` network automatically created by the Docker compose tool. Finally, all services will try to locate images either in the local image cache or try to pull them from the Docker hub. For example, the `web` service will try to use the `dockerforjobseekers/web:1.0` image.

Now, let us assume we want to define an override for the CI environment, that is, the build agent of our automated CI/CD pipeline. On a build server or agent, we usually want to build the Docker images from the source code. We will have to modify or override the compose file accordingly. Let us further assume that there we want to use a different password to access the database. For the sake of this example, let us also assume we want to use a different port mapping for the `web` service. Finally, we

```yaml
version: '2'
services:
    web:
        image: dockerforjobseekers/web:1.0
        ports:
            - 3000:3000

    api:
        image: dockerforjobseekers/api:1.0
        environment:
            connection_string: "postgresql://database/demo-database"
            db-username: demo-user
            db-password: Top-Secret

    database:
        image: postgres:alpine
        volumes:
            - demo-app-data:/var/lib/postgresql/data
        environment:
            POSTGRES_USER: demo-user
            POSTGRES_PASSWORD: Top-Secret

volumes:
    demo-app-data
```

Figure 7.46: Base Docker compose file

As we can see, the `web` service defines a port mapping and three environment variables `connection-string`, `db-username`, and `db-password`, with some default values. The `api` service does the same for variables `POSTGRES_USER` and `POSTGRES_PASSWORD`. Let us assume that these values are valid in local development, that is, on the developer's laptop. Furthermore, the base file defines no explicit network. All containers will be mounted to the `default` network automatically created by the Docker compose tool. Finally, all services will try to locate images either in the local image cache or try to pull them from the Docker hub. For example, the `web` service will try to use the `dockerforjobseekers/web:1.0` image.

Now, let us assume we want to define an override for the CI environment, that is, the build agent of our automated CI/CD pipeline. On a build server or agent, we usually want to build the Docker images from the source code. We will have to modify or override the compose file accordingly. Let us further assume that there we want to use a different password to access the database. For the sake of this example, let us also assume we want to use a different port mapping for the `web` service. Finally, we

want to use a custom network on the build agent. All this will lead to the following override that we will save in a file called **docker-compose-ci.yml**:

```
version: '2'
services:
    web:
        build: web
        ports:
            - 5000:3000
        networks:
            - ci-network
    api:
        build: api
        environment:
            POSTGRES_PASSWORD: ci-pass
        networks:
            - ci-network

    database:
        networks:
            - ci-network

networks:
    ci-network:
```

Figure 7.47: Docker compose override for the CI environment

In the override file, we only need to define entries whose values we want to override, such as the port mapping or the network settings.

We can then use the override by defining the compose files (using the parameter **-f**) in the right sequence, starting with the base file:

```
docker compose -f docker-compose.yml -f docker-compose-ci.yml up
```

Note that you can use as many override files as you like in the preceding command. Just make sure that you list them in the right sequence.

Conclusion

In this chapter, we have learned how to use Docker compose. This tool has been designed to simplify working with applications consisting of more than one containerized service. First, we have discussed the syntax of a Docker compose file, which is using a declarative approach. Then, we created an application consisting of three services, such as that we could experiment with the compose tool. We learned how to use the tool to build images, push them to a registry such as Docker Hub and run the whole application or individual services of it. Finally, we investigated some advanced usage scenarios for Docker compose, such as defining dependencies between the services and defining custom networks or overrides.

want to use a custom network on the build agent. All this will lead to the following override that we will save in a file called **docker-compose-ci.yml**:

```
version: '2'
services:
    web:
        build: web
        ports:
            - 5000:3000
        networks:
            - ci-network
    api:
        build: api
        environment:
            POSTGRES_PASSWORD: ci-pass
        networks:
            - ci-network

    database:
        networks:
            - ci-network

networks:
    ci-network:
```

Figure 7.47: Docker compose override for the CI environment

In the override file, we only need to define entries whose values we want to override, such as the port mapping or the network settings.

We can then use the override by defining the compose files (using the parameter **-f**) in the right sequence, starting with the base file:

```
docker compose -f docker-compose.yml -f docker-compose-ci.yml up
```

Note that you can use as many override files as you like in the preceding command. Just make sure that you list them in the right sequence.

Conclusion

In this chapter, we have learned how to use Docker compose. This tool has been designed to simplify working with applications consisting of more than one containerized service. First, we have discussed the syntax of a Docker compose file, which is using a declarative approach. Then, we created an application consisting of three services, such as that we could experiment with the compose tool. We learned how to use the tool to build images, push them to a registry such as Docker Hub and run the whole application or individual services of it. Finally, we investigated some advanced usage scenarios for Docker compose, such as defining dependencies between the services and defining custom networks or overrides.

Questions

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

1. What is the correct command when using Docker-compose to run a multi-service application in detach mode?
2. Write down the command you would use to list all the services of a Docker compose application.
3. How would you structure your Docker-compose files if you want to run a multi-service application in different environments, say a development, user acceptance testing (UAT) and production environment?
4. What can you do to scale up a particular service of a multi-service application to three instances using Docker compose?
5. Which of the following tasks can you execute with Docker compose:
    a. Building Docker images for all services of a multi-service application
    b. Building the Docker image for a single service
    c. Pushing the Docker images of all services
    d. Running a multi-service application with a single command
    e. Running a single service of a multi-service application
    f. All of the above

Job interview sample questions

You have been invited to a video interview for a job as a senior software developer. The job is offered by a renowned consulting company. During the interview, you are confronted with the following questions:

1. We are using Docker in our company. Our application consists of multiple services that interact with each other. What tool would you recommend us to use to run this application with ease on any developer's laptop. What would be valid alternatives, and why are they inferior or superior to the first one you name?
2. Should we use Docker compose to deploy our application in production? If not, what are the reasons to not leverage it for that scenario?
3. We currently use multiple Docker-compose files for different scenarios. This is a bit of a maintenance hassle. How could we improve the situation?
4. We have heard that Docker compose can also be used to deploy to Kubernetes. Is that true? What can you tell us about that feature?

Questions

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

1. What is the correct command when using Docker-compose to run a multi-service application in detach mode?
2. Write down the command you would use to list all the services of a Docker compose application.
3. How would you structure your Docker-compose files if you want to run a multi-service application in different environments, say a development, user acceptance testing (UAT) and production environment?
4. What can you do to scale up a particular service of a multi-service application to three instances using Docker compose?
5. Which of the following tasks can you execute with Docker compose:
    a. Building Docker images for all services of a multi-service application
    b. Building the Docker image for a single service
    c. Pushing the Docker images of all services
    d. Running a multi-service application with a single command
    e. Running a single service of a multi-service application
    f. All of the above

Job interview sample questions

You have been invited to a video interview for a job as a senior software developer. The job is offered by a renowned consulting company. During the interview, you are confronted with the following questions:

1. We are using Docker in our company. Our application consists of multiple services that interact with each other. What tool would you recommend us to use to run this application with ease on any developer's laptop. What would be valid alternatives, and why are they inferior or superior to the first one you name?
2. Should we use Docker compose to deploy our application in production? If not, what are the reasons to not leverage it for that scenario?
3. We currently use multiple Docker-compose files for different scenarios. This is a bit of a maintenance hassle. How could we improve the situation?
4. We have heard that Docker compose can also be used to deploy to Kubernetes. Is that true? What can you tell us about that feature?

Sample answers

Here are potential answers to the preceding questions.

1. Whenever a developer wants to run an application consisting of more than one container locally on her desktop, then she should consider using Docker compose. This is the preferred and most powerful tool for exactly this scenario. The alternative is to author bash or PowerShell scripts, but that would mean to replicate functionality that is already available for her out of the box with Docker Desktop.
2. Although it is technically possible to use Docker compose to deploy applications to production, we would not advise to do so. Usually, in a fully automated CI/CD (continuous integration/continues deliver or deployment), one deploys a single container at a time.
3. One could use a base compose file and use this file in different scenarios with overrides. One for each environment or scenario.
4. It is indeed true that Docker compose can be used to deploy to Kubernetes. The tool will analyze the compose file and automatically create the corresponding Kubernetes YAML files that are commonly used to deploy and manage artifacts to and on Kubernetes. It is an attractive means for testing your applications on a local Kubernetes cluster, such as the single node clusters provided by Minicube or Docker Desktop.

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



Sample answers

Here are potential answers to the preceding questions.

1. Whenever a developer wants to run an application consisting of more than one container locally on her desktop, then she should consider using Docker compose. This is the preferred and most powerful tool for exactly this scenario. The alternative is to author bash or PowerShell scripts, but that would mean to replicate functionality that is already available for her out of the box with Docker Desktop.
2. Although it is technically possible to use Docker compose to deploy applications to production, we would not advise to do so. Usually, in a fully automated CI/CD (continuous integration/continues deliver or deployment), one deploys a single container at a time.
3. One could use a base compose file and use this file in different scenarios with overrides. One for each environment or scenario.
4. It is indeed true that Docker compose can be used to deploy to Kubernetes. The tool will analyze the compose file and automatically create the corresponding Kubernetes YAML files that are commonly used to deploy and manage artifacts to and on Kubernetes. It is an attractive means for testing your applications on a local Kubernetes cluster, such as the single node clusters provided by Minicube or Docker Desktop.

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 8

Testing and Debugging Containerized Applications

Introduction

In the previous chapter, you got an in-depth introduction to Docker Compose and its usage. You built a complex sample application consisting of a Web UI, a backend API, and a database to demonstrate the capabilities of Docker compose in the context of a realistic distributed and containerized application.

In this chapter, you will learn how to significantly reduce friction in the development process by enabling hot reloading and debugging of an application running inside a container. You will also learn how to write and run automated tests for such an application and how to optimally instrument the application. Furthermore, you will get some useful 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.

Structure

In this chapter, we will discuss the following topics:

- Enabling hot reload in a container
- Debugging an app running inside a container

Chapter 8

Testing and Debugging Containerized Applications

Introduction

In the previous chapter, you got an in-depth introduction to Docker Compose and its usage. You built a complex sample application consisting of a Web UI, a backend API, and a database to demonstrate the capabilities of Docker compose in the context of a realistic distributed and containerized application.

In this chapter, you will learn how to significantly reduce friction in the development process by enabling hot reloading and debugging of an application running inside a container. You will also learn how to write and run automated tests for such an application and how to optimally instrument the application. Furthermore, you will get some useful 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.

Structure

In this chapter, we will discuss the following topics:

- Enabling hot reload in a container
- Debugging an app running inside a container

- Writing and running unit and integration tests
- Instrumenting a containerized app

Objective

Once you have read through this chapter, you will be able to enable hot reload for your application and use VS Code to debug your Java, .NET, NodeJS, or Python application running inside a container line by line. You will learn to run unit and integration tests for your application inside a container and instrument your application and thus prepare it to run in production.

Without further ado, let us dive into the first topic.

Enabling hot reload in a container

"Hot reload" (or "hot restart" or "hot code swapping") is a feature that allows developers to update the code of a running application without having to stop and restart the entire application. This feature is particularly useful for Web applications, where a full restart can cause disruption for users or lead to a loss of data.

For developers, it is very important to get into what they call "the flow". This is a state of mind where ideas on how to solve a problem just seemingly pour out of your head like a river. You are "in the zone". If a developer achieves this state of mind, he/she usually is very productive and gets things done in no time. But there are always lurking some factors in the dark that may pull out the developer from that ideal state. It is someone interrupting with a question, or a meeting has to be attended, and more. Other factors that can be disruptive are that after each minor code change, one has to interrupt the running application that one was debugging and recompile the code, restart the application, and navigate to the point where one did some changes. What if we could just automate all this and have it done for us without us even having to think about it?

For traditional applications such as Node JS, .NET, Java, or Python application, developers have long used special tools that monitor your code and, in case something has changed, automatically stop the running application, compile the new version, and restart the application. Then came containers, and developers seemingly got stuck again in the old habit of manually stopping, compiling, and restarting the application. But not for long. Quickly clever developers found a way to have the same flow when working with containers as they did before using containers. Let us see how that is possible.

- Writing and running unit and integration tests
- Instrumenting a containerized app

Objective

Once you have read through this chapter, you will be able to enable hot reload for your application and use VS Code to debug your Java, .NET, NodeJS, or Python application running inside a container line by line. You will learn to run unit and integration tests for your application inside a container and instrument your application and thus prepare it to run in production.

Without further ado, let us dive into the first topic.

Enabling hot reload in a container

"Hot reload" (or "hot restart" or "hot code swapping") is a feature that allows developers to update the code of a running application without having to stop and restart the entire application. This feature is particularly useful for Web applications, where a full restart can cause disruption for users or lead to a loss of data.

For developers, it is very important to get into what they call "the flow". This is a state of mind where ideas on how to solve a problem just seemingly pour out of your head like a river. You are "in the zone". If a developer achieves this state of mind, he/she usually is very productive and gets things done in no time. But there are always lurking some factors in the dark that may pull out the developer from that ideal state. It is someone interrupting with a question, or a meeting has to be attended, and more. Other factors that can be disruptive are that after each minor code change, one has to interrupt the running application that one was debugging and recompile the code, restart the application, and navigate to the point where one did some changes. What if we could just automate all this and have it done for us without us even having to think about it?

For traditional applications such as Node JS, .NET, Java, or Python application, developers have long used special tools that monitor your code and, in case something has changed, automatically stop the running application, compile the new version, and restart the application. Then came containers, and developers seemingly got stuck again in the old habit of manually stopping, compiling, and restarting the application. But not for long. Quickly clever developers found a way to have the same flow when working with containers as they did before using containers. Let us see how that is possible.

Hot reloading a .NET application

We will now create a simple .NET application which we will then use to explain the principle of hot reloading:

1. Open a new terminal and navigate to your source code folder for this chapter, e.g.:

   ```
   cd ~/src/ch08
   ```

2. Create a new .NET application of type WebApi and call it `hot-reload-net`:

   ```
   dotnet new webapi --name hot-reload-net
   ```

3. Navigate to this newly created sub-folder:

   ```
   cd hot-reload-net
   ```

4. Open VS Code for this folder:

   ```
   code .
   ```

5. Open a terminal inside VS Code and then start the application with the following command from within that terminal:

   ```
   dotnet run
   ```

6. Open a new browser tab and navigate to the URL

   https://localhost:5001/weatherforecast

   The output should be a JSON-formatted array of weather forecast items.

7. Back in VS Code, let us edit some code. Extend the range from 1 to 5 to 1 to 10 on line 30 in the class `WeatherForecastController.cs`, which is located in the `Controllers` folder. Save the file after you have changed it.

8. In the terminal, stop the running application with *CTRL-c* and compile and run it again with the following:

   ```
   dotnet run
   ```

   Then refresh the browser tab that shows the weather forecast results and verify that now each time you refresh 10 weather items are returned instead of 5 as before.

9. Step 8 is quite annoying, and developers will find it to add friction to their flow over time. How can we mitigate that? Well, .NET got you covered. Instead of running the application, we can watch and run it. In the terminal, execute the following:

   ```
   dotnet watch run
   ```

Hot reloading a .NET application

We will now create a simple .NET application which we will then use to explain the principle of hot reloading:

1. Open a new terminal and navigate to your source code folder for this chapter, e.g.:

   ```
   cd ~/src/ch08
   ```

2. Create a new .NET application of type WebApi and call it `hot-reload-net`:

   ```
   dotnet new webapi --name hot-reload-net
   ```

3. Navigate to this newly created sub-folder:

   ```
   cd hot-reload-net
   ```

4. Open VS Code for this folder:

   ```
   code .
   ```

5. Open a terminal inside VS Code and then start the application with the following command from within that terminal:

   ```
   dotnet run
   ```

6. Open a new browser tab and navigate to the URL

   https://localhost:5001/weatherforecast

   The output should be a JSON-formatted array of weather forecast items.

7. Back in VS Code, let us edit some code. Extend the range from 1 to 5 to 1 to 10 on line 30 in the class `WeatherForecastController.cs`, which is located in the `Controllers` folder. Save the file after you have changed it.

8. In the terminal, stop the running application with *CTRL-c* and compile and run it again with the following:

   ```
   dotnet run
   ```

   Then refresh the browser tab that shows the weather forecast results and verify that now each time you refresh 10 weather items are returned instead of 5 as before.

9. Step 8 is quite annoying, and developers will find it to add friction to their flow over time. How can we mitigate that? Well, .NET got you covered. Instead of running the application, we can watch and run it. In the terminal, execute the following:

   ```
   dotnet watch run
   ```

Notice the `watch` in the preceding command. This little change is a life changer.

10. Change the range on line 30 back to 1 to 5 and save the file. Observe in the output of the terminal window that dotnet has discovered a file change and automatically stops, rebuilds, and restarts the application. When you hit refresh in the browser, the new range value (1–5) is effective.

Hot reloading a NodeJS application

We will now create a simple Node JS application which we will use once again to explain the principle:

1. Open a new terminal and navigate to your source code folder for this chapter, e.g.:

   ```
   cd ~/src/ch08
   ```

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

   ```
   cd hot-reload-node && cd hot-reload-node
   ```

3. Initialize the application using the following:

   ```
   npm init
   ```

   and keep the default values suggested.

4. Open VS Code for this folder:

   ```
   code .
   ```

5. Add a file called `index.js` to the project and add this content to it:

```
index.js
index.js > server > http.createServer() callback
1   const http = require('http');
2
3   const hostname = '127.0.0.1';
4   const port = 3000;
5
6   const server = http.createServer((req, res) => {
7     res.statusCode = 200;
8     res.setHeader('Content-Type', 'text/plain');
9     res.end('Hello World');
10  });
11
12  server.listen(port, hostname, () => {
13    console.log(`Server running at http://${hostname}:${port}/`);
14  });
```

Figure 8.1: A simple Node JS application

Notice the `watch` in the preceding command. This little change is a life changer.

10. Change the range on line 30 back to 1 to 5 and save the file. Observe in the output of the terminal window that dotnet has discovered a file change and automatically stops, rebuilds, and restarts the application. When you hit refresh in the browser, the new range value (1–5) is effective.

Hot reloading a NodeJS application

We will now create a simple Node JS application which we will use once again to explain the principle:

1. Open a new terminal and navigate to your source code folder for this chapter, e.g.:

   ```
   cd ~/src/ch08
   ```

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

   ```
   cd hot-reload-node && cd hot-reload-node
   ```

3. Initialize the application using the following:

   ```
   npm init
   ```

   and keep the default values suggested.

4. Open VS Code for this folder:

   ```
   code .
   ```

5. Add a file called `index.js` to the project and add this content to it:

```
index.js
index.js > server > http.createServer() callback
1   const http = require('http');
2
3   const hostname = '127.0.0.1';
4   const port = 3000;
5
6   const server = http.createServer((req, res) => {
7     res.statusCode = 200;
8     res.setHeader('Content-Type', 'text/plain');
9     res.end('Hello World');
10  });
11
12  server.listen(port, hostname, () => {
13    console.log(`Server running at http://${hostname}:${port}/`);
14  });
```

Figure 8.1: A simple Node JS application

6. Open a terminal window inside VS Code and execute the following command in that terminal to run the application:

   ```
   node index.js
   ```

7. Open a new browser tab and navigate to **`http://localhost:3000`** and observe that the message **`Hello World`** is output on the page.

8. Change a value in the index.js file, e.g., change Line 9 to:

   ```
   res.end('Hello docker Job Seeker');
   ```

   and then save the modified file.

9. In the terminal, use CTRL-c to stop the running application and then execute the **`node index.js`** command again to restart the application.

10. Refresh the browser tab to make sure that the message has indeed been changed to **`Hello docker Job Seeker`**.

11. Having to repeat Steps 9 and 10 each time one changes a bit of code is tiresome, to say the least, over time. But people have mitigated this by using a tool such as **`nodemon`** to improve the situation. Please install **`nodemon`** globally with the following command:

    ```
    npm install -g nodemon
    ```

12. Use **`nodemon`** to run your application. In the terminal, execute:

    ```
    nodemon
    ```

    you should see something like this:

    ```
    gabriel@Solaris hot-reload-node % nodemon
    [nodemon] 2.0.12
    [nodemon] to restart at any time, enter `rs`
    [nodemon] watching path(s): *.*
    [nodemon] watching extensions: js,mjs,json
    [nodemon] starting `node index.js`
    Server running at http://0.0.0.0:3000/
    ```

    Figure 8.2: Running a Node JS application with nodemon

    Notice in the output preceding the third and fourth lines saying that files are watched.

13. Change something in the **`index.js`** file; for example, revert the message back to "Hello World" on Line 9, and then save the modified file. Notice in the terminal output that **`nodemon`** automatically stops and restarts the application for you.

14. Refresh the browser to verify that the new value is indeed used.

6. Open a terminal window inside VS Code and execute the following command in that terminal to run the application:

   ```
   node index.js
   ```

7. Open a new browser tab and navigate to **`http://localhost:3000`** and observe that the message **`Hello World`** is output on the page.

8. Change a value in the index.js file, e.g., change Line 9 to:

   ```
   res.end('Hello docker Job Seeker');
   ```

   and then save the modified file.

9. In the terminal, use CTRL-c to stop the running application and then execute the **`node index.js`** command again to restart the application.

10. Refresh the browser tab to make sure that the message has indeed been changed to **`Hello docker Job Seeker`**.

11. Having to repeat Steps 9 and 10 each time one changes a bit of code is tiresome, to say the least, over time. But people have mitigated this by using a tool such as **`nodemon`** to improve the situation. Please install **`nodemon`** globally with the following command:

    ```
    npm install -g nodemon
    ```

12. Use **`nodemon`** to run your application. In the terminal, execute:

    ```
    nodemon
    ```

    you should see something like this:

    ```
    gabriel@Solaris hot-reload-node % nodemon
    [nodemon] 2.0.12
    [nodemon] to restart at any time, enter `rs`
    [nodemon] watching path(s): *.*
    [nodemon] watching extensions: js,mjs,json
    [nodemon] starting `node index.js`
    Server running at http://0.0.0.0:3000/
    ```

    Figure 8.2: Running a Node JS application with nodemon

    Notice in the output preceding the third and fourth lines saying that files are watched.

13. Change something in the **`index.js`** file; for example, revert the message back to "Hello World" on Line 9, and then save the modified file. Notice in the terminal output that **`nodemon`** automatically stops and restarts the application for you.

14. Refresh the browser to verify that the new value is indeed used.

Hot reloading a ReactJS application

Today's modern Web applications are mostly written using one of the popular **single-page application (SPA)** frameworks such as React, Angular, or Vue. In this section, we are going to learn how to hot reload a React JS application that has been containerized. To do so, please proceed as follows:

1. Open a new terminal and navigate to your source code folder for this chapter, e.g.:

   ```
   cd ~/src/ch08
   ```

2. Initialize a new React application called `hot-reload-react` using the following command:

   ```
   npx create-react-app hot-reload-react
   ```

   when asked if you want to install the `create-react-app` module, answer with yes. As soon as this package is installed, the React application will be bootstrapped. This may take a moment since many packages have to be downloaded and installed. Thus, please be patient.

3. Navigate to the `hot-reload-react` subfolder that was created in the previous step:

   ```
   cd hot-reload-react
   ```

4. Open VS Code for this folder:

   ```
   code .
   ```

5. Open a terminal within VS Code, and in this terminal, start the React development server with the following:

   ```
   npm start
   ```

   Automatically a new browser window will open at `http://localhost:3000`, and the default React JS application will be displayed in the browser:

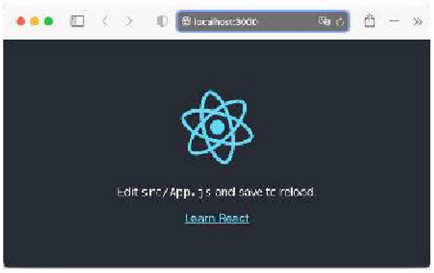


Figure 8.3: Default React JS application displayed in the browser

Hot reloading a ReactJS application

Today's modern Web applications are mostly written using one of the popular **single-page application** (SPA) frameworks such as React, Angular, or Vue. In this section, we are going to learn how to hot reload a React JS application that has been containerized. To do so, please proceed as follows:

1. Open a new terminal and navigate to your source code folder for this chapter, e.g.:

   ```
   cd ~/src/ch08
   ```

2. Initialize a new React application called `hot-reload-react` using the following command:

   ```
   npx create-react-app hot-reload-react
   ```

   when asked if you want to install the `create-react-app` module, answer with yes. As soon as this package is installed, the React application will be bootstrapped. This may take a moment since many packages have to be downloaded and installed. Thus, please be patient.

3. Navigate to the `hot-reload-react` subfolder that was created in the previous step:

   ```
   cd hot-reload-react
   ```

4. Open VS Code for this folder:

   ```
   code .
   ```

5. Open a terminal within VS Code, and in this terminal, start the React development server with the following:

   ```
   npm start
   ```

   Automatically a new browser window will open at `http://localhost:3000`, and the default React JS application will be displayed in the browser:



Figure 8.3: Default React JS application displayed in the browser

6. In VS Code, locate the file **app.js** in the subfolder **src**. Change its content to look like the following:

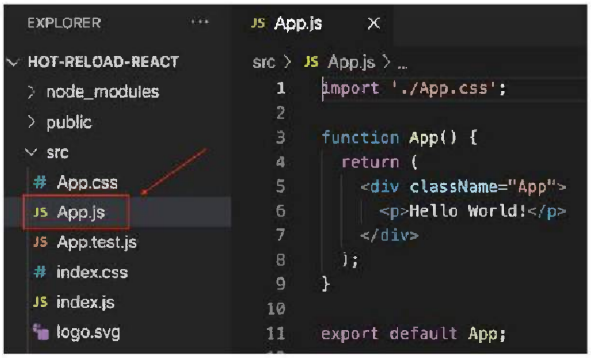


Figure 8.4: Simplified app.js of the React JS application

7. Save the file and notice in the terminal output that the application is automatically rebuilt and restarted. Also, notice that the content of the browser page updated automatically as well. The React community has done a tremendous job and already enabled hot reload out of the box!

8. Do more changes to the code to see hot reloading in action.

9. When done, stop the application by pressing CTRL-c in the terminal window.

Now, we want to Dockerize the React application as follows:

1. Start by adding a new file called **.Dockerignore** file to the root of the project and add the following content:

```
.vscode
node_modules
```

2. Add a new file called **Dockerfile** also to the root of the project and add the following content:

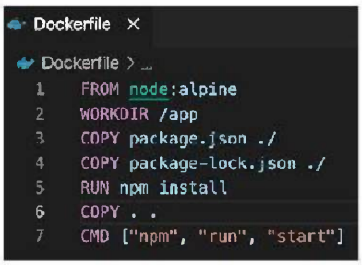


Figure 8.5: Dockerfile for the React JS application

6. In VS Code, locate the file **app.js** in the subfolder **src**. Change its content to look like the following:

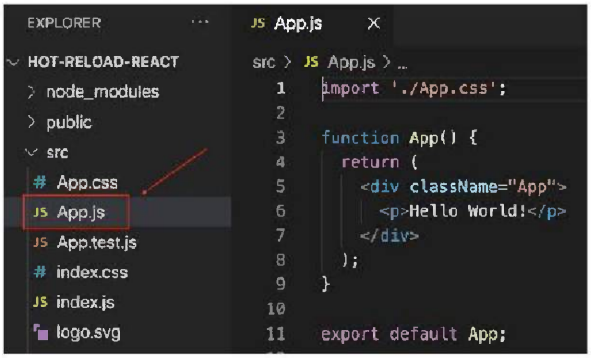


Figure 8.4: Simplified app.js of the React JS application

7. Save the file and notice in the terminal output that the application is automatically rebuilt and restarted. Also, notice that the content of the browser page updated automatically as well. The React community has done a tremendous job and already enabled hot reload out of the box!

8. Do more changes to the code to see hot reloading in action.

9. When done, stop the application by pressing CTRL-c in the terminal window.

Now, we want to Dockerize the React application as follows:

1. Start by adding a new file called **.Dockerignore** file to the root of the project and add the following content:

```
.vscode

node_modules
```

2. Add a new file called **Dockerfile** also to the root of the project and add the following content:

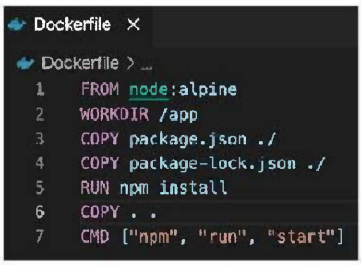


Figure 8.5: Dockerfile for the React JS application

We are using the `node:alpine` image as the base for our React application (Line 1). Then, we copy the `package.json` and `package-lock.json` into the container (Lines 3 and 4) to prepare us for the `npm install` command (Line 5) that we want to execute. Finally, in Line 6, we copy all the sources into the working directory (`/app`) inside the container. In Line 7, we define the start command that Docker shall use whenever an instance of this image is run as a container. Remember that this command (`npm run start`) will start the development server of our React application. This Dockerfile has been optimized such as the `npm install` on Line 5 is executed before the whole source code is added to the container.

3. In the terminal window, execute the following command to generate a new Docker image called hot-reload-react from the project using the preceding Dockerfile:

```
docker image build -t hot-reload-react .
```

4. Run an instance of this image and map container port `3000` to host port `4000` using the following command:

```
docker container run -p 4000:3000 hot-reload-react
```

You should see something like the following:

```
gabriel@Solaris hot-reload-react % docker container run -p 4000:3000 hot-reload-react

> hot-reload-react@0.1.0 start
> react-scripts start

ℹ ｢wds｣: Project is running at http://172.17.0.3/
ℹ ｢wds｣: webpack output is served from
ℹ ｢wds｣: Content not from webpack is served from /app/public
ℹ ｢wds｣: 404s will fallback to /
Starting the development server...

Compiled successfully!

You can now view hot-reload-react in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://172.17.0.3:3000

Note that the development build is not optimized.
To create a production build, use npm run build.
```

Figure 8.6: Starting the React application in a container

5. Open a new browser tab and navigate to `http://localhost:4000`. You should now see the welcome message displayed in the browser window. Note, you need to use port `4000` and not port `3000` since the latter is the container port and the former is the host port to which the container port has

We are using the `node:alpine` image as the base for our React application (Line 1). Then, we copy the `package.json` and `package-lock.json` into the container (Lines 3 and 4) to prepare us for the `npm install` command (Line 5) that we want to execute. Finally, in Line 6, we copy all the sources into the working directory (`/app`) inside the container. In Line 7, we define the start command that Docker shall use whenever an instance of this image is run as a container. Remember that this command (`npm run start`) will start the development server of our React application. This Dockerfile has been optimized such as the `npm install` on Line 5 is executed before the whole source code is added to the container.

3. In the terminal window, execute the following command to generate a new Docker image called hot-reload-react from the project using the preceding Dockerfile:

```
docker image build -t hot-reload-react .
```

4. Run an instance of this image and map container port `3000` to host port `4000` using the following command:

```
docker container run -p 4000:3000 hot-reload-react
```

You should see something like the following:

```
gabriel@Solaris hot-reload-react % docker container run -p 4000:3000 hot-reload-react

> hot-reload-react@0.1.0 start
> react-scripts start

ℹ ｢wds｣: Project is running at http://172.17.0.3/
ℹ ｢wds｣: webpack output is served from
ℹ ｢wds｣: Content not from webpack is served from /app/public
ℹ ｢wds｣: 404s will fallback to /
Starting the development server...

Compiled successfully!

You can now view hot-reload-react in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://172.17.0.3:3000

Note that the development build is not optimized.
To create a production build, use npm run build.
```

Figure 8.6: Starting the React application in a container

5. Open a new browser tab and navigate to `http://localhost:4000`. You should now see the welcome message displayed in the browser window. Note, you need to use port `4000` and not port `3000` since the latter is the container port and the former is the host port to which the container port has

been mapped with the parameter `-p 4000:3000` in the `docker container run` command.

6. Now make a little change in the `App.js` file of the project—e.g., change the message on Line 6 to "Hello beautiful World!"—and then save the change.

   Bummer..., nothing happens. Neither is the development server of the React application restarted nor is the browser pointing to the application refreshed.

   But this is to be expected. The development server is running inside the container, and it uses and monitors the code that is also packed into the container. But you changed the code directly on your system, and the code inside the container remained unaffected. Luckily, we can change that.

7. Stop the running container by pressing *CTRL-c* in the terminal window where you started it.

8. Use the following command to start another instance of the container, this time using volume mapping to enable hot reloading:

   ```
   docker container run -p 4000:3000 \
       -v $PWD/src:/app/src \
       -v $PWD/public:/app/public \
       hot-reload-react
   ```

   With the preceding command, we are mapping the two folders `src` and `public`, into the container, replacing their respective content with the content of our host folders. This enables dynamic updates of the code inside the container without having to rebuild and restart the container.

9. Now, once again, change something in the `App.js` file, e.g., change the message to something like "Hello Docker Job Seeker!" and save the changes. Observe in the terminal output that the development server running inside the container is picking up the change and compiling and restarting the app. You should see the following output:

   ```
   Compiling...
   Compiled successfully!
   ```

   Also, notice that the browser window is refreshing automatically too, and displaying your change.

Now that we have significantly decreased the friction in the development process when using containers let us turn the notch up a bit and learn how to debug an application step-by-step that is running inside a container.

been mapped with the parameter `-p 4000:3000` in the `docker container run` command.

6. Now make a little change in the `App.js` file of the project—e.g., change the message on Line 6 to "Hello beautiful World!"—and then save the change.

   Bummer..., nothing happens. Neither is the development server of the React application restarted nor is the browser pointing to the application refreshed.

   But this is to be expected. The development server is running inside the container, and it uses and monitors the code that is also packed into the container. But you changed the code directly on your system, and the code inside the container remained unaffected. Luckily, we can change that.

7. Stop the running container by pressing *CTRL-c* in the terminal window where you started it.

8. Use the following command to start another instance of the container, this time using volume mapping to enable hot reloading:

   ```
   docker container run -p 4000:3000 \
       -v $PWD/src:/app/src \
       -v $PWD/public:/app/public \
       hot-reload-react
   ```

   With the preceding command, we are mapping the two folders `src` and `public`, into the container, replacing their respective content with the content of our host folders. This enables dynamic updates of the code inside the container without having to rebuild and restart the container.

9. Now, once again, change something in the `App.js` file, e.g., change the message to something like "Hello Docker Job Seeker!" and save the changes. Observe in the terminal output that the development server running inside the container is picking up the change and compiling and restarting the app. You should see the following output:

   ```
   Compiling...
   Compiled successfully!
   ```

   Also, notice that the browser window is refreshing automatically too, and displaying your change.

Now that we have significantly decreased the friction in the development process when using containers let us turn the notch up a bit and learn how to debug an application step-by-step that is running inside a container.

Debugging an app running inside a container

We will be using Visual Studio Code to demonstrate how to debug applications that run inside a container. Most other IDEs, such as IntelliJ, Eclipse, or Visual Studio, support this as well, but you will want to read the respective documentation to find out exactly how this is working.

Let us demonstrate how you can debug a Java application running inside a container first.

Debugging a Java application

As indicated previously, we will be using Visual Studio Code to demonstrate this. This code editor has become the ubiquitous editor on all popular operating systems, such as Windows 10 or 11, MacOS, and Linux. Let us start with a simple "Hello World" type application to demonstrate the principle.

1. Open a new terminal and navigate to your source code folder for this chapter, for example:

   ```
   cd ~/src/ch08
   ```

2. Create a new sub-folder and call it **debug-java** and navigate to it:

   ```
   mkdir debug-java && cd debug-java
   ```

3. Open VS Code for this folder:

   ```
   code .
   ```

4. Double-check that you have the necessary Java plugins installed. If not, proceed as follows:
   a. Open the extensions side panel with ⇧-⌘-x on a Mac (or *SHIFT-CTRL-x* on Windows).
   b. Look for the "Extension Pack for Java" package and install it.

Debugging an app running inside a container

We will be using Visual Studio Code to demonstrate how to debug applications that run inside a container. Most other IDEs, such as IntelliJ, Eclipse, or Visual Studio, support this as well, but you will want to read the respective documentation to find out exactly how this is working.

Let us demonstrate how you can debug a Java application running inside a container first.

Debugging a Java application

As indicated previously, we will be using Visual Studio Code to demonstrate this. This code editor has become the ubiquitous editor on all popular operating systems, such as Windows 10 or 11, MacOS, and Linux. Let us start with a simple "Hello World" type application to demonstrate the principle.

1. Open a new terminal and navigate to your source code folder for this chapter, for example:

   ```
   cd ~/src/ch08
   ```

2. Create a new sub-folder and call it **debug-java** and navigate to it:

   ```
   mkdir debug-java && cd debug-java
   ```

3. Open VS Code for this folder:

   ```
   code .
   ```

4. Double-check that you have the necessary Java plugins installed. If not, proceed as follows:
   a. Open the extensions side panel with ⇧-⌘-x on a Mac (or *SHIFT-CTRL-x* on Windows).
   b. Look for the "Extension Pack for Java" package and install it.

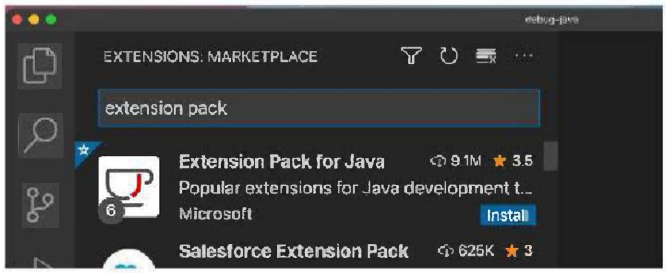


Figure 8.7: Installing extension pack for Java in Visual Studio Code

This extension pack contains extensions such as Language Support for Java by RedHat, Debugger for Java, Test Runner for Java, and more.

c. Restart VS Code if requested.

5. Press ⇧-⌘-e (*SHIFT-CTRL-e*) to display the Project Explorer sidebar in VS Code.

6. Press ⇧-⌘-p (*SHIFT-CTRL-p*) to display the command bar and enter "`Maven:`". From the list of suggested commands, select "**Maven: Create Maven Project**."

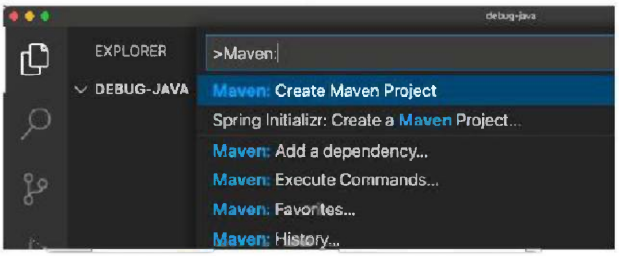


Figure 8.8: Creating a new Maven project via the command bar

7. From the list of archetypes, select the archetype **`maven-archetype-quickstart`**, version `1.4`, and leave the default group ID **`example.com`** and artifact ID **`debug-java`** to have VS Code create a simple Maven-based Java application.

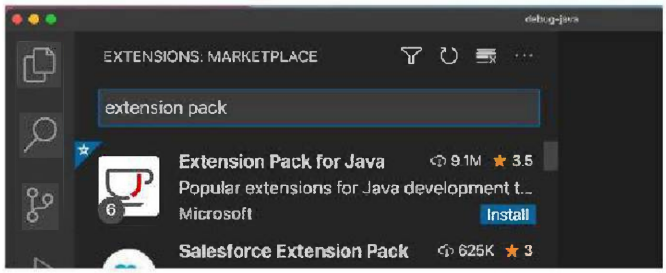


Figure 8.7: Installing extension pack for Java in Visual Studio Code

This extension pack contains extensions such as Language Support for Java by RedHat, Debugger for Java, Test Runner for Java, and more.

c. Restart VS Code if requested.

5. Press ⬆-⌘-e (*SHIFT-CTRL-e*) to display the Project Explorer sidebar in VS Code.

6. Press ⬆-⌘-p (*SHIFT-CTRL-p*) to display the command bar and enter "`Maven:`". From the list of suggested commands, select "**Maven: Create Maven Project**."

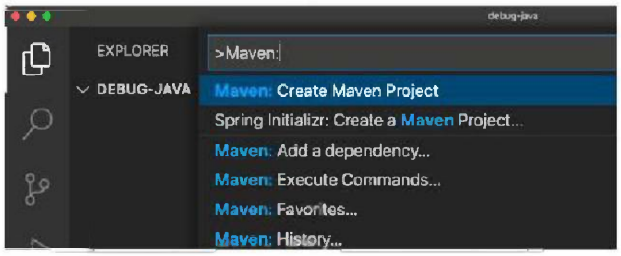


Figure 8.8: Creating a new Maven project via the command bar

7. From the list of archetypes, select the archetype `maven-archetype-quickstart`, version `1.4`, and leave the default group ID `example.com` and artifact ID `debug-java` to have VS Code create a simple Maven-based Java application.

8. Explore the solution created and open the `App.java` file. Note that in the sample shown as follows, we have removed the comments from the code for more clarity. Add a break point on `println` statement and then click on the Debug link right above the `main` method.

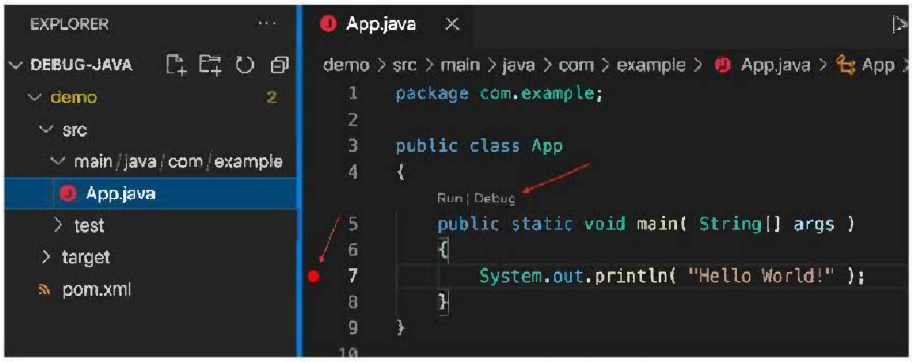


Figure 8.9: Debugging a Java application in VS Code

Notice how the application starts and then is stopped on Line 7 at the breakpoint. You can now use the debug toolbar to step through your code.

So far, so good. We are able to debug a native Java application from within VS Code. But now the interesting question is: Can I do the same if the application does not run natively but rather inside a container?

To start, let us try to attach VS Code to the Java application when it is running natively. We will then use what we have learned and use it when running the Java application inside a container. Proceed as follows:

1. First, we replace the content of the main method in the App class as follows:

```
public static void main( String[] args )
{
    System.out.println("Starting debugging demo application");
    int i = 1;
    while(true){
        System.out.println("Round " + i);
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        i++;
    }
}
```

Figure 8.10: Main method of Java debug demo app

8. Explore the solution created and open the `App.java` file. Note that in the sample shown as follows, we have removed the comments from the code for more clarity. Add a break point on `println` statement and then click on the Debug link right above the `main` method.

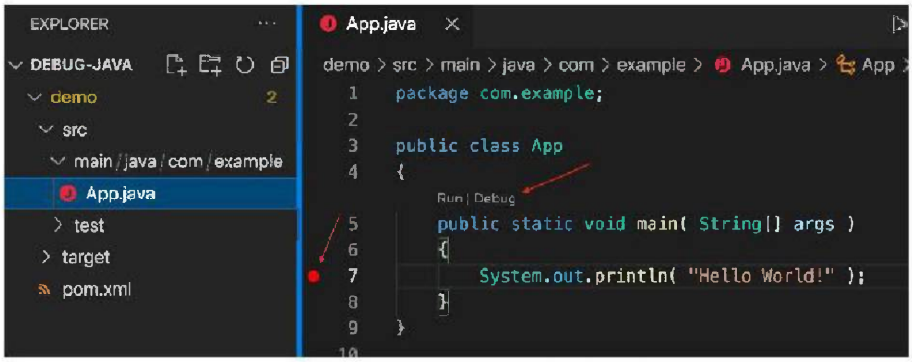


Figure 8.9: Debugging a Java application in VS Code

Notice how the application starts and then is stopped on Line 7 at the breakpoint. You can now use the debug toolbar to step through your code.

So far, so good. We are able to debug a native Java application from within VS Code. But now the interesting question is: Can I do the same if the application does not run natively but rather inside a container?

To start, let us try to attach VS Code to the Java application when it is running natively. We will then use what we have learned and use it when running the Java application inside a container. Proceed as follows:

1. First, we replace the content of the main method in the App class as follows:

```
public static void main( String[] args )
{
    System.out.println("Starting debugging demo application");
    int i = 1;
    while(true){
        System.out.println("Round " + i);
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        i++;
    }
}
```

Figure 8.10: Main method of Java debug demo app

This keeps the application running forever and gives us plenty of time to attach a debugger and play with it.

2. On the command line, execute the Maven package command:

   ```
   mvn package
   ```

   This will create a jar file called **debug-java-1.0-SNAPSHOT.jar** in the target folder.

3. Define the **JAVA_TOOLS_OPTIONS** environment variable such as that when running the Java application, it is instrumented for debugging:

   ```
   export JAVA_TOOL_OPTIONS=-agentlib:jdwp=transport=dt_
   socket,address=8000,server=y,suspend=n
   ```

   Notice that we are using port **8000** for the debugger to attach.

4. Run the application from the command line:

   ```
   java -cp target/debug-java-1.0-SNAPSHOT.jar com.example.App
   ```

   The output should look like the following:

```
Picked up JAVA_TOOL_OPTIONS: -agentlib:jdwp=transport=dt_socket,address=8000,serve
r=y,suspend=n
Listening for transport dt_socket at address: 8000
Starting debugging demo application
Round 1
Round 2
Round 3
```

Figure 8.11: Output of locally running Java application

The application runs and outputs every 5 seconds a line with "Round <X>" where <X> is an ever-increasing number. Notice the output where the debugger is telling us that it is listening at port 8000.

5. In VS Code, open the debug side panel and click on "**create launch.json file**":

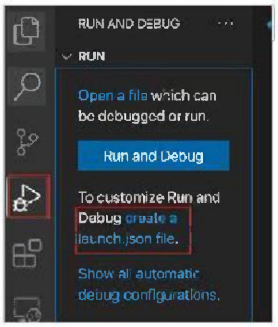


Figure 8.12: Create a launch.json file for attaching a debugger to the Java app

This keeps the application running forever and gives us plenty of time to attach a debugger and play with it.

2. On the command line, execute the Maven package command:

   ```
   mvn package
   ```

   This will create a jar file called **debug-java-1.0-SNAPSHOT.jar** in the target folder.

3. Define the **JAVA_TOOLS_OPTIONS** environment variable such as that when running the Java application, it is instrumented for debugging:

   ```
   export JAVA_TOOL_OPTIONS=-agentlib:jdwp=transport=dt_
   socket,address=8000,server=y,suspend=n
   ```

   Notice that we are using port **8000** for the debugger to attach.

4. Run the application from the command line:

   ```
   java -cp target/debug-java-1.0-SNAPSHOT.jar com.example.App
   ```

   The output should look like the following:

```
Picked up JAVA_TOOL_OPTIONS: -agentlib:jdwp=transport=dt_socket,address=8000,serve
r=y,suspend=n
Listening for transport dt_socket at address: 8000
Starting debugging demo application
Round 1
Round 2
Round 3
```

Figure 8.11: Output of locally running Java application

The application runs and outputs every 5 seconds a line with "Round <X>" where <X> is an ever-increasing number. Notice the output where the debugger is telling us that it is listening at port 8000.

5. In VS Code, open the debug side panel and click on "**create launch.json file**":

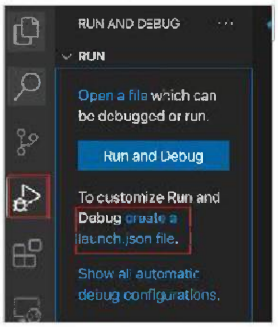


Figure 8.12: Create a launch.json file for attaching a debugger to the Java app

6. Add the following launch definition called "attach":

launch.json

```
.vscode > {} launch.json > ...
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "java",
            "name": "Attach",
            "request": "attach",
            "hostName": "localhost",
            "port": 8000
        }
    ]
}
```

Figure 8.13: Launch task for VS Code to attach to Java application

7. Put a breakpoint on Line 16 of the `main` method in the `App` class.
8. Now, hit *F5* to start debugging and notice how the debugger hits the breakpoint at the line.

Now, we are ready to transfer what we have learned by attaching the VS Code Java debugger to a natively running application to the situation when we run the application inside a container. For this, follow the following steps:

1. Add a new file called `dockerfile` to the root of the project and add this content:

Dockerfile

```
Dockerfile > ...
FROM openjdk:8
WORKDIR /app
ENV JAVA_TOOL_OPTIONS -agentlib:jdwp=transport=dt_socket,address=8000,server=y,suspend=n
COPY target/debug-java-1.0-SNAPSHOT.jar  debug-java.jar
ENTRYPOINT ["java", "-cp", "debug-java.jar", "com.example.App"]
```

Figure 8.14: Dockerfile for debugging a Java application

2. Make sure you have built the application with `mvn package` before you continue.
3. Create a Docker image using the preceding Dockerfile:

```
docker image build -t debug-java .
```

4. Finally, run a container using the just-built image:

```
docker container run -p 8000:8000 debug-java
```

6. Add the following launch definition called "attach":

launch.json

```
.vscode > {} launch.json > ...
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "java",
            "name": "Attach",
            "request": "attach",
            "hostName": "localhost",
            "port": 8000
        }
    ]
}
```

Figure 8.13: Launch task for VS Code to attach to Java application

7. Put a breakpoint on Line 16 of the `main` method in the `App` class.
8. Now, hit *F5* to start debugging and notice how the debugger hits the breakpoint at the line.

Now, we are ready to transfer what we have learned by attaching the VS Code Java debugger to a natively running application to the situation when we run the application inside a container. For this, follow the following steps:

1. Add a new file called `dockerfile` to the root of the project and add this content:

Dockerfile

```
Dockerfile > ...
FROM openjdk:8
WORKDIR /app
ENV JAVA_TOOL_OPTIONS -agentlib:jdwp=transport=dt_socket,address=8000,server=y,suspend=n
COPY target/debug-java-1.0-SNAPSHOT.jar  debug-java.jar
ENTRYPOINT ["java", "-cp", "debug-java.jar", "com.example.App"]
```

Figure 8.14: Dockerfile for debugging a Java application

2. Make sure you have built the application with `mvn package` before you continue.
3. Create a Docker image using the preceding Dockerfile:

```
docker image build -t debug-java .
```

4. Finally, run a container using the just-built image:

```
docker container run -p 8000:8000 debug-java
```

Note how we map port 8000 to the equivalent host port, so the debugger can attach to the Java application running inside the container. The application inside the container starts running and produces the same output to the console as we saw when running the application natively.

5. Back in VS Code, hit *F5* to start debugging the application. Notice how the debugger stops at the breakpoint at Line 16:

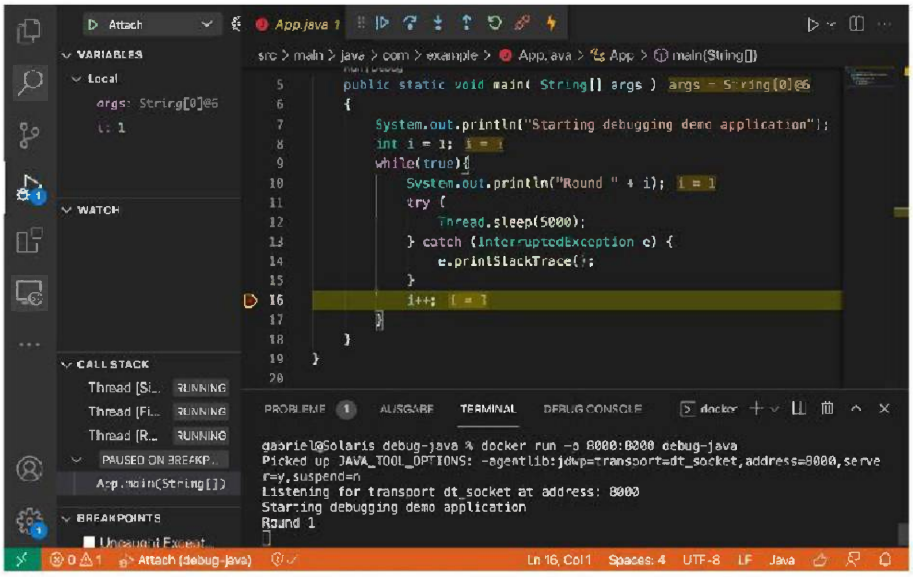


Figure 8.15: Debugger on a breakpoint in Java app running inside a container

Now that you know how to debug a Java application line by line, even when it is running inside a container, let us see how to do the same with a .NET C# application.

Debugging a .NET application

Now, let us try the same with a simple .NET application:

1. Open a new terminal and navigate to your source code folder for this chapter, for example.:

   ```
   cd ~/src/ch08
   ```

2. Create a new .NET console application using the following command:

   ```
   dotnet new console --name debug-net
   ```

Note how we map port 8000 to the equivalent host port, so the debugger can attach to the Java application running inside the container. The application inside the container starts running and produces the same output to the console as we saw when running the application natively.

5. Back in VS Code, hit *F5* to start debugging the application. Notice how the debugger stops at the breakpoint at Line 16:

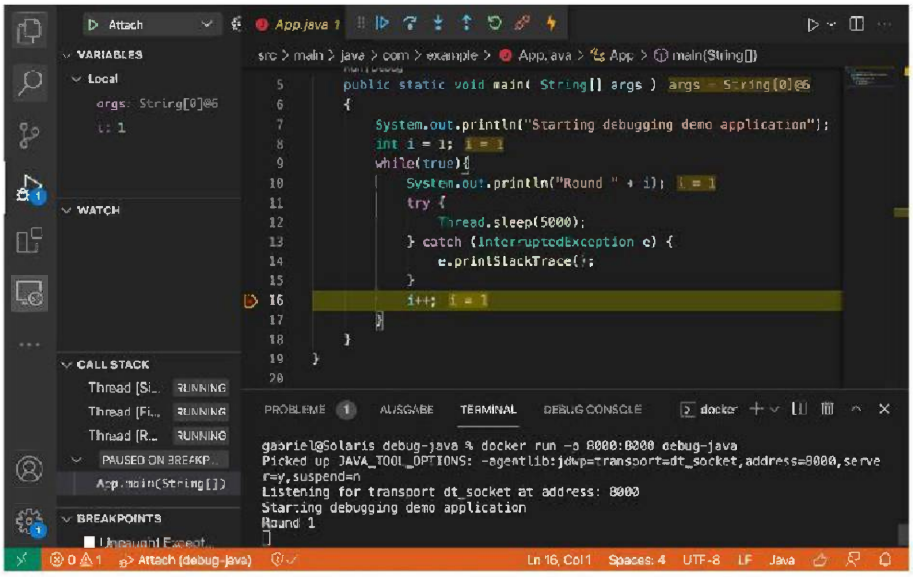


Figure 8.15: Debugger on a breakpoint in Java app running inside a container

Now that you know how to debug a Java application line by line, even when it is running inside a container, let us see how to do the same with a .NET C# application.

Debugging a .NET application

Now, let us try the same with a simple .NET application:

1. Open a new terminal and navigate to your source code folder for this chapter, for example.:

   ```
   cd ~/src/ch08
   ```

2. Create a new .NET console application using the following command:

   ```
   dotnet new console --name debug-net
   ```

3. Navigate to the subfolder **debug-net** created by the previous command:

   ```
   cd debug-net
   ```

4. Open VS Code for this folder:

   ```
   code .
   ```

5. Double-check that you have the necessary .NET plugins installed. If not, proceed as follows:

   a. Open the extensions side panel with ⇧-⌘-x on a Mac (or *SHIFT-CTRL-x* on Windows).

   b. Look for the "C# for Visual Studio Code" plugin from Microsoft and install it.

   c. Restart VS Code if requested.

6. Explore the project and open the file **Program.cs**. Set a breakpoint on Line 9 (the **Console.WriteLine** statement)

7. Hit *F5* to start debugging the application and notice that the execution will stop at the breakpoint:

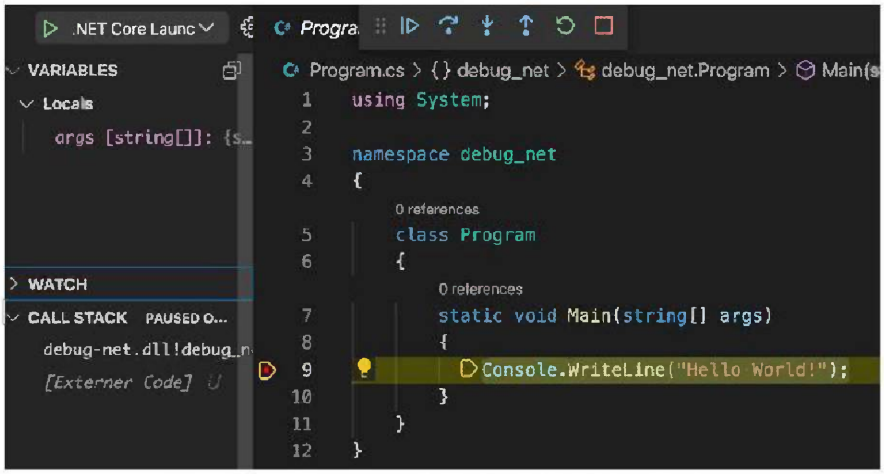


Figure 8.16: Debugger stopped at break point in .NET application

You can use the debug toolbar to now interactively step through the code.

Once again, this was the easy part. We can now debug a .NET application that runs natively on our system from withing VS Code. But now let us do the same when the application runs inside a container:

3. Navigate to the subfolder **debug-net** created by the previous command:

   ```
   cd debug-net
   ```

4. Open VS Code for this folder:

   ```
   code .
   ```

5. Double-check that you have the necessary .NET plugins installed. If not, proceed as follows:

   a. Open the extensions side panel with ⇧-⌘-x on a Mac (or *SHIFT-CTRL-x* on Windows).

   b. Look for the "C# for Visual Studio Code" plugin from Microsoft and install it.

   c. Restart VS Code if requested.

6. Explore the project and open the file **Program.cs**. Set a breakpoint on Line 9 (the **Console.WriteLine** statement)

7. Hit *F5* to start debugging the application and notice that the execution will stop at the breakpoint:

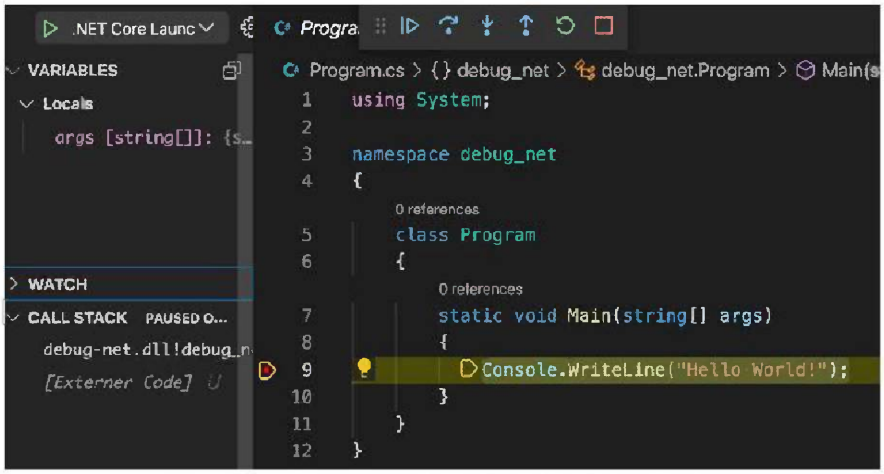


Figure 8.16: Debugger stopped at break point in .NET application

You can use the debug toolbar to now interactively step through the code.

Once again, this was the easy part. We can now debug a .NET application that runs natively on our system from withing VS Code. But now let us do the same when the application runs inside a container:

From within VS Code use the command "Docker: Add Docker Files to Workspace" command to add the necessary Dockerfile to our project. When asked, select .NET ".NET: Core Console" as the application platform and "Linux" as the operating system. For the moment, we do not need a *Docker-compose file; thus, answer that question with "No". We are now having a* Dockerfile *added to our project that looks similar to this:*

```
Dockerfile > RUN
1   FROM mcr.microsoft.com/dotnet/runtime:5.0-focal AS base
2   WORKDIR /app
3
4   RUN adduser -u 5678 --disabled-password --gecos "" appuser && chown -R appuser /app
5   USER appuser
6
7   FROM mcr.microsoft.com/dotnet/sdk:5.0-focal AS build
8   WORKDIR /src
9   COPY ["debug-net.csproj", "./"]
10  RUN dotnet restore "debug-net.csproj"
11  COPY . .
12  WORKDIR "/src/."
13  RUN dotnet build "debug-net.csproj" -c Release -o /app/build
14
15  FROM build AS publish
16  RUN dotnet publish "debug-net.csproj" -c Release -o /app/publish
17
18  FROM base AS final
19  WORKDIR /app
20  COPY --from=publish /app/publish .
21  ENTRYPOINT ["dotnet", "debug-net.dll"]
```

Figure 8.17: Dockerfile added to the .NET project

1. Next, we need to initialize our project for debugging with Docker. Hit ⇧-⌘-p (SHIFT-CTRL-p) to open the command palette. Search for the command "Docker: Initialize for Docker debugging" and execute it to prepare our project. Once again, select "**.NET: Core Console**" as the application platform and "**Linux**" as the operating system when asked.

2. Open the file **`launch.json`** inside the **`.vscode`** folder in your project. The previous command should have added a new launch task for Docker inside this file, looking similar to this:

```
{
    "name": "Docker .NET Core Launch",
    "type": "docker",
    "request": "launch",
    "preLaunchTask": "docker-run: debug",
    "netCore": {
        "appProject": "${workspaceFolder}/debug-net.csproj"
    }
}
```

Figure 8.18: Docker launch task for .NET application

From within VS Code use the command "Docker: Add Docker Files to Workspace" command to add the necessary Dockerfile to our project. When asked, select .NET ".NET: Core Console" as the application platform and "Linux" as the operating system. For the moment, we do not need a *Docker-compose file; thus, answer that question with "No". We are now having a* Dockerfile *added to our project that looks similar to this:*

```
Dockerfile > RUN
1   FROM mcr.microsoft.com/dotnet/runtime:5.0-focal AS base
2   WORKDIR /app
3
4   RUN adduser -u 5678 --disabled-password --gecos "" appuser && chown -R appuser /app
5   USER appuser
6
7   FROM mcr.microsoft.com/dotnet/sdk:5.0-focal AS build
8   WORKDIR /src
9   COPY ["debug-net.csproj", "./"]
10  RUN dotnet restore "debug-net.csproj"
11  COPY . .
12  WORKDIR "/src/."
13  RUN dotnet build "debug-net.csproj" -c Release -o /app/build
14
15  FROM build AS publish
16  RUN dotnet publish "debug-net.csproj" -c Release -o /app/publish
17
18  FROM base AS final
19  WORKDIR /app
20  COPY --from=publish /app/publish .
21  ENTRYPOINT ["dotnet", "debug-net.dll"]
```

Figure 8.17: Dockerfile added to the .NET project

1. Next, we need to initialize our project for debugging with Docker. Hit ⇧-⌘-p (SHIFT-CTRL-p) to open the command palette. Search for the command "Docker: Initialize for Docker debugging" and execute it to prepare our project. Once again, select "**.NET: Core Console**" as the application platform and "**Linux**" as the operating system when asked.

2. Open the file **`launch.json`** inside the **`.vscode`** folder in your project. The previous command should have added a new launch task for Docker inside this file, looking similar to this:

```
{
    "name": "Docker .NET Core Launch",
    "type": "docker",
    "request": "launch",
    "preLaunchTask": "docker-run: debug",
    "netCore": {
        "appProject": "${workspaceFolder}/debug-net.csproj"
    }
}
```

Figure 8.18: Docker launch task for .NET application

3. It did also add a few entries to the `tasks.json` file that also resides inside the `.vscode` folder. Please analyze them carefully.

Press ⬆-⌘-d (*SHIFT-CTRL-d*) to open the debug side panel in VS Code and then select the correct launch task from the drop-down menu. In our case, this is "**docker .NET Core Launch**":



Figure 8.19: Select the launch command in VS Code

Hit *F5* to start debugging. The execution should once again stop at the breakpoint. But this time, the code runs inside the container.

4. Open the dashboard of Docker Desktop to confirm that, indeed, there is a Docker container running with the application code that we are currently debugging. It is called `debugnet-dev`:

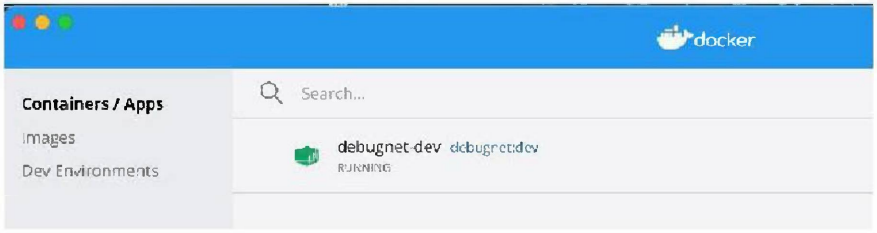


Figure 8.20: Dashboard of Docker Desktop showing the .NET debug container

5. On the dashboard, hover over the container and click on the first icon, which will open a shell inside of the running container. Execute the following command:

```
ls -al
```

you should see this:

3. It did also add a few entries to the **tasks.json** file that also resides inside the **.vscode** folder. Please analyze them carefully.

Press ⬆-⌘-d (*SHIFT-CTRL-d*) to open the debug side panel in VS Code and then select the correct launch task from the drop-down menu. In our case, this is "**docker .NET Core Launch**":



Figure 8.19: Select the launch command in VS Code

Hit *F5* to start debugging. The execution should once again stop at the breakpoint. But this time, the code runs inside the container.

4. Open the dashboard of Docker Desktop to confirm that, indeed, there is a Docker container running with the application code that we are currently debugging. It is called **debugnet-dev**:

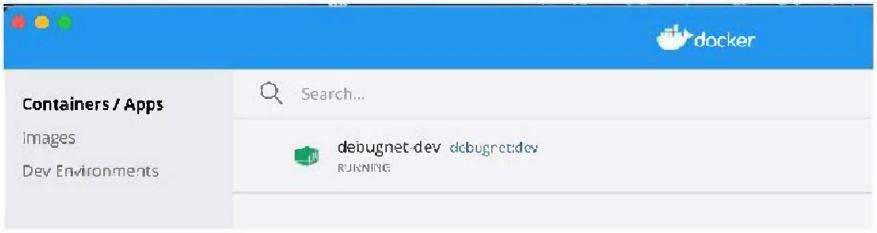


Figure 8.20: Dashboard of Docker Desktop showing the .NET debug container

5. On the dashboard, hover over the container and click on the first icon, which will open a shell inside of the running container. Execute the following command:

```
ls -al
```

you should see this:

```
$ ls -al
total 20
drwxr-xr-x 9 appuser appuser  288 Sep  5 15:08 .
drwxr-xr-x 1 root    root    4096 Sep  5 15:37 ..
-rw------- 1 appuser appuser  332 Sep  5 15:08 .dockerignore
drwxr-xr-x 4 appuser appuser  128 Sep  5 14:53 .vscode
-rw------- 1 appuser appuser  607 Sep  5 15:09 Dockerfile
-rw------- 1 appuser appuser  191 Sep  5 14:48 Program.cs
drwxr-xr-x 3 appuser appuser   96 Sep  5 14:52 bin
-rw------- 1 appuser appuser  217 Sep  5 14:48 debug-net.csproj
drwxr-xr-x 8 appuser appuser  256 Sep  5 14:52 obj
```

Figure 8.21: Listing the contents in the container used for debugging

These are the files and folders from our project, thus confirming that this is indeed the container containing our project.

Now, you can start extending this project and continue debugging it while it is running inside a Docker container.

Next, let us have a look at how this would work for a Node JS application.

Debugging a NodeJS application

Now, let us try the same with a simple Node JS application.

1. Open a new terminal and navigate to your source code folder for this chapter, for example: `cd ~/src/ch08`
2. Create a new sub-folder `debug-node` and navigate to it::

   ```
   mkdir debug-node && cd debug-node
   ```

3. Initialize a new Node JS application using the following command:

   ```
   npm init
   ```

   Accept all the defaults.

4. Open VS Code for this folder:

   ```
   code .
   ```

```
$ ls -al
total 20
drwxr-xr-x 9 appuser appuser  288 Sep  5 15:08 .
drwxr-xr-x 1 root    root    4096 Sep  5 15:37 ..
-rw------- 1 appuser appuser  332 Sep  5 15:08 .dockerignore
drwxr-xr-x 4 appuser appuser  128 Sep  5 14:53 .vscode
-rw------- 1 appuser appuser  607 Sep  5 15:09 Dockerfile
-rw------- 1 appuser appuser  191 Sep  5 14:48 Program.cs
drwxr-xr-x 3 appuser appuser   96 Sep  5 14:52 bin
-rw------- 1 appuser appuser  217 Sep  5 14:48 debug-net.csproj
drwxr-xr-x 8 appuser appuser  256 Sep  5 14:52 obj
```

Figure 8.21: Listing the contents in the container used for debugging

These are the files and folders from our project, thus confirming that this is indeed the container containing our project.

Now, you can start extending this project and continue debugging it while it is running inside a Docker container.

Next, let us have a look at how this would work for a Node JS application.

Debugging a NodeJS application

Now, let us try the same with a simple Node JS application.

1. Open a new terminal and navigate to your source code folder for this chapter, for example: `cd ~/src/ch08`
2. Create a new sub-folder `debug-node` and navigate to it::

   ```
   mkdir debug-node && cd debug-node
   ```

3. Initialize a new Node JS application using the following command:

   ```
   npm init
   ```

   Accept all the defaults.

4. Open VS Code for this folder:

   ```
   code .
   ```

5. Add a file called **`index.js`** to the project and add this content to it:

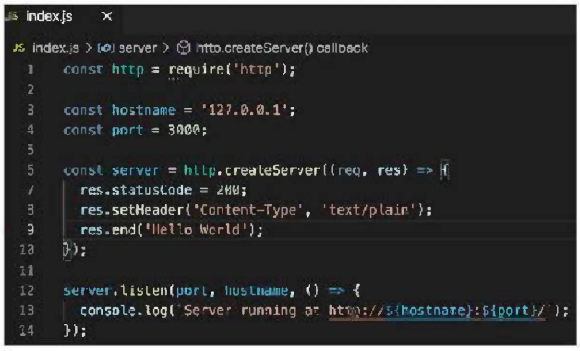


Figure 8.22: A simple Node JS application

6. Add a breakpoint on Line 9 and then hit *F5* to run the application (select "Node JS" as the environment when asked). Notice how the application is started.

7. Open a browser tab and navigate to **`http://localhost:3000`** and observe how the application execution stops on the breakpoint.

That was easy. By now, you are used to this procedure of creating a simple application and then start debugging it in VS Code. But now let us add support for debugging sessions when the code runs inside a container:

1. Press ⬆-⌘-p (*SHIFT-CTRL-p*) to open the command palette and locate and execute the command "**`docker: Add docker Files to Workspace`**". Select **Node.js** as the Application Platform and leave the default value of **`3000`** as the port. Select **No** when asked whether or not to add Docker-compose support.

   This command adds a Dockerfile to our project. It should look like the following:



Figure 8.23: Dockerfile added to Node JS project

5. Add a file called **`index.js`** to the project and add this content to it:

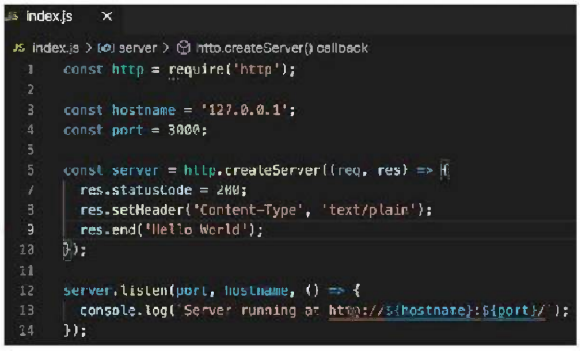


Figure 8.22: A simple Node JS application

6. Add a breakpoint on Line 9 and then hit *F5* to run the application (select "Node JS" as the environment when asked). Notice how the application is started.

7. Open a browser tab and navigate to **`http://localhost:3000`** and observe how the application execution stops on the breakpoint.

That was easy. By now, you are used to this procedure of creating a simple application and then start debugging it in VS Code. But now let us add support for debugging sessions when the code runs inside a container:

1. Press ⬆-⌘-p (*SHIFT-CTRL-p*) to open the command palette and locate and execute the command "**`docker: Add docker Files to Workspace`**". Select **Node.js** as the Application Platform and leave the default value of **`3000`** as the port. Select **No** when asked whether or not to add Docker-compose support.

   This command adds a Dockerfile to our project. It should look like the following:



Figure 8.23: Dockerfile added to Node JS project

2. Once again, press ⇧-⌘-p (*SHIFT-CTRL-p*) to open the command palette and locate and execute the command "**`docker: Initialize docker container`**". Select "**`Node JS`**" as the application platform when asked. This command will add/modify the **`launch.json`** file in the **`.vscode`** folder. It should now look like the following:

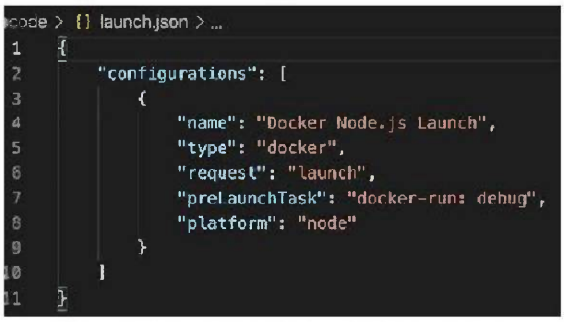


Figure 8.24: launch.json file for a Node JS application

3. You may also want to analyze the **`tasks.json`** file that was created by the preceding command.

4. To be able to work in a container, the Node JS Web server must listen at IP address 0.0.0.0 and not at the usual **`127.0.0.1`** (or **`localhost`**). Thus, before you proceed, please update the code in the file **`index.js`** and replace Line 3 with the following value:

```
const hostname = '0.0.0.0';
```

5. This following step is due to a flaw in the VS Code launch task. The next step (or task) will fail if we do not execute this one first. The task expects that there is a **`node_modules`** subfolder in the project with some content in it.

   Thus, please add a dummy Node **`npm`** package to the project, e.g., use this command to install the popular moment library:

```
npm install moment -save
```

   This will add the moment library as a dependency to the **`package.json`** file (please verify this) and also create a folder **`node_modules`** in the project, to which it will add the downloaded moment package and its dependencies.

6. Open the debug side panel and select the correct launch task. In our case, it is "Docker Node.js Launch". Hit *F5* to start debugging the application as it is running inside the container.

7. Open the dashboard of the Docker Desktop application and notice how there is a container called **`debugnode-dev`** running. Click on it and then select the

2. Once again, press ⬆-⌘-p (*SHIFT-CTRL-p*) to open the command palette and locate and execute the command "**`docker: Initialize docker container`**". Select "**`Node JS`**" as the application platform when asked. This command will add/modify the **`launch.json`** file in the **`.vscode`** folder. It should now look like the following:

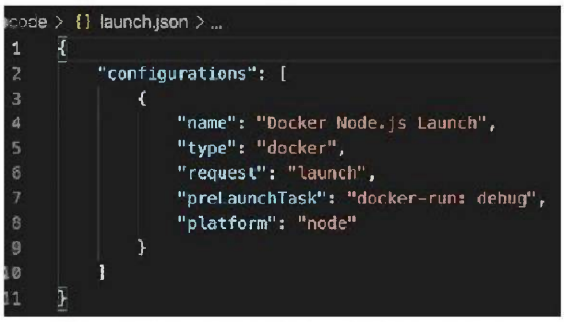


Figure 8.24: launch.json file for a Node JS application

3. You may also want to analyze the **`tasks.json`** file that was created by the preceding command.

4. To be able to work in a container, the Node JS Web server must listen at IP address 0.0.0.0 and not at the usual **`127.0.0.1`** (or **`localhost`**). Thus, before you proceed, please update the code in the file **`index.js`** and replace Line 3 with the following value:

```
const hostname = '0.0.0.0';
```

5. This following step is due to a flaw in the VS Code launch task. The next step (or task) will fail if we do not execute this one first. The task expects that there is a **`node_modules`** subfolder in the project with some content in it.

   Thus, please add a dummy Node **`npm`** package to the project, e.g., use this command to install the popular moment library:

```
npm install moment -save
```

   This will add the moment library as a dependency to the **`package.json`** file (please verify this) and also create a folder **`node_modules`** in the project, to which it will add the downloaded moment package and its dependencies.

6. Open the debug side panel and select the correct launch task. In our case, it is "Docker Node.js Launch". Hit *F5* to start debugging the application as it is running inside the container.

7. Open the dashboard of the Docker Desktop application and notice how there is a container called **`debugnode-dev`** running. Click on it and then select the

INSPECT tab to find out to which node the internal port `3000` is mapped on the host:

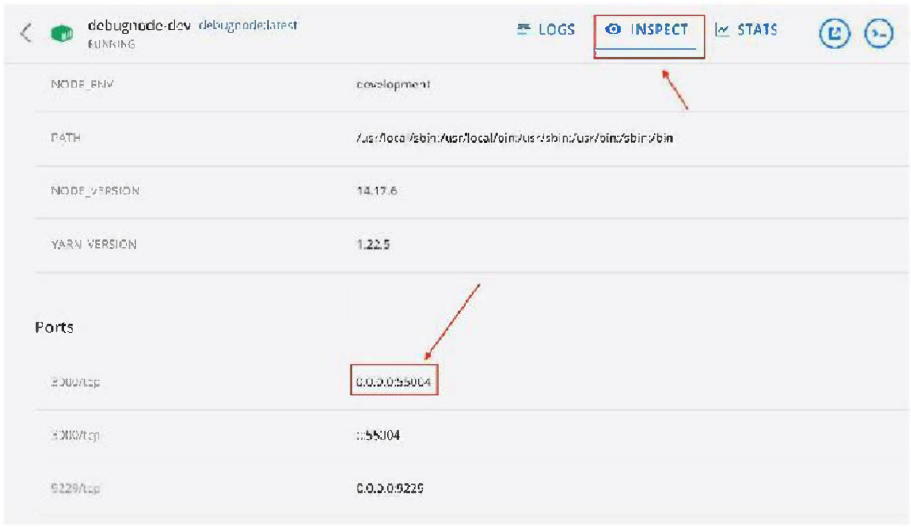


Figure 8.25: Inspecting the attributes of the debugnode-dev container

In the preceding case, the host port is `55004`, but this may be a different number in your case.

8. Open a new browser tab and navigate to `http://localhost:<host-port>`, where `<host-port>` is the port number to which the internal container port `3000` is mapped on the host. Take the number that you found out in the previous step. Observe how the breakpoint is hit in the application.

It is now time to explore how one can debug an application running in a container that has been written in the most popular language, Python.

Debugging a Python application

Last but not least, it is the turn on Python. Let us investigate how to run and debug a Python application running inside a container.

1. Open a new terminal and navigate to your source code folder for this chapter, for example:

   ```
   cd ~/src/ch08
   ```

2. Create a new sub-folder `debug-python` and navigate to it:

   ```
   cd debug-python && cd debug-python
   ```

INSPECT tab to find out to which node the internal port `3000` is mapped on the host:

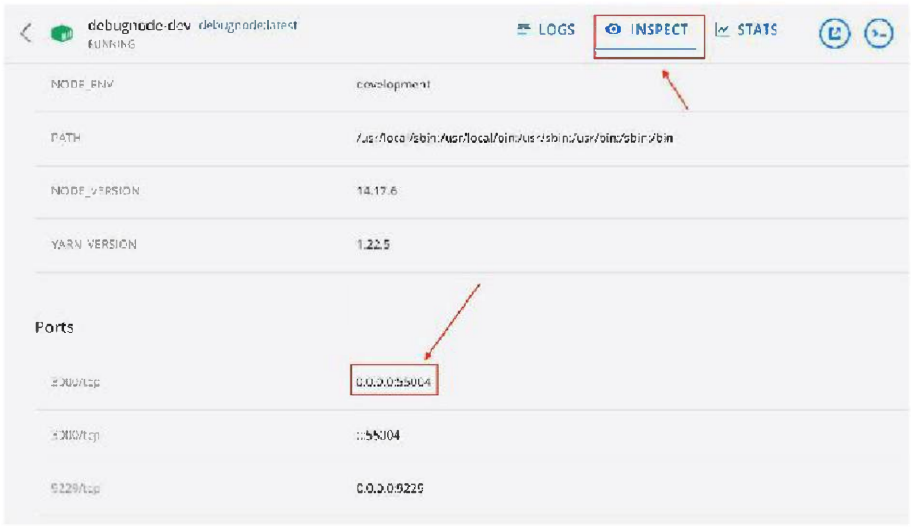


Figure 8.25: Inspecting the attributes of the debugnode-dev container

In the preceding case, the host port is `55004`, but this may be a different number in your case.

8. Open a new browser tab and navigate to `http://localhost:<host-port>`, where `<host-port>` is the port number to which the internal container port `3000` is mapped on the host. Take the number that you found out in the previous step. Observe how the breakpoint is hit in the application.

It is now time to explore how one can debug an application running in a container that has been written in the most popular language, Python.

Debugging a Python application

Last but not least, it is the turn on Python. Let us investigate how to run and debug a Python application running inside a container.

1. Open a new terminal and navigate to your source code folder for this chapter, for example:

   ```
   cd ~/src/ch08
   ```

2. Create a new sub-folder `debug-python` and navigate to it:

   ```
   cd debug-python && cd debug-python
   ```

3. Double-check that you have Python installed on your machine:

   ```
   python3 --version
   ```

   If this command fails, you may have to install Python first. Follow the following instructions:

   a. On a Mac, execute:

      ```
      brew install python
      ```

   b. On a Windows machine, execute the following:

      ```
      choco install python
      ```

   In both cases, use,

   ```
   pip3 --version
   ```

   to verify that the pip has been installed as part of the preceding installation procedure.

4. Create a virtual environment called `venv` for this folder:

   ```
   python3 -m venv
   ```

   and now activate this virtual environment

   `. venv/bin/activate` (or on Windows `venv\Scripts\activate`)

5. Now, install Flask in this environment:

   ```
   pip install flask
   ```

6. Open VS Code for this folder:

   ```
   code .
   ```

7. Let us now select the Python interpreter that VS Code is going to use. Hit ⬆-⌘-p (*SHIFT-CTRL-p*) to open the command palette and locate and execute the command "`Python: Select Interpreter`". From the list, select your virtual environment:

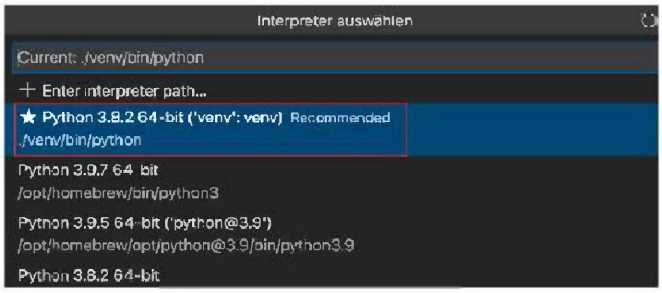


Figure 8.26: Selecting the Python interpreter in VS Code

3. Double-check that you have Python installed on your machine:

   ```
   python3 --version
   ```

   If this command fails, you may have to install Python first. Follow the following instructions:

   a. On a Mac, execute:

      ```
      brew install python
      ```

   b. On a Windows machine, execute the following:

      ```
      choco install python
      ```

   In both cases, use,

   ```
   pip3 --version
   ```

   to verify that the pip has been installed as part of the preceding installation procedure.

4. Create a virtual environment called `venv` for this folder:

   ```
   python3 -m venv
   ```

   and now activate this virtual environment

   `. venv/bin/activate` (or on Windows `venv\Scripts\activate`)

5. Now, install Flask in this environment:

   ```
   pip install flask
   ```

6. Open VS Code for this folder:

   ```
   code .
   ```

7. Let us now select the Python interpreter that VS Code is going to use. Hit ⇧-⌘-p (*SHIFT-CTRL-p*) to open the command palette and locate and execute the command "`Python: Select Interpreter`". From the list, select your virtual environment:

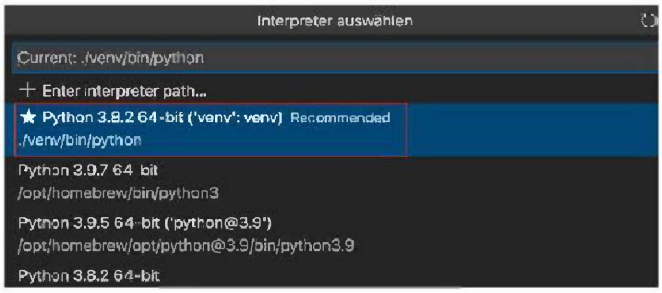


Figure 8.26: Selecting the Python interpreter in VS Code

8. We are going to create a minimal Flask application. Thus add a file called **app.py** to the project and add this content to it:

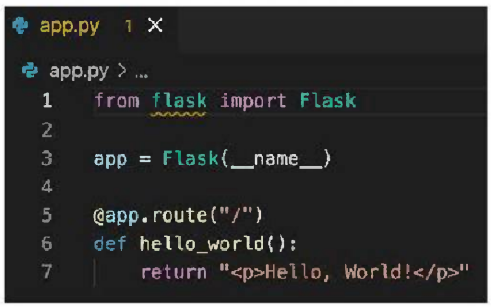


Figure 8.27: Minimal Python flask application

9. Since this is a Flask application, we also need to add Flask as a dependency. Thus, add a new file called **requirements.txt** to the project and add this as content:

   flask

   Yes, it is that simple, a single line with the keyword **flask**.

10. Add a breakpoint to Line 7 in the code of the **app.py** file. Hit F5 to start debugging the application. When asked, select "**Flask**" as the debug configuration.

11. Open a new browser tab and navigate to **http://localhost:5000**. Notice that the application execution will stop at the breakpoint, as expected.

Nothing unexpected here, I hear you say, and you are right. Once again, we only have shown how one can debug line by line a Python application running natively on your system using VS Code.

But we shall not stop here and learn how to debug the same application once it is running inside a container, as follows:

1. Use the "**docker: Add docker Files to Workspace**" command to generate a Dockerfile for our project. Select "**Python: Flask**" as the Application Platform, select **app.py** as an entry point, and leave the default value of **5000** as the port. Select "No" when asked whether or not to add Docker-compose support.

2. Hit *F5* to start debugging. Notice that automatically a browser tab is opened for you and that after a moment during which the container is prepared and launched, the breakpoint is hit in code:

8. We are going to create a minimal Flask application. Thus add a file called **app.py** to the project and add this content to it:

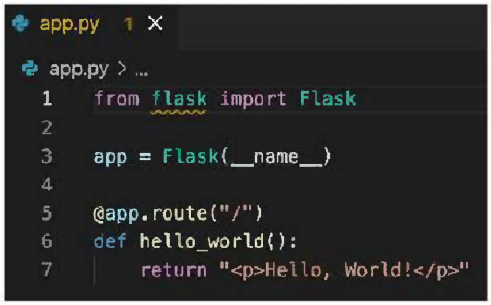


Figure 8.27: Minimal Python flask application

9. Since this is a Flask application, we also need to add Flask as a dependency. Thus, add a new file called **requirements.txt** to the project and add this as content:

   flask

   Yes, it is that simple, a single line with the keyword **flask**.

10. Add a breakpoint to Line 7 in the code of the **app.py** file. Hit F5 to start debugging the application. When asked, select "**Flask**" as the debug configuration.

11. Open a new browser tab and navigate to **http://localhost:5000**. Notice that the application execution will stop at the breakpoint, as expected.

Nothing unexpected here, I hear you say, and you are right. Once again, we only have shown how one can debug line by line a Python application running natively on your system using VS Code.

But we shall not stop here and learn how to debug the same application once it is running inside a container, as follows:

1. Use the "**docker: Add docker Files to Workspace**" command to generate a Dockerfile for our project. Select "**Python: Flask**" as the Application Platform, select **app.py** as an entry point, and leave the default value of **5000** as the port. Select "No" when asked whether or not to add Docker-compose support.

2. Hit *F5* to start debugging. Notice that automatically a browser tab is opened for you and that after a moment during which the container is prepared and launched, the breakpoint is hit in code:

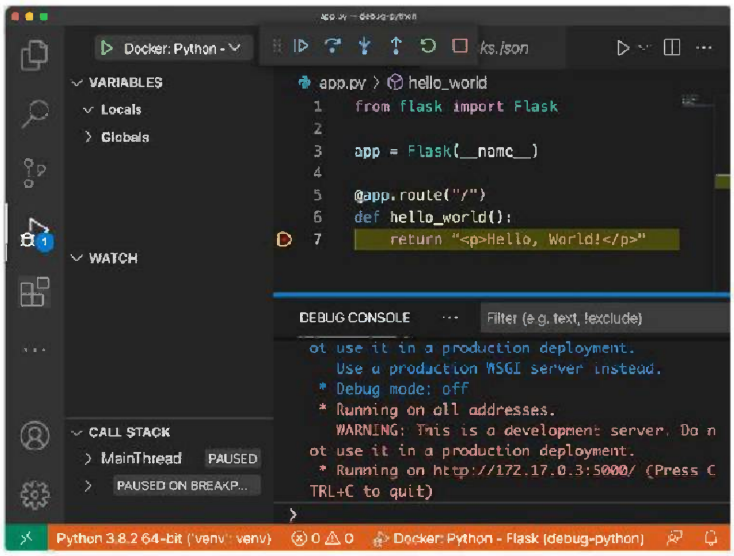


Figure 8.28: Debugging a Python app running inside a container

3. Open the dashboard on Docker Desktop and verify that a container called **debugpython-dev** is running. Analyze the logs of the container and use the **INSPECT** tab to figure out to which host port the container port 5000 is mapped.

Great, we are done with debugging for now. You should have a good grasp on how to configure your code editor to be able to debug line by line an application written in one of the four most popular languages that are running inside a container.

Next, we are going to see how we can combine writing automated tests, such as unit- and integration tests, and running them in a container.

Writing and running unit and integration tests

It is a recommended best practice to write automated tests that cover all or most of your applications' code. The simplest and usually most numerous tests are the unit tests that test a piece of code, such as a class of a function in isolation. Also important are integration tests that test a few collaborating components as a whole. This book is not about how to write tests, though, and consequently, we want to talk about the topic strictly in the context of Docker containers. How can we enable a frictionless test-driven development process when using containers?

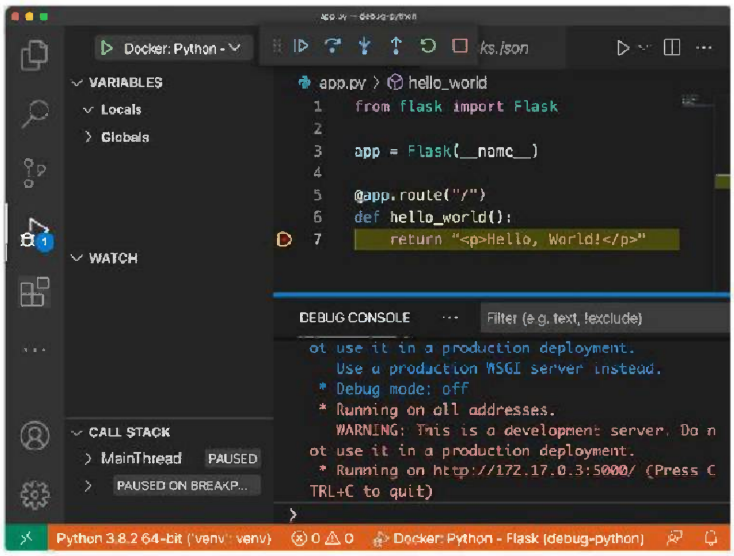


Figure 8.28: Debugging a Python app running inside a container

3. Open the dashboard on Docker Desktop and verify that a container called **debugpython-dev** is running. Analyze the logs of the container and use the **INSPECT** tab to figure out to which host port the container port 5000 is mapped.

Great, we are done with debugging for now. You should have a good grasp on how to configure your code editor to be able to debug line by line an application written in one of the four most popular languages that are running inside a container.

Next, we are going to see how we can combine writing automated tests, such as unit- and integration tests, and running them in a container.

Writing and running unit and integration tests

It is a recommended best practice to write automated tests that cover all or most of your applications' code. The simplest and usually most numerous tests are the unit tests that test a piece of code, such as a class of a function in isolation. Also important are integration tests that test a few collaborating components as a whole. This book is not about how to write tests, though, and consequently, we want to talk about the topic strictly in the context of Docker containers. How can we enable a frictionless test-driven development process when using containers?

Let us see how one can do this when, e.g., working with .NET.

1. Open a new terminal window and navigate to the source folder of this chapter:

   ```
   cd ~/src/ch08
   ```

2. Make a new subfolder called **testing-net** and navigate to it:

   ```
   mkdir testing-net && cd testing-net
   ```

3. Create a new .NET Console application called **App**:

   ```
   dotnet new console --name App
   ```

4. Create a new .NET xUnit project called **UnitTests**:

   ```
   dotnet new xunit --name UnitTests
   ```

   Note: It is a best practice not to add test code to the actual application. For this reason, we create a second project here, which will contain all test code.

5. Add the application project as a reference to the unit test project:

   ```
   dotnet add UnitTests reference App
   ```

6. Open the projects in VS Code with the following:

   ```
   code .
   ```

In the **UnitTests** project, rename the class to UnitTest1 to **CalculatorTests** and modify its code to look like this:

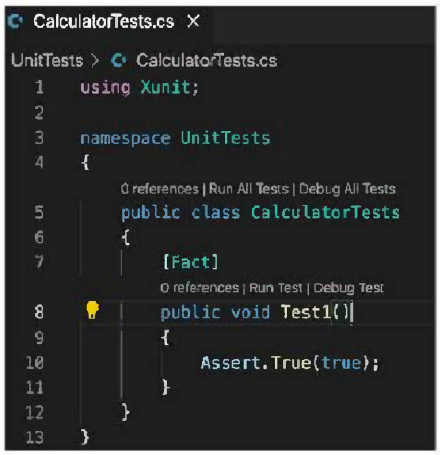


Figure 8.29: Empty unit test class in .NET sample application

Let us see how one can do this when, e.g., working with .NET.

1. Open a new terminal window and navigate to the source folder of this chapter:

   ```
   cd ~/src/ch08
   ```

2. Make a new subfolder called **testing-net** and navigate to it:

   ```
   mkdir testing-net && cd testing-net
   ```

3. Create a new .NET Console application called **App**:

   ```
   dotnet new console --name App
   ```

4. Create a new .NET xUnit project called **UnitTests**:

   ```
   dotnet new xunit --name UnitTests
   ```

   Note: It is a best practice not to add test code to the actual application. For this reason, we create a second project here, which will contain all test code.

5. Add the application project as a reference to the unit test project:

   ```
   dotnet add UnitTests reference App
   ```

6. Open the projects in VS Code with the following:

   ```
   code .
   ```

In the **UnitTests** project, rename the class to UnitTest1 to **CalculatorTests** and modify its code to look like this:

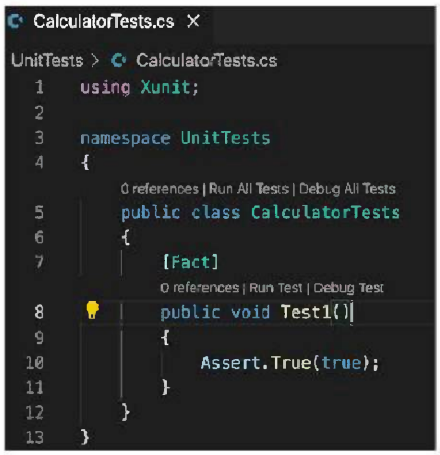


Figure 8.29: Empty unit test class in .NET sample application

7. Back in the terminal window, run the tests with the following:

   ```
   dotnet tests UnitTests
   ```

   The application is compiled, and the tests are executed. The output generated by xUnit should tell you that 1 test has passed successfully.

8. The preceding method is not very developer friendly since after each change in the code, and the developer has to rerun the previous command. Worry not, and there is a better way of doing it. Use the following command instead:

   ```
   dotnet watch test --project UnitTests
   ```

   The preceding command uses the watch to monitor for code changes and re-execute the command whenever the user changes the code. This time the output ends with the following:

   ```
   watch : Waiting for a file to change before restarting dotnet...
   ```

   This indicates that the tests will be re-executed automatically the moment some code changes are detected.

9. To verify that this is true, add a new test method to the class:

   ```
   [Fact]
   public void ShouldAdd2Numbers()
   {
       Assert.False(true);
   }
   ```

   Save the changes and observe that the compiler rebuilds the code and that the tests are re-executed. This time the output will tell you that one test passed and one failed (the new one). Nice!

Let us now containerize the whole thing in a way that allows for frictionless testing:

1. Add a Dockerfile to the root of the solution that is, directly into the **testing-net** folder with the following content:

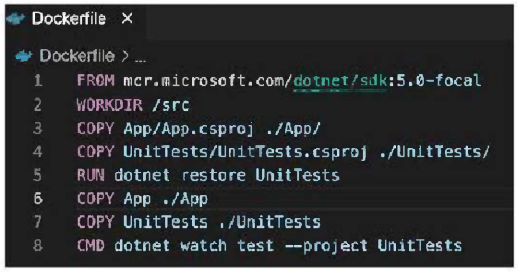


Figure 8.30: Dockerfile for running tests in .NET

7. Back in the terminal window, run the tests with the following:

   ```
   dotnet tests UnitTests
   ```

   The application is compiled, and the tests are executed. The output generated by xUnit should tell you that 1 test has passed successfully.

8. The preceding method is not very developer friendly since after each change in the code, and the developer has to rerun the previous command. Worry not, and there is a better way of doing it. Use the following command instead:

   ```
   dotnet watch test --project UnitTests
   ```

   The preceding command uses the watch to monitor for code changes and re-execute the command whenever the user changes the code. This time the output ends with the following:

   ```
   watch : Waiting for a file to change before restarting dotnet...
   ```

   This indicates that the tests will be re-executed automatically the moment some code changes are detected.

9. To verify that this is true, add a new test method to the class:

   ```
   [Fact]
   public void ShouldAdd2Numbers()
   {
       Assert.False(true);
   }
   ```

   Save the changes and observe that the compiler rebuilds the code and that the tests are re-executed. This time the output will tell you that one test passed and one failed (the new one). Nice!

Let us now containerize the whole thing in a way that allows for frictionless testing:

1. Add a Dockerfile to the root of the solution that is, directly into the **testing-net** folder with the following content:



Figure 8.30: Dockerfile for running tests in .NET

Note that we have to add two projects to the container, the application project and the unit tests project, where the latter references the former.

2. Build the Docker image with the following:

```
docker image build testing-net .
```

3. And run the tests with the following:

```
docker container run testing-net
```

The output should be similar to the one when you ran the tests natively. One test is passing, and one test is failing. Also, the output ends with the same statement:

```
watch : Waiting for a file to change before restarting dotnet...
```

Indicating that dotnet is waiting for code changes. But how are these code changes able to appear inside the container without us having to rebuild and restart the container? Right, we can use volume mapping!

4. Add a file called `docker-compose.yml` right next to the `Dockerfile` with the following content:

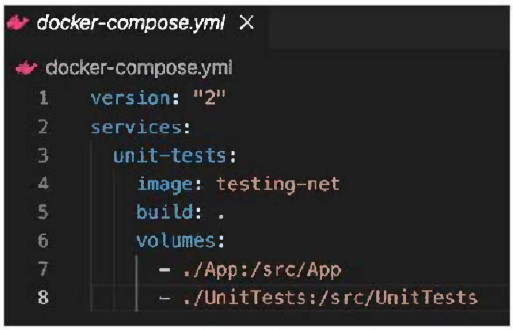


Figure 8.31: Docker compose file used to run unit tests continuously

5. This compose file uses volume mapping to dynamically update the code inside the container as you continue developing. Having such a Docker compose file is more convenient than remembering the "complicated" Docker run the command with the necessary volume mappings. It is much easier to instead start the container with the following:

```
docker compose up
```

6. Add a file called `Calculator.cs` to the `App` project with the following content:

Note that we have to add two projects to the container, the application project and the unit tests project, where the latter references the former.

2. Build the Docker image with the following:

```
docker image build testing-net .
```

3. And run the tests with the following:

```
docker container run testing-net
```

The output should be similar to the one when you ran the tests natively. One test is passing, and one test is failing. Also, the output ends with the same statement:

```
watch : Waiting for a file to change before restarting dotnet...
```

Indicating that dotnet is waiting for code changes. But how are these code changes able to appear inside the container without us having to rebuild and restart the container? Right, we can use volume mapping!

4. Add a file called `docker-compose.yml` right next to the `Dockerfile` with the following content:

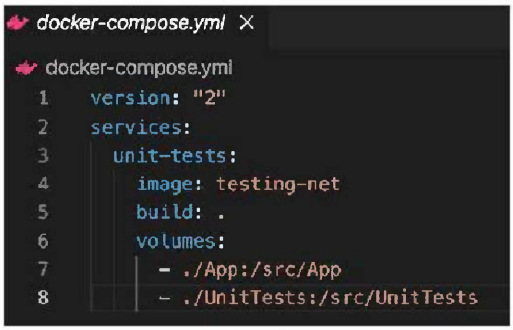


Figure 8.31: Docker compose file used to run unit tests continuously

5. This compose file uses volume mapping to dynamically update the code inside the container as you continue developing. Having such a Docker compose file is more convenient than remembering the "complicated" Docker run the command with the necessary volume mappings. It is much easier to instead start the container with the following:

```
docker compose up
```

6. Add a file called `Calculator.cs` to the `App` project with the following content:

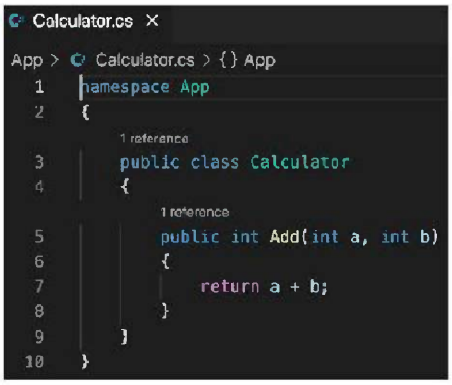


Figure 8.32: Calculator class

Notice that when you save your changes, the code inside the container is rebuilt, and the tests are re-executed as expected!

7. Change the second test method to look like the following:

```
[Fact]
public void ShouldAdd2Numbers()
{
    var calculator = new Calculator();
    Assert.Equal(4, calculator.Add(1,3));
}
```

Also, add the following using statement at the top of the file:

```
using App;
```

Notice that after saving, the tests are rerun, and now the result is that both tests pass. Good job!

You can use the very same approach to Dockerize unit or integration tests written in Java or any other language. The principle remains the same. You will want to continuously run tests in the background as new code is added or existing one is modified. When you run the code and tests inside a container, you will use volume mapping to dynamically update the code inside the container as you change it in your code editor.

Now, we are ready for the next topic; namely, we want to know how we can instrument our code and, as such, prepare it for the time it will run in production, and we cannot easily debug it.

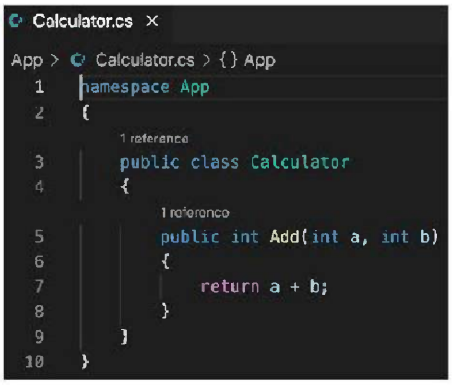


Figure 8.32: Calculator class

Notice that when you save your changes, the code inside the container is rebuilt, and the tests are re-executed as expected!

7. Change the second test method to look like the following:

```
[Fact]
public void ShouldAdd2Numbers()
{
    var calculator = new Calculator();
    Assert.Equal(4, calculator.Add(1,3));
}
```

Also, add the following using statement at the top of the file:

```
using App;
```

Notice that after saving, the tests are rerun, and now the result is that both tests pass. Good job!

You can use the very same approach to Dockerize unit or integration tests written in Java or any other language. The principle remains the same. You will want to continuously run tests in the background as new code is added or existing one is modified. When you run the code and tests inside a container, you will use volume mapping to dynamically update the code inside the container as you change it in your code editor.

Now, we are ready for the next topic; namely, we want to know how we can instrument our code and, as such, prepare it for the time it will run in production, and we cannot easily debug it.

Instrumenting a containerized app

Once you are perfectly happy with your application and it has been tested thoroughly, you will want to run it in production. But once it is running in production, you will not be able to debug in the same way as you are used to do while developing locally on your machine. Although it is technically possible to remote debug an application running in production, we strongly discourage you from doing so.

Thus, we need to come up with some alternative way of finding the root cause if the system malfunctions or shows unexpected behavior. The one thing developers resort to is instrumenting their applications such as that more useful runtime information is produced. One way of doing so is to use a logging library in your application and use it to produce meaningful log output. Typically, would want to log business-relevant information, exceptional cases, or just unexpected situations. It is common to classify log output into various severity levels. Let us see what levels exist and what their meaning is. The following descriptions are borrowed from the very popular Java logging library log4j:

- **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 helping to pin down potential problems if they occur.
- **INFO**: Normal application behavior or milestones.
- **WARN**: 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.

A logging library allows you to define so-called sinks to which the logging information is written to. This can be the console output, a file, a database, or a network connection, to name just a few. When working with Docker containers, it is important to know that the Docker infrastructure is expecting you to output log entries to the console, or more precisely, to STDOUT (and STDERR in case of an error). Thus, please configure your logging library to use the console as a sink. Docker will then make this information available to you via the `docker container logs` command. Other tools that build on top of Docker and may want to collect logging information also leverage this logging infrastructure or behavior provided by Docker.

Instrumenting a containerized app

Once you are perfectly happy with your application and it has been tested thoroughly, you will want to run it in production. But once it is running in production, you will not be able to debug in the same way as you are used to do while developing locally on your machine. Although it is technically possible to remote debug an application running in production, we strongly discourage you from doing so.

Thus, we need to come up with some alternative way of finding the root cause if the system malfunctions or shows unexpected behavior. The one thing developers resort to is instrumenting their applications such as that more useful runtime information is produced. One way of doing so is to use a logging library in your application and use it to produce meaningful log output. Typically, would want to log business-relevant information, exceptional cases, or just unexpected situations. It is common to classify log output into various severity levels. Let us see what levels exist and what their meaning is. The following descriptions are borrowed from the very popular Java logging library log4j:

- **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 helping to pin down potential problems if they occur.
- **INFO**: Normal application behavior or milestones.
- **WARN**: 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.

A logging library allows you to define so-called sinks to which the logging information is written to. This can be the console output, a file, a database, or a network connection, to name just a few. When working with Docker containers, it is important to know that the Docker infrastructure is expecting you to output log entries to the console, or more precisely, to STDOUT (and STDERR in case of an error). Thus, please configure your logging library to use the console as a sink. Docker will then make this information available to you via the `docker container logs` command. Other tools that build on top of Docker and may want to collect logging information also leverage this logging infrastructure or behavior provided by Docker.

Instrumenting a Java application

We will create a new Spring Boot Java application using VS Code and then use it to illustrate how we can instrument it. We will start with logging information directly produced by our application logic.

Application level logging

We will start instrumenting our application using the log4j logging library. But first, we have to create a Spring Boot application. Please follow the instructions as follows:

1. Open a new terminal window and navigate to the source folder for this chapter:

   ```
   cd ~/src/ch08
   ```

2. Open VS Code in this folder:

   ```
   code .
   ```

3. IN VS Code, press *SHIFT-CMD-P* (or *SHIFT-CTRL-P* in Windows or Linux) to open the command palette and search and execute the command "`Java: Create Java Project...`"

4. From the options presented, select **Spring Boot**:

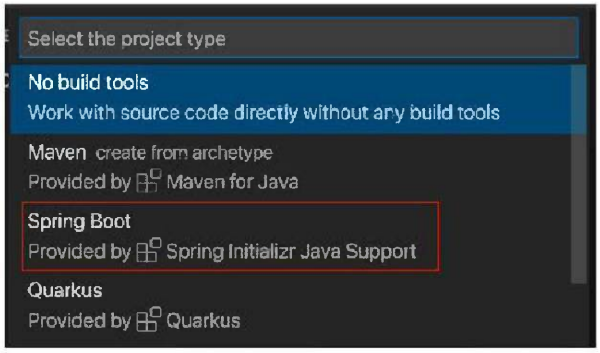


Figure 8.33: Creating a Spring Boot Java application from within VS Code

Then, in the dialog popping up:

1. Select **Gradle** as the Project Type.
2. Select **2.5.4** (or newer) as the Spring Boot version.
3. **Java** as the project language when asked.

Instrumenting a Java application

We will create a new Spring Boot Java application using VS Code and then use it to illustrate how we can instrument it. We will start with logging information directly produced by our application logic.

Application level logging

We will start instrumenting our application using the log4j logging library. But first, we have to create a Spring Boot application. Please follow the instructions as follows:

1. Open a new terminal window and navigate to the source folder for this chapter:

   ```
   cd ~/src/ch08
   ```

2. Open VS Code in this folder:

   ```
   code .
   ```

3. IN VS Code, press *SHIFT-CMD-P* (or *SHIFT-CTRL-P* in Windows or Linux) to open the command palette and search and execute the command "`Java: Create Java Project...`"

4. From the options presented, select **Spring Boot**:

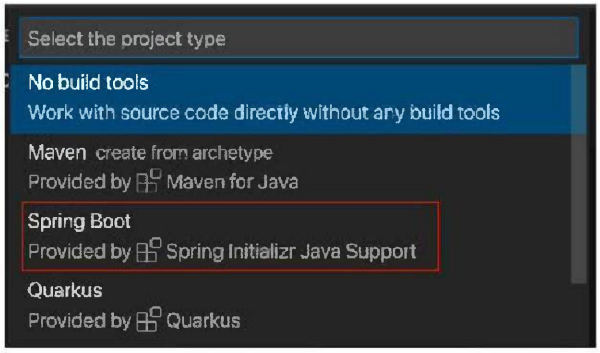


Figure 8.33: Creating a Spring Boot Java application from within VS Code

Then, in the dialog popping up:

1. Select **Gradle** as the Project Type.
2. Select **2.5.4** (or newer) as the Spring Boot version.
3. **Java** as the project language when asked.

4. Leave the default of Group Id (`com.example`).
5. Use `log-demo` as the Artifact Id.
6. Define **Jar** as the packaging type.
7. Select `11` as the Java version.
8. Select the following additional dependency: **Spring Web**
9. Have the code be generated into the `src/ch08` folder we created earlier for this chapter of the book.

5. Locate the class `src/main/java/com/example/logdemo/LogDemo.java`. This is our application class and the entry point into the application.
6. Next to the application class, create a new file called `HelloController.java` and add this content:

```
1  package com.example.logdemo;
2
3  import org.springframework.web.bind.annotation.GetMapping;
4  import org.springframework.web.bind.annotation.RestController;
5
6  @RestController
7  public class HelloController {
8
9      @GetMapping("/")
10     public String index() {
11         return "Greetings from Spring Boot!";
12     }
13
14 }
```

Figure 8.34: A simple controller in our Spring Boot app

7. In VS Code, open a new terminal and execute the following command:

   ```
   ./gradlew bootRun
   ```

   Notice in the output generated in the terminal how the app gets compiled, and then Gradle starts a dev Web server listening at `localhost:8080`
8. Open a new browser tab and navigate to `localhost:8080`. You should be greeted with this message: "Greetings from Spring Boot!"
9. Back to our `HelloController`, add the following two import statements:

   ```
   import org.slf4j.Logger;

   import org.slf4j.LoggerFactory;
   ```

4. Leave the default of Group Id (`com.example`).
5. Use `log-demo` as the Artifact Id.
6. Define **Jar** as the packaging type.
7. Select `11` as the Java version.
8. Select the following additional dependency: **Spring Web**
9. Have the code be generated into the `src/ch08` folder we created earlier for this chapter of the book.

5. Locate the class `src/main/java/com/example/logdemo/LogDemo.java`. This is our application class and the entry point into the application.
6. Next to the application class, create a new file called `HelloController.java` and add this content:

```
1  package com.example.logdemo;
2
3  import org.springframework.web.bind.annotation.GetMapping;
4  import org.springframework.web.bind.annotation.RestController;
5
6  @RestController
7  public class HelloController {
8
9      @GetMapping("/")
10     public String index() {
11         return "Greetings from Spring Boot!";
12     }
13
14 }
```

Figure 8.34: A simple controller in our Spring Boot app

7. In VS Code, open a new terminal and execute the following command:

   ```
   ./gradlew bootRun
   ```

   Notice in the output generated in the terminal how the app gets compiled, and then Gradle starts a dev Web server listening at `localhost:8080`
8. Open a new browser tab and navigate to `localhost:8080`. You should be greeted with this message: "Greetings from Spring Boot!"
9. Back to our `HelloController`, add the following two import statements:

   ```
   import org.slf4j.Logger;

   import org.slf4j.LoggerFactory;
   ```

10. Then, add an instance variable called `logger` to the class:

```
Logger = LoggerFactory.getLogger(HelloController.class);
```

11. Next, extend the index method as follows, such that it produces a few log entries of different severity:

```
@GetMapping("/")
public String index() {
    logger.trace("Sample TRACE message");
    logger.debug("Sample DEBUG message");
    logger.info("Sample INFO message");
    logger.warn("Sample WARN message");
    logger.error("Sample ERROR message");

    return "Greetings from my Spring Boot!";
}
```

Figure 8.35: Producing log output of different severity

12. Run the application again and notice the console output generated once you open a new browser tab at `localhost:8080`:

```
Online/src/ch08/log-demo/bin/main started by gabriel in /Users/gabriel/OneDrive/Publications/BPB Online/s
rc/ch08/log-demo)
2021-09-19 14:00:22.855 DEBUG 22141 --- [           main] com.example.logdemo.LogDemoApplication   : Runnin
g with Spring Boot v2.5.4, Spring v5.3.9
2021-09-19 14:00:22.855  INFO 22141 --- [           main] com.example.logdemo.LogDemoApplication   : No act
ive profile set, falling back to default profiles: default
2021-09-19 14:00:23.657  INFO 22141 --- [           main] com.example.logdemo.LogDemoApplication   : Starte
d LogDemoApplication in 1.056 seconds (JVM running for 1.238)
2021-09-19 14:00:33.595 TRACE 22141 --- [nio-8080-exec-1] com.example.logdemo.HelloController      : Sample
 TRACE message
2021-09-19 14:00:33.595 DEBUG 22141 --- [nio-8080-exec-1] com.example.logdemo.HelloController      : Sample
 DEBUG message
2021-09-19 14:00:33.595  INFO 22141 --- [nio-8080-exec-1] com.example.logdemo.HelloController      : Sample
 INFO message
2021-09-19 14:00:33.595  WARN 22141 --- [nio-8080-exec-1] com.example.logdemo.HelloController      : Sample
 WARN message
2021-09-19 14:00:33.595 ERROR 22141 --- [nio-8080-exec-1] com.example.logdemo.HelloController      : Sample
 ERROR message
```

Figure 8.36: Log output generated by instrumented Java application

We can clearly see our five log entries we generated in the `HelloController` class, and also, we should notice the color coding used by Spring Boot. This makes identifying warnings and errors even more straightforward.

Using Spring Boot actuators

Spring Boot includes a number of features to monitor and manage applications that run in production. The actuator module provides features such as health and metrics endpoints. To enable those features, we only need to add the `spring-boot-actuator` library to the project and add some minor configurations and tweak the main method. Follow this step-by-step guide to explore what the actuator offers:

10. Then, add an instance variable called `logger` to the class:

```
Logger = LoggerFactory.getLogger(HelloController.class);
```

11. Next, extend the index method as follows, such that it produces a few log entries of different severity:

```
@GetMapping("/")
public String index() {
    logger.trace("Sample TRACE message");
    logger.debug("Sample DEBUG message");
    logger.info("Sample INFO message");
    logger.warn("Sample WARN message");
    logger.error("Sample ERROR message");

    return "Greetings from my Spring Boot!";
}
```

Figure 8.35: Producing log output of different severity

12. Run the application again and notice the console output generated once you open a new browser tab at `localhost:8080`:

```
Online/src/ch08/log-demo/bin/main started by gabriel in /Users/gabriel/OneDrive/Publications/BPB Online/s
rc/ch08/log-demo)
2021-09-19 14:00:22.855 DEBUG 22141 --- [           main] com.example.logdemo.LogDemoApplication   : Runnin
g with Spring Boot v2.5.4, Spring v5.3.9
2021-09-19 14:00:22.855  INFO 22141 --- [           main] com.example.logdemo.LogDemoApplication   : No act
ive profile set, falling back to default profiles: default
2021-09-19 14:00:23.657  INFO 22141 --- [           main] com.example.logdemo.LogDemoApplication   : Starte
d LogDemoApplication in 1.056 seconds (JVM running for 1.238)
2021-09-19 14:00:33.595 TRACE 22141 --- [nio-8080-exec-1] com.example.logdemo.HelloController      : Sample
 TRACE message
2021-09-19 14:00:33.595 DEBUG 22141 --- [nio-8080-exec-1] com.example.logdemo.HelloController      : Sample
 DEBUG message
2021-09-19 14:00:33.595  INFO 22141 --- [nio-8080-exec-1] com.example.logdemo.HelloController      : Sample
 INFO message
2021-09-19 14:00:33.595  WARN 22141 --- [nio-8080-exec-1] com.example.logdemo.HelloController      : Sample
 WARN message
2021-09-19 14:00:33.595 ERROR 22141 --- [nio-8080-exec-1] com.example.logdemo.HelloController      : Sample
 ERROR message
```

Figure 8.36: Log output generated by instrumented Java application

We can clearly see our five log entries we generated in the `HelloController` class, and also, we should notice the color coding used by Spring Boot. This makes identifying warnings and errors even more straightforward.

Using Spring Boot actuators

Spring Boot includes a number of features to monitor and manage applications that run in production. The actuator module provides features such as health and metrics endpoints. To enable those features, we only need to add the `spring-boot-actuator` library to the project and add some minor configurations and tweak the main method. Follow this step-by-step guide to explore what the actuator offers:

1. Open the `build.gradle` file and add the following line to the `dependencies` section:

   ```
   implementation 'org.springframework.boot:spring-boot-starter-actuator'
   ```

2. Locate the `application.properties` file in the folder `src/main/resources` and add the following line to it to enable all HTTP endpoints:

   ```
   management.endpoints.web.exposure.include=*
   ```

3. To probe the health endpoint, use the following:

   ```
   curl localhost:8080/actuator/health
   ```

   The answer should be the following:

   ```
   {"status":"UP"}
   ```

4. To get the list of all metrics, use the following:

   ```
   curl localhost:8080/actuator/metrics
   ```

5. To get startup information, we need to change our main method a bit. Locate the `LogDemoApplication` class and change the `main` method as follows:

```
public static void main(String[] args) {
    // SpringApplication.run(LogDemoApplication.class, args);
    SpringApplication app = new SpringApplication(LogDemoApplication.class);
    app.setApplicationStartup(new BufferingApplicationStartup(1000));
    app.run(args);
}
```

Figure 8.37: The main method enabling the collection of startup information

   Once you have changed the `main` method, use the following command:

   ```
   curl localhost:8080/actuator/startup
   ```

   to get valuable startup information provided by the Spring Boot framework

6. **Optional:** experiment with other endpoints provided by the actuator module. Find the details in the Spring Boot documentation (**https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html**).

With this, we have reached the end of this chapter. Although we showed many examples, we have merely scratched the surface of what is possible when debugging, testing, and instrumenting your containerized applications.

Conclusion

In this chapter, we have learned how to remove friction from the development cycle when using containers by introducing hot reload of code. This method has long

1. Open the `build.gradle` file and add the following line to the `dependencies` section:

   ```
   implementation 'org.springframework.boot:spring-boot-starter-actuator'
   ```

2. Locate the `application.properties` file in the folder `src/main/resources` and add the following line to it to enable all HTTP endpoints:

   ```
   management.endpoints.web.exposure.include=*
   ```

3. To probe the health endpoint, use the following:

   ```
   curl localhost:8080/actuator/health
   ```

   The answer should be the following:

   ```
   {"status":"UP"}
   ```

4. To get the list of all metrics, use the following:

   ```
   curl localhost:8080/actuator/metrics
   ```

5. To get startup information, we need to change our main method a bit. Locate the `LogDemoApplication` class and change the `main` method as follows:

```
public static void main(String[] args) {
    // SpringApplication.run(LogDemoApplication.class, args);
    SpringApplication app = new SpringApplication(LogDemoApplication.class);
    app.setApplicationStartup(new BufferingApplicationStartup(1000));
    app.run(args);
}
```

Figure 8.37: The main method enabling the collection of startup information

   Once you have changed the `main` method, use the following command:

   ```
   curl localhost:8080/actuator/startup
   ```

   to get valuable startup information provided by the Spring Boot framework

6. **Optional:** experiment with other endpoints provided by the actuator module. Find the details in the Spring Boot documentation (**https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html**).

With this, we have reached the end of this chapter. Although we showed many examples, we have merely scratched the surface of what is possible when debugging, testing, and instrumenting your containerized applications.

Conclusion

In this chapter, we have learned how to remove friction from the development cycle when using containers by introducing hot reload of code. This method has long

been known to advanced developers, but we added containers to the mix. We have shown how one can achieve hot reloading when coding in various languages such as .NET, Java, Node, or Python.

We then showed how one can debug line-by-line code that is running inside a container. We used the same principles that developers use when remote debugging an application. An application running inside a container is remote to your code editor. You have learned again how to do this for various languages and when using VS Code as your editor.

Another topic we discussed is how the right use of Docker containers can accelerate our test-driven development. We have shown it with an example coded in .NET C#. But the principle once again is similar to all other known and popular development languages. We can use volume mapping and hot reloading to enable this continuous testing cycle.

Finally, we ended the chapter by showing how one can instrument an application that is destined to run in production where debugging is hardly possible. We have shown how to produce application-level information using a logging framework. We then discussed how one can add more system or framework-level information to the mix, taking as an example a Java Spring Boot application.

In the upcoming chapter, we will learn how to establish a complete CI/CD pipeline using containers. Stay tuned.

Questions

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

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

Answers

Here are the answers to the questions:

1. **Possible answers:** (a) Volume mount your source code in the container; (b) use a tool that automatically restarts the app running inside the container when code changes are detected; and (c) configure your container for remote debugging.

been known to advanced developers, but we added containers to the mix. We have shown how one can achieve hot reloading when coding in various languages such as .NET, Java, Node, or Python.

We then showed how one can debug line-by-line code that is running inside a container. We used the same principles that developers use when remote debugging an application. An application running inside a container is remote to your code editor. You have learned again how to do this for various languages and when using VS Code as your editor.

Another topic we discussed is how the right use of Docker containers can accelerate our test-driven development. We have shown it with an example coded in .NET C#. But the principle once again is similar to all other known and popular development languages. We can use volume mapping and hot reloading to enable this continuous testing cycle.

Finally, we ended the chapter by showing how one can instrument an application that is destined to run in production where debugging is hardly possible. We have shown how to produce application-level information using a logging framework. We then discussed how one can add more system or framework-level information to the mix, taking as an example a Java Spring Boot application.

In the upcoming chapter, we will learn how to establish a complete CI/CD pipeline using containers. Stay tuned.

Questions

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

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

Answers

Here are the answers to the questions:

1. **Possible answers:** (a) Volume mount your source code in the container; (b) use a tool that automatically restarts the app running inside the container when code changes are detected; and (c) 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 the lack of the necessary language or framework.
4. Once the 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.

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 recently introduced containers. Developers must use containers to run and test their code locally, as one of our company policies. A lot of developers are complaining that this introduces more friction into their development process. What went possibly wrong?
2. Describe the steps necessary to debug a Java application running inside a container, assuming you are using VS Code as your code editor.
3. Why should we containerize our unit and integration tests? What are the pros? Are there any cons?
4. Tell us why it is so important to instrument an application prior to run it in production.

Sample answers to interview questions

Here are some sample answers to the interview questions:

1. Most probably, there has not been a standard on how to use containers correctly established among the teams. One important factor is to know how to use hot reloading and volume mapping when running code inside a container and possibly even debug it. It is very important to enable dynamic code updates inside a container via volume mapping when a developer is changing or adding code in their code editor.

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 the lack of the necessary language or framework.
4. Once the 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.

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 recently introduced containers. Developers must use containers to run and test their code locally, as one of our company policies. A lot of developers are complaining that this introduces more friction into their development process. What went possibly wrong?
2. Describe the steps necessary to debug a Java application running inside a container, assuming you are using VS Code as your code editor.
3. Why should we containerize our unit and integration tests? What are the pros? Are there any cons?
4. Tell us why it is so important to instrument an application prior to run it in production.

Sample answers to interview questions

Here are some sample answers to the interview questions:

1. Most probably, there has not been a standard on how to use containers correctly established among the teams. One important factor is to know how to use hot reloading and volume mapping when running code inside a container and possibly even debug it. It is very important to enable dynamic code updates inside a container via volume mapping when a developer is changing or adding code in their code editor.

2. Debugging an application running inside a container is the same as debugging a remote application. The latter is a well-known technique for developers, and most debuggers support this scenario. The main steps to debug a Java application running inside a container are:
    a. Create a new or use an existing Java application
    b. Add a Dockerfile to the solution and make sure to define the JAVA_TOOL_OPTIONS environment variable in a way that the Java runtime runs in debug mode. Specifically, you have to define on which port the debugger server is listening
    c. Build an image using this Dockerfile
    d. Run a container from the preceding Docker image and make sure to map the debugger port (for example, 8000) to the equivalent host port
    e. Configure a launch task in VS Code that attaches the debugger to localhost and the debugger port mapped to the host
    f. Run VS Code in debug mode by pressing *F5*
3. Containerizing tests the right way, using volume mapping and hot reloading, enables a developer to continuously run tests in the background while she is working in her code editor and updating code. The use of Dockerfiles and Docker-compose files and sharing them as part of the project with all team members guarantees that the tests are always run the same way and with the same boundary conditions on each developer's machine. Furthermore, the same technique can be used to run the tests as part of the build in the CI/CD pipeline.
4. An application running in production cannot and should not be (remote-) debugged. But it is a fact that even the most thoroughly tested application sooner or later will show unexpected behavior in production. To give the developers enough information on what went wrong and why it went wrong is a key factor in locating the root cause of the unwanted application behavior. Instrumentation helps in this area by providing application, framework, and infrastructure-level information to the developers.

2. Debugging an application running inside a container is the same as debugging a remote application. The latter is a well-known technique for developers, and most debuggers support this scenario. The main steps to debug a Java application running inside a container are:
    a. Create a new or use an existing Java application
    b. Add a Dockerfile to the solution and make sure to define the JAVA_TOOL_OPTIONS environment variable in a way that the Java runtime runs in debug mode. Specifically, you have to define on which port the debugger server is listening
    c. Build an image using this Dockerfile
    d. Run a container from the preceding Docker image and make sure to map the debugger port (for example, 8000) to the equivalent host port
    e. Configure a launch task in VS Code that attaches the debugger to localhost and the debugger port mapped to the host
    f. Run VS Code in debug mode by pressing *F5*
3. Containerizing tests the right way, using volume mapping and hot reloading, enables a developer to continuously run tests in the background while she is working in her code editor and updating code. The use of Dockerfiles and Docker-compose files and sharing them as part of the project with all team members guarantees that the tests are always run the same way and with the same boundary conditions on each developer's machine. Furthermore, the same technique can be used to run the tests as part of the build in the CI/CD pipeline.
4. An application running in production cannot and should not be (remote-) debugged. But it is a fact that even the most thoroughly tested application sooner or later will show unexpected behavior in production. To give the developers enough information on what went wrong and why it went wrong is a key factor in locating the root cause of the unwanted application behavior. Instrumentation helps in this area by providing application, framework, and infrastructure-level information to the developers.

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



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 9

Establishing an Automated Build Pipeline

Introduction

No software 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 on, we will discuss and compare some free and some commercial SaaS offerings and implement a full CI/CD pipeline using GitHub actions as an example. We then provide some tips on how to prepare for a job interview for a DevOps engineer covering mainly the CI part. We conclude the chapter with a recap of the key learnings. Tools used will be the terminal, Docker Desktop, and VS Code.

Structure

In this chapter, we will discuss the following topics:

- Building a Docker image using GitHub actions
- Using Jenkins to setup a build pipeline
- Discussing some cloud-based SaaS offerings for your CI/CD pipeline

Without further ado, let us start learning how to build a Docker image using GitHub actions.

CHAPTER 9

Establishing an Automated Build Pipeline

Introduction

No software 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 on, we will discuss and compare some free and some commercial SaaS offerings and implement a full CI/CD pipeline using GitHub actions as an example. We then provide some tips on how to prepare for a job interview for a DevOps engineer covering mainly the CI part. We conclude the chapter with a recap of the key learnings. Tools used will be the terminal, Docker Desktop, and VS Code.

Structure

In this chapter, we will discuss the following topics:

- Building a Docker image using GitHub actions
- Using Jenkins to setup a build pipeline
- Discussing some cloud-based SaaS offerings for your CI/CD pipeline

Without further ado, let us start learning how to build a Docker image using GitHub actions.

Objectives

After reading this chapter, the reader will be able to build a fully automated CI/CD pipeline, using either GitHub actions or Jenkins, to build, test and ship a custom-built Docker image containing their own application service. Furthermore, the reader will have a good understanding, which are some of the most popular options when selecting a fully managed offering for CI/CD pipelines.

Building a Docker image using GitHub actions

GitHub is probably the most popular source code management solution. It is completely free for any public repository, including all open-source software projects. Recently, the addition of GitHub actions has tremendously increased the usefulness of this offering and made it a true development platform instead of "just" a source code management tool. In this section, we want to show how GitHub actions can be used to create a **continuous integration (CI)** pipeline for a code repository. Let us start by defining a useful project first.

Preparing an application and a GitHub repository

To demonstrate how one can use GitHub actions to create a CI pipeline, we will first prepare a Node JS application. We will be using the demo application we created in *Chapter 7, Managing Complex Apps with Docker Compose*, for this purpose. Please follow the following step-by-step instructions to do so:

1. Open a new terminal window and navigate to the source code root folder

   ```
   cd ~/src
   ```

2. Copy the demo-app folder containing our sample multi-service application we built in *Chapter 7, Managing Complex Apps with Docker Compose* to the this chapter's folder:

   ```
   cp -r ch07/demo-app ch09/
   ```

3. Navigate into the demo-app folder:

   ```
   cd ch09/demo-app
   ```

4. Initialize this project as a Git repository:

   ```
   git init
   ```

Objectives

After reading this chapter, the reader will be able to build a fully automated CI/CD pipeline, using either GitHub actions or Jenkins, to build, test and ship a custom-built Docker image containing their own application service. Furthermore, the reader will have a good understanding, which are some of the most popular options when selecting a fully managed offering for CI/CD pipelines.

Building a Docker image using GitHub actions

GitHub is probably the most popular source code management solution. It is completely free for any public repository, including all open-source software projects. Recently, the addition of GitHub actions has tremendously increased the usefulness of this offering and made it a true development platform instead of "just" a source code management tool. In this section, we want to show how GitHub actions can be used to create a **continuous integration (CI)** pipeline for a code repository. Let us start by defining a useful project first.

Preparing an application and a GitHub repository

To demonstrate how one can use GitHub actions to create a CI pipeline, we will first prepare a Node JS application. We will be using the demo application we created in *Chapter 7, Managing Complex Apps with Docker Compose*, for this purpose. Please follow the following step-by-step instructions to do so:

1. Open a new terminal window and navigate to the source code root folder

   ```
   cd ~/src
   ```

2. Copy the demo-app folder containing our sample multi-service application we built in *Chapter 7, Managing Complex Apps with Docker Compose* to the this chapter's folder:

   ```
   cp -r ch07/demo-app ch09/
   ```

3. Navigate into the demo-app folder:

   ```
   cd ch09/demo-app
   ```

4. Initialize this project as a Git repository:

   ```
   git init
   ```

5. Open the demo application in VS Code:

   ```
   code.
   ```

6. Add a file **.gitignore** to the **demo-app** folder with the following content:



Figure 9.1: .gitignore file for the demo application

7. Stage all files and create a first commit with the following:

   ```
   git add .
   git commit -m "First commit"
   ```

8. **Note**: If you have not done this earlier, you may need to initialize Git with your name and e-mail address before you can execute the preceding command:

   ```
   git config --global user.email "you@example.com"
   git config --global user.name "Your Name"
   ```

9. Open a new browser tab and navigate to GitHub at **https://github.com**.

10. Login to your GitHub account. If you do not have a GitHub account yet, then this is a good time to create one now. It is completely free!

11. Create a new repository on GitHub and call it **ch09-demo-app**.

12. Back in the terminal window, execute the following command to add the newly created GitHub repository as a remote called origin:

    ```
    git remote add origin https://github.com/docker-for-job-seekers/
    ch09-demo-app.git
    ```

    IMPORTANT: Please make sure to replace the part **docker-for-job-seekers** in the preceding URL with your own GitHub account name.

13. Call the current branch you are in **main** and push the code to the repository:

    ```
    git branch -M main
    git push -u origin main
    ```

5. Open the demo application in VS Code:

   ```
   code.
   ```

6. Add a file **.gitignore** to the **demo-app** folder with the following content:



Figure 9.1: .gitignore file for the demo application

7. Stage all files and create a first commit with the following:

   ```
   git add .
   git commit -m "First commit"
   ```

8. **Note**: If you have not done this earlier, you may need to initialize Git with your name and e-mail address before you can execute the preceding command:

   ```
   git config --global user.email "you@example.com"
   git config --global user.name "Your Name"
   ```

9. Open a new browser tab and navigate to GitHub at **https://github.com**.

10. Login to your GitHub account. If you do not have a GitHub account yet, then this is a good time to create one now. It is completely free!

11. Create a new repository on GitHub and call it **ch09-demo-app**.

12. Back in the terminal window, execute the following command to add the newly created GitHub repository as a remote called origin:

    ```
    git remote add origin https://github.com/docker-for-job-seekers/
    ch09-demo-app.git
    ```

    IMPORTANT: Please make sure to replace the part **docker-for-job-seekers** in the preceding URL with your own GitHub account name.

13. Call the current branch you are in **main** and push the code to the repository:

    ```
    git branch -M main
    git push -u origin main
    ```

14. If you go back to GitHub and browse your repository, you should see something like the following:

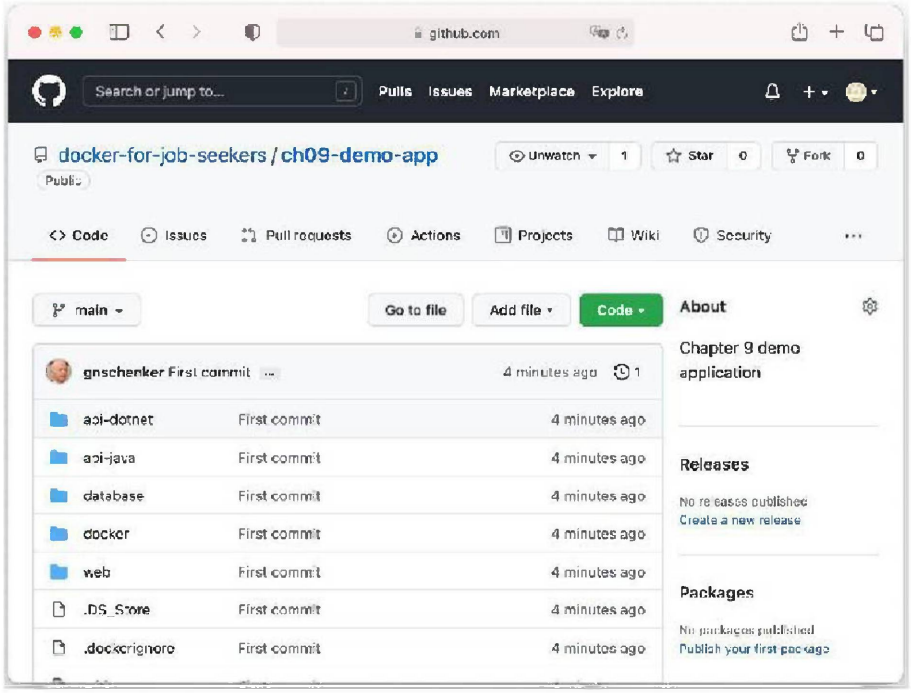


Figure 9.2: The GitHub repository for the demo application

Great, now we are ready to start working with GitHub actions.

Introducing GitHub actions and workflows

GitHub Actions is a powerful tool that allows developers to automate various tasks related to their software development workflow. Some of the things that GitHub Actions can be used for include:

1. **Continuous integration (CI) and continuous delivery (CD)**: GitHub Actions can be used to automatically build, test, and deploy code changes to different environments.
2. **Automated testing**: GitHub Actions can be used to run automated tests (unit tests, integration tests, and so on) on every code change, and to report the results back to the developer.

14. If you go back to GitHub and browse your repository, you should see something like the following:

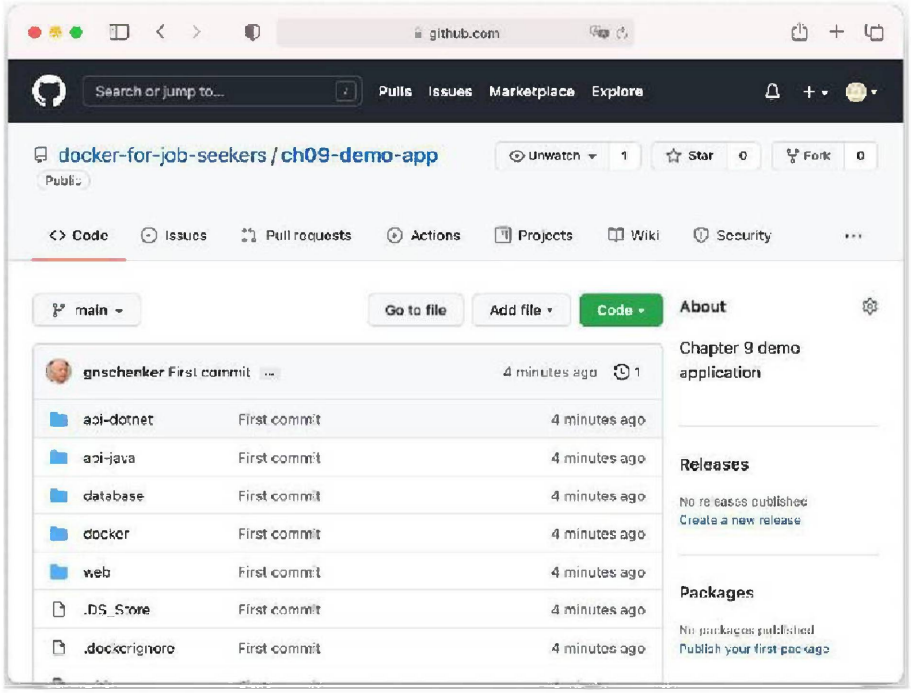


Figure 9.2: The GitHub repository for the demo application

Great, now we are ready to start working with GitHub actions.

Introducing GitHub actions and workflows

GitHub Actions is a powerful tool that allows developers to automate various tasks related to their software development workflow. Some of the things that GitHub Actions can be used for include:

1. **Continuous integration (CI) and continuous delivery (CD)**: GitHub Actions can be used to automatically build, test, and deploy code changes to different environments.
2. **Automated testing**: GitHub Actions can be used to run automated tests (unit tests, integration tests, and so on) on every code change, and to report the results back to the developer.

3. **Code quality checks**: GitHub Actions can be used to automatically check the code for quality, including checks for coding style, security vulnerabilities, and other issues.
4. **Building and deploying applications**: GitHub Actions can be used to automatically build, package, and deploy applications to different environments, including cloud providers such as AWS, Azure, and Google Cloud, and container registries like Docker Hub.
5. **Automating workflows and tasks**: GitHub Actions can be used to automate tasks such as sending notifications, updating documentation, and much more.
6. **Automating deployments**: GitHub Actions can be used to automate deployments and releases by creating and tagging releases and deploying them to different environments.
7. **Automating tasks on schedule**: GitHub Actions can be used to schedule tasks to run at a specific time; this is useful for tasks like backups, reports, and other scheduled tasks.
8. **Collaboration**: GitHub Actions can be used to collaborate with other developers by triggering actions when a pull request is opened or when a new branch is created, and many other events.
9. **Custom actions**: Developers can create their own actions, share them with the community, and use them in their own projects.

Please note that GitHub Actions is a powerful tool that allows for many different use-cases and scenarios, this list is not exhaustive, and there are many other ways to use it.

As mentioned previously, it is possible to use GitHub actions to automate tasks as a result of an event that happens. Events can be a push operation to a GitHub repository or a merge request to the repository. In GitHub, you can define one to many workflows that organize the actions you want to execute.

Workflows contain one-to-many jobs. Each job is made up of one-to-many items. An item can be a GitHub action or a shell script, or a shell command.

There are many predefined GitHub actions that we can use in our workflows. But it is also possible to author a custom action. The latter is outside of the scope of this book, though.

1. Checkout to a new feature branch called `actions-first-steps`:

```
git checkout -b actions-first-steps
```

3. **Code quality checks**: GitHub Actions can be used to automatically check the code for quality, including checks for coding style, security vulnerabilities, and other issues.
4. **Building and deploying applications**: GitHub Actions can be used to automatically build, package, and deploy applications to different environments, including cloud providers such as AWS, Azure, and Google Cloud, and container registries like Docker Hub.
5. **Automating workflows and tasks**: GitHub Actions can be used to automate tasks such as sending notifications, updating documentation, and much more.
6. **Automating deployments**: GitHub Actions can be used to automate deployments and releases by creating and tagging releases and deploying them to different environments.
7. **Automating tasks on schedule**: GitHub Actions can be used to schedule tasks to run at a specific time; this is useful for tasks like backups, reports, and other scheduled tasks.
8. **Collaboration**: GitHub Actions can be used to collaborate with other developers by triggering actions when a pull request is opened or when a new branch is created, and many other events.
9. **Custom actions**: Developers can create their own actions, share them with the community, and use them in their own projects.

Please note that GitHub Actions is a powerful tool that allows for many different use-cases and scenarios, this list is not exhaustive, and there are many other ways to use it.

As mentioned previously, it is possible to use GitHub actions to automate tasks as a result of an event that happens. Events can be a push operation to a GitHub repository or a merge request to the repository. In GitHub, you can define one to many workflows that organize the actions you want to execute.

Workflows contain one-to-many jobs. Each job is made up of one-to-many items. An item can be a GitHub action or a shell script, or a shell command.

There are many predefined GitHub actions that we can use in our workflows. But it is also possible to author a custom action. The latter is outside of the scope of this book, though.

1. Checkout to a new feature branch called `actions-first-steps`:

```
git checkout -b actions-first-steps
```

2. Create a new folder **`.github`** in the project folder
3. Inside this **`.github`** folder, create a subfolder **`workflows`**
4. Inside the workflows folder, create a YAML file called **`hello-world-workflow.yml`** containing the instructions for our first sample workflow:



Figure 9.3: Actions tab of the repository in GitHub

A simple hello world style GitHub workflow

   a. Line 1 defines the name for the workflow, here **`hello-world-workflow`**
   b. Line 2 defines which event will trigger this workflow. In this case, any **push** to the repository on GitHub will trigger it
   c. Line 3 starts the section where you define your jobs. In this simple sample, we have only one job called "**`say-hello`**" (line 4)
   d. Line 4 defines the name of the job – in this case **`say-hello`**
   e. Line 5 indicates on which type of agent the job shall run. Options are Linux, Windows, or MacOS. Here, we have chosen Linux, and to be more precise, we have chosen an Ubuntu Linux agent
   f. Line 6 starts the list of steps this (first) job is made of
   g. Line 7 defines our (only) step in this job. It is a shell script-type step. We are simply going to output "Hello world!" with the help of the **`echo`** command.

5. Save the preceding file and then commit and push it to GitHub:

```
git add .
git push -u origin actions-first-steps
```

6. Go back to GitHub and navigate to the **Actions** tab. You should see the following:

2. Create a new folder **`.github`** in the project folder
3. Inside this **`.github`** folder, create a subfolder **`workflows`**
4. Inside the workflows folder, create a YAML file called **`hello-world-workflow.yml`** containing the instructions for our first sample workflow:



Figure 9.3: Actions tab of the repository in GitHub

A simple hello world style GitHub workflow

   a. Line 1 defines the name for the workflow, here **`hello-world-workflow`**
   b. Line 2 defines which event will trigger this workflow. In this case, any **push** to the repository on GitHub will trigger it
   c. Line 3 starts the section where you define your jobs. In this simple sample, we have only one job called "**`say-hello`**" (line 4)
   d. Line 4 defines the name of the job – in this case **`say-hello`**
   e. Line 5 indicates on which type of agent the job shall run. Options are Linux, Windows, or MacOS. Here, we have chosen Linux, and to be more precise, we have chosen an Ubuntu Linux agent
   f. Line 6 starts the list of steps this (first) job is made of
   g. Line 7 defines our (only) step in this job. It is a shell script-type step. We are simply going to output "Hello world!" with the help of the **`echo`** command.

5. Save the preceding file and then commit and push it to GitHub:

```
git add .
git push -u origin actions-first-steps
```

6. Go back to GitHub and navigate to the **Actions** tab. You should see the following:

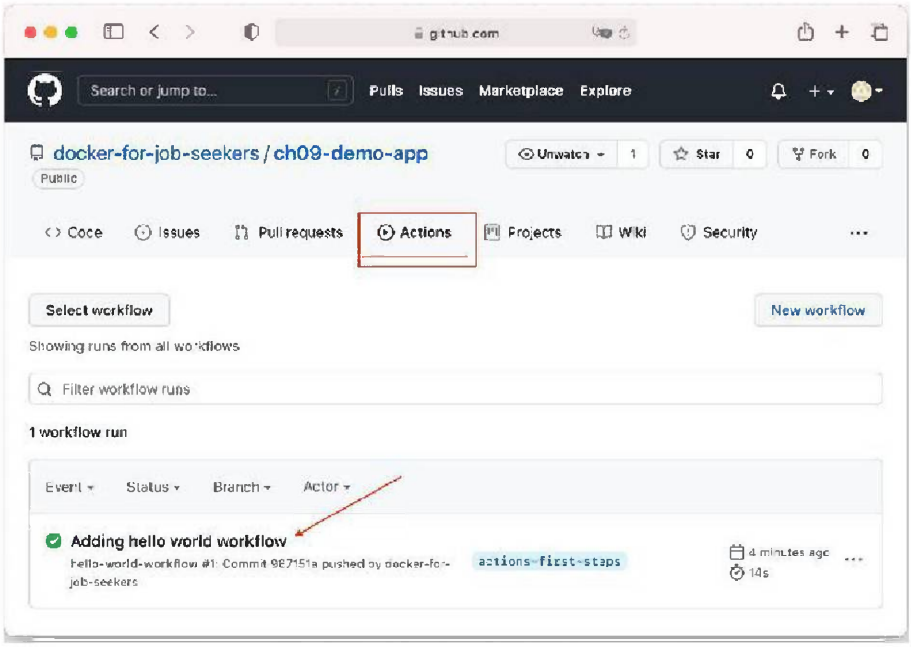


Figure 9.4: Actions tab of the repository in GitHub

Notice the item called "**Adding hello world workflow**" at the bottom of the view. This is our workflow that apparently has successfully run, indicated by the green check mark next to it.

7. Click on this item to see the details:

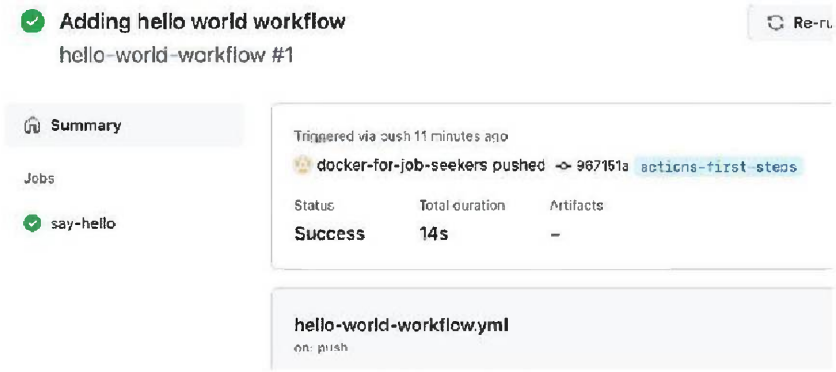


Figure 9.5: Details of the hello world GitHub workflow

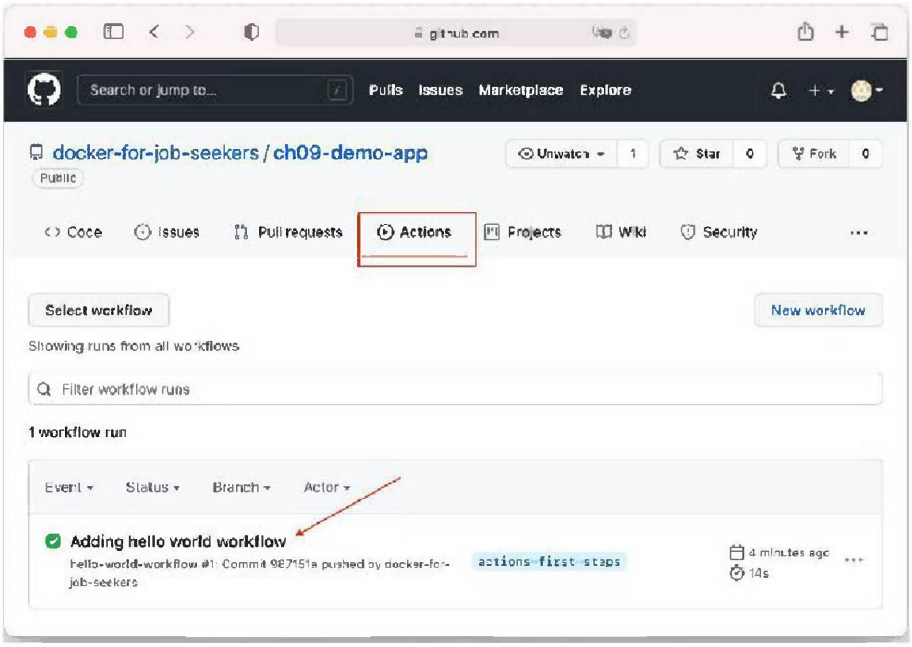


Figure 9.4: Actions tab of the repository in GitHub

Notice the item called **"Adding hello world workflow"** at the bottom of the view. This is our workflow that apparently has successfully run, indicated by the green check mark next to it.

7. Click on this item to see the details:



Figure 9.5: Details of the hello world GitHub workflow

Here, we can see that the workflow had a single job called **say-hello**, which was executed successfully and indicated once again with the green checkmark.

8. Click on the say-hello job on the left of the view to see the details of the job:

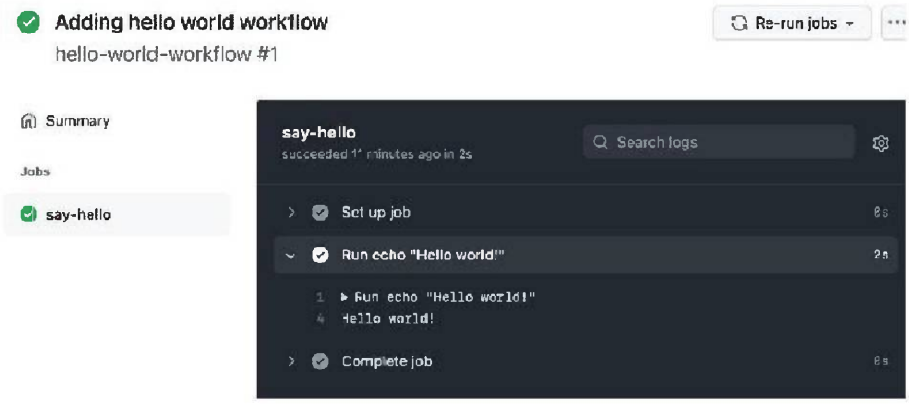


Figure 9.6: Details of the say-hello job in the GitHub workflow

In the preceding screenshot, we can see the details of the job down to the individual statement. Please note our run statement, which outputs "Hello world!". This view is specifically useful if you have to debug a failed job since you will find all the error messages in this output.

9. Since adding the workflow was successful, we can create a pull request (PR) of our feature branch against the **main** branch, review and merge it.

    a. In GitHub, navigate to the **Code** tab of the repository. You should see a yellow popup, similar to the one shown as follows, inviting you to create a PR.

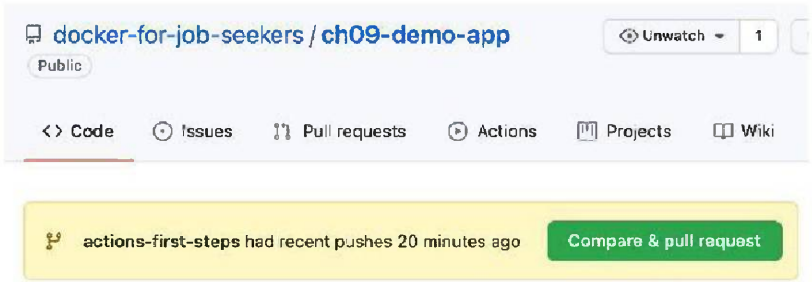


Figure 9.7: Create a pull request in GitHub

Here, we can see that the workflow had a single job called **say-hello**, which was executed successfully and indicated once again with the green checkmark.

8. Click on the say-hello job on the left of the view to see the details of the job:

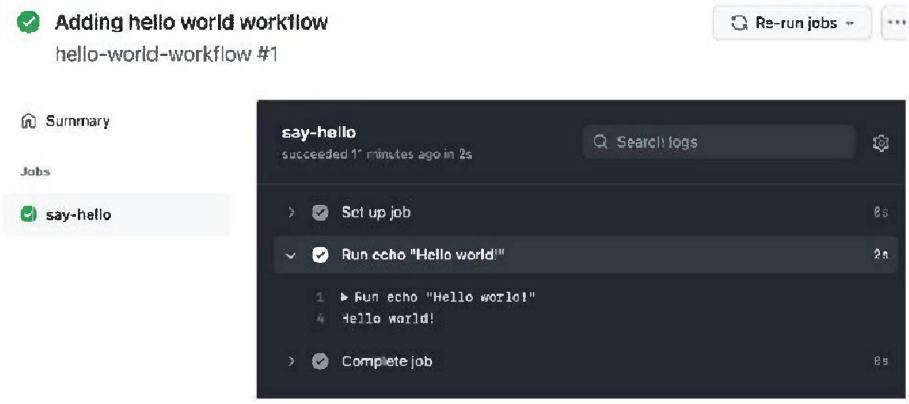


Figure 9.6: Details of the say-hello job in the GitHub workflow

In the preceding screenshot, we can see the details of the job down to the individual statement. Please note our run statement, which outputs "Hello world!". This view is specifically useful if you have to debug a failed job since you will find all the error messages in this output.

9. Since adding the workflow was successful, we can create a pull request (PR) of our feature branch against the `main` branch, review and merge it.

    a. In GitHub, navigate to the **Code** tab of the repository. You should see a yellow popup, similar to the one shown as follows, inviting you to create a PR.

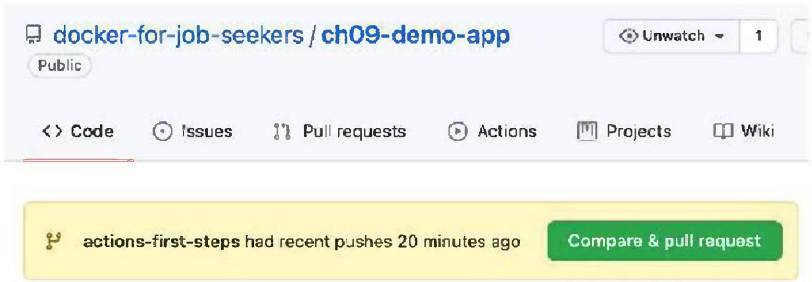


Figure 9.7: Create a pull request in GitHub

b. Click the green **Compare & pull request** button. In the **Open a pull request**, view and leave the suggested title for the PR and optionally add a comment. Then, click the green **Create pull request** button.

c. Review the PR, and if everything is OK, click the green **Merge pull request** button:

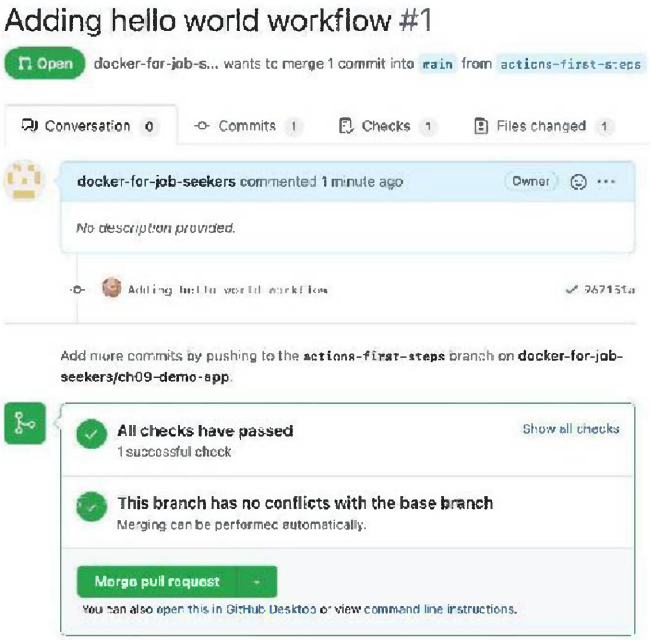


Figure 9.8: Pull request details in GitHub

10. Once the PR is merged, we want to update our local code as follows:

```
git checkout main
git pull
```

Great, we have just built and run our first workflow on GitHub. Now we are ready to add a workflow that is a bit more sophisticated. Let us try to build a Docker image for the `web` service of our demo application.

Adding a build Docker image workflow

Now that we have learned what a GitHub workflow is, it is time to create the workflow that will build our Docker image.

b. Click the green **Compare & pull request** button. In the **Open a pull request**, view and leave the suggested title for the PR and optionally add a comment. Then, click the green **Create pull request** button.

c. Review the PR, and if everything is OK, click the green **Merge pull request** button:

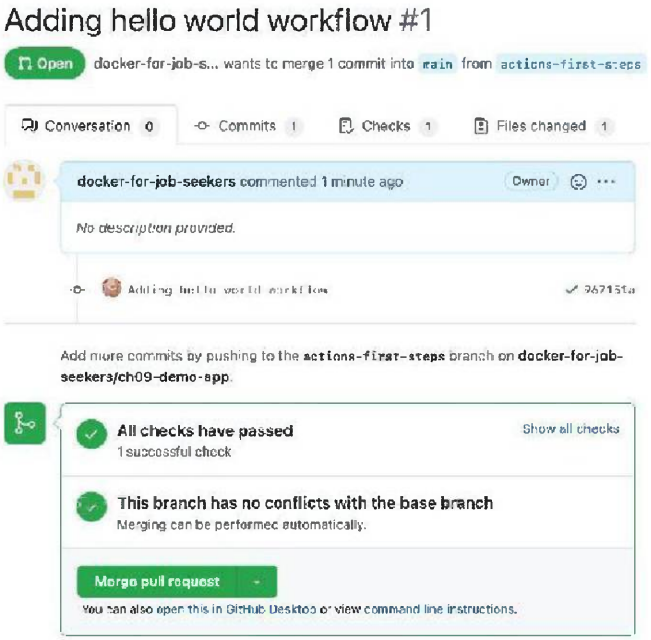


Figure 9.8: Pull request details in GitHub

10. Once the PR is merged, we want to update our local code as follows:

```
git checkout main
git pull
```

Great, we have just built and run our first workflow on GitHub. Now we are ready to add a workflow that is a bit more sophisticated. Let us try to build a Docker image for the `web` service of our demo application.

Adding a build Docker image workflow

Now that we have learned what a GitHub workflow is, it is time to create the workflow that will build our Docker image.

1. Create a new Git branch for this new feature called **build-image-workflow**:

   ```
   git checkout -b build-image-workflow
   ```

2. Add a file called **build-web-image-workflow.yml** in the **.github/workflows** folder and add the following content to it:

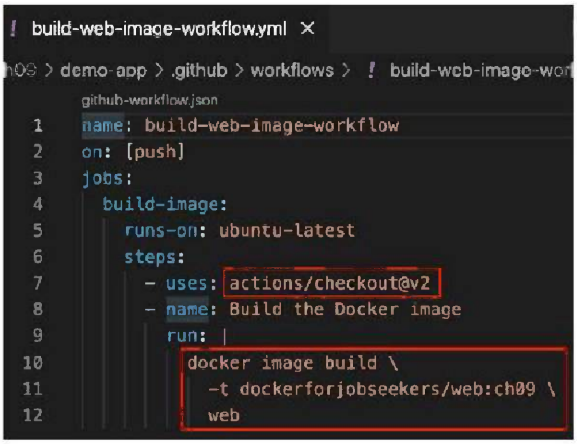


Figure 9.9: Build Docker image workflow

   a. Lines 1 to 6 are similar to what we learned in the first workflow.

   b. Line 7: defines the first step in the job, where we use a predefined action called **actions/checkout@v2**, which is checking out the repository to the runner such as that the subsequent actions or commands can work with this code.

   c. Lines 8 to 12 define the second step in the job with the command to build a Docker image from the sources in the **web** subfolder of the demo app. This is just the normal Docker command, as you would use it when manually building the image. Please note on Line 9 how we use the | symbol to allow for a multi-line script on the subsequent lines (Lines 10 to 12).

3. Commit this new code to the branch and push everything to GitHub:

   ```
   git add .
   git commit -m "Adding build image workflow"
   git push
   ```

4. Back in GitHub, navigate to the Actions tab and make sure that the new workflow is executed without errors:

1. Create a new Git branch for this new feature called **build-image-workflow**:

   ```
   git checkout -b build-image-workflow
   ```

2. Add a file called **build-web-image-workflow.yml** in the **.github/workflows** folder and add the following content to it:

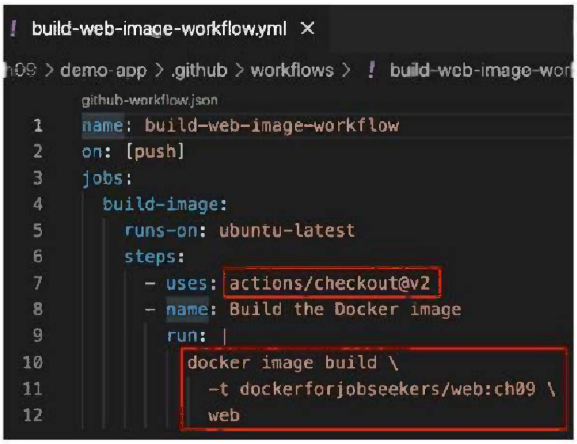


Figure 9.9: Build Docker image workflow

   a. Lines 1 to 6 are similar to what we learned in the first workflow.

   b. Line 7: defines the first step in the job, where we use a predefined action called **actions/checkout@v2**, which is checking out the repository to the runner such as that the subsequent actions or commands can work with this code.

   c. Lines 8 to 12 define the second step in the job with the command to build a Docker image from the sources in the **web** subfolder of the demo app. This is just the normal Docker command, as you would use it when manually building the image. Please note on Line 9 how we use the | symbol to allow for a multi-line script on the subsequent lines (Lines 10 to 12).

3. Commit this new code to the branch and push everything to GitHub:

   ```
   git add .
   git commit -m "Adding build image workflow"
   git push
   ```

4. Back in GitHub, navigate to the Actions tab and make sure that the new workflow is executed without errors:

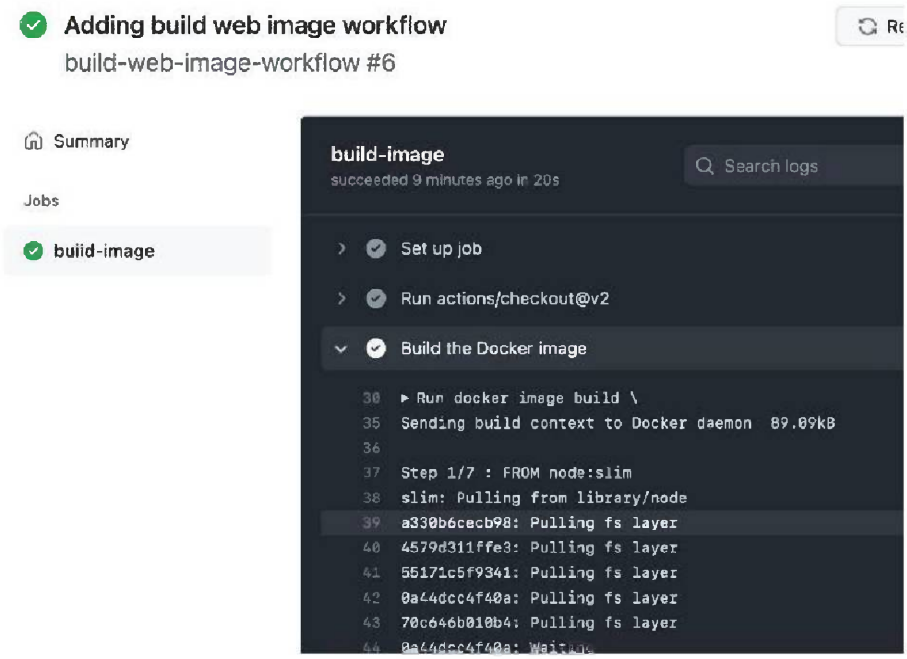


Figure 9.10: Log of the build Docker image action

If, for some reason, the build job fails, then please use this log to help you debug and find the problem.

Please note that our hello world workflow is also executed each time we push some code.

That was super easy, right? Now each time you push some code changes, the workflow on GitHub will try to build the image. With this, you have a quick feedback loop telling you if your code change has broken something. If so, it is easy to find the root cause (assuming that the last push did contain only a limited amount of code changes).

Creating a complete CI pipeline

Typically, a complete CI pipeline includes more than just building the final artifact to deploy. Some of the steps included are as follows:

- Linting the code
- Running automated tests

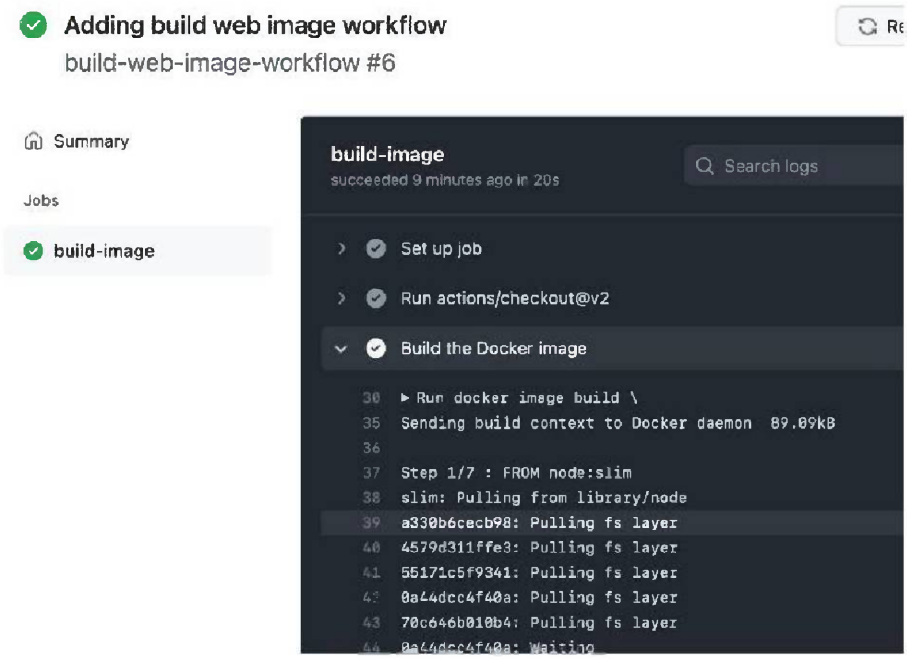


Figure 9.10: Log of the build Docker image action

If, for some reason, the build job fails, then please use this log to help you debug and find the problem.

Please note that our hello world workflow is also executed each time we push some code.

That was super easy, right? Now each time you push some code changes, the workflow on GitHub will try to build the image. With this, you have a quick feedback loop telling you if your code change has broken something. If so, it is easy to find the root cause (assuming that the last push did contain only a limited amount of code changes).

Creating a complete CI pipeline

Typically, a complete CI pipeline includes more than just building the final artifact to deploy. Some of the steps included are as follows:

- Linting the code
- Running automated tests

- Building the deployment artifacts, such as a Docker image

Let us take the `api-dotnet` project of our `demo-app` to demonstrate how we can run automated tests as part of the pipeline.

1. Open a new terminal window and navigate to the `ch09/demo-app` folder:

   ```
   cd ~/src/ch09/demo-app
   ```

2. Create a new feature branch in Git:

   ```
   git checkout -b add-tests-to-api-dotnet
   ```

3. Use the dotnet CLI to create a new .NET test project (called `api-dotnet-tests`) for the `api-dotnet` project:

   ```
   dotnet new xunit -o api-dotnet-tests
   ```

4. Add the `api-dotnet` project as a reference to the new tests project:

   ```
   dotnet add api-dotnet-tests reference api-dotnet
   ```

5. Watch and run the tests every time a change is made to the project:

   ```
   dotnet watch test --project api-dotnet-tests
   ```

6. Open the demo-app in VS Code:

   ```
   code .
   ```

7. Locate the `api-dotnet-tests` project, and in it, the class `UnitTest1` has been automatically added to the project. Change its code and add a test like the following:

```
using System;
using Xunit;
using api_dotnet.Models;

namespace api_dotnet_tests
{
    0 references
    public class UnitTest1
    {
        [Fact]
        0 references
        public void Test1()
        {
            var order = new Order{
                Id = 1,
                OrderDate = new DateTime(2021,10,1),
                Status = OrderStatus.Draft
            };
            var text = order.ToString();
            Assert.Equal("Id=1, OrderDate=10/01/2021 00:00:00, Status=Draft", text);
        }
    }
}
```

Figure 9.11: First unit test for the order class

- Building the deployment artifacts, such as a Docker image

Let us take the `api-dotnet` project of our `demo-app` to demonstrate how we can run automated tests as part of the pipeline.

1. Open a new terminal window and navigate to the `ch09/demo-app` folder:

   ```
   cd ~/src/ch09/demo-app
   ```

2. Create a new feature branch in Git:

   ```
   git checkout -b add-tests-to-api-dotnet
   ```

3. Use the dotnet CLI to create a new .NET test project (called `api-dotnet-tests`) for the `api-dotnet` project:

   ```
   dotnet new xunit -o api-dotnet-tests
   ```

4. Add the `api-dotnet` project as a reference to the new tests project:

   ```
   dotnet add api-dotnet-tests reference api-dotnet
   ```

5. Watch and run the tests every time a change is made to the project:

   ```
   dotnet watch test --project api-dotnet-tests
   ```

6. Open the demo-app in VS Code:

   ```
   code .
   ```

7. Locate the `api-dotnet-tests` project, and in it, the class `UnitTest1` has been automatically added to the project. Change its code and add a test like the following:

```
using System;
using Xunit;
using api_dotnet.Models;

namespace api_dotnet_tests
{
    0 references
    public class UnitTest1
    {
        [Fact]
        0 references
        public void Test1()
        {
            var order = new Order{
                Id = 1,
                OrderDate = new DateTime(2021,10,1),
                Status = OrderStatus.Draft
            };
            var text = order.ToString();
            Assert.Equal("Id=1, OrderDate=10/01/2021 00:00:00, Status=Draft", text);
        }
    }
}
```

Figure 9.11: First unit test for the order class

8. Save your changes and note in the terminal how the project is compiled, and the tests are run automatically. The test should pass with a message similar to this

   ```
   A total of 1 test files matched the specified pattern.
   Passed! - Failed: 0, Passed: 1, Skipped: 0, Total: 1, Duration:<1 ms
   ```

   Note: depending on what locale you are using on your laptop, the preceding test may fail because it assumes `en-US` locale with a date format of `MM/DD/YYYY`. If you have a different local, you can change the code in the order class to output the date with the correct format.

9. **Optional**: Try to add more tests. Make sure they all pass.

Great, we are now ready to define the CI pipeline on GitHub using GitHub actions. The pipeline to build will consist of a single job with the following three steps:

1. Checkout source code to the build agent
2. Run automated tests
3. Build the final Docker image of the .NET API

Let us do that. We start by building a container in which we can run the tests.

1. Create a folder `.docker` in the `demo-app` solution folder
2. Inside the preceding `.docker` folder, create a subfolder `api-dotnet-tests`
3. Inside this subfolder, create a file called `Dockerfile` and add the following content:

```
1  FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
2  WORKDIR /src
3  COPY ["./api-dotnet/api-dotnet.csproj", "./api-dotnet/"]
4  COPY ["./api-dotnet-tests/api-dotnet-tests.csproj", "./api-dotnet-tests/"]
5  RUN dotnet restore api-dotnet-tests
6  COPY ["./api-dotnet", "./api-dotnet"]
7  COPY ["./api-dotnet-tests", "./api-dotnet-tests"]
8  CMD dotnet test ./api-dotnet-tests
```

Figure 9.12: Dockerfile to create an environment to run automated tests

4. Build the image:

   ```
   docker image build \
       -f ./.docker/api-dotnet-tests/Dockerfile \
       -t api-dotnet-tests .
   ```

8. Save your changes and note in the terminal how the project is compiled, and the tests are run automatically. The test should pass with a message similar to this

   ```
   A total of 1 test files matched the specified pattern.
   Passed! - Failed: 0, Passed: 1, Skipped: 0, Total: 1, Duration:<1 ms
   ```

   Note: depending on what locale you are using on your laptop, the preceding test may fail because it assumes `en-US` locale with a date format of `MM/DD/YYYY`. If you have a different local, you can change the code in the order class to output the date with the correct format.

9. **Optional**: Try to add more tests. Make sure they all pass.

Great, we are now ready to define the CI pipeline on GitHub using GitHub actions. The pipeline to build will consist of a single job with the following three steps:

1. Checkout source code to the build agent
2. Run automated tests
3. Build the final Docker image of the .NET API

Let us do that. We start by building a container in which we can run the tests.

1. Create a folder `.docker` in the `demo-app` solution folder
2. Inside the preceding `.docker` folder, create a subfolder `api-dotnet-tests`
3. Inside this subfolder, create a file called `Dockerfile` and add the following content:

```
1  FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
2  WORKDIR /src
3  COPY ["./api-dotnet/api-dotnet.csproj", "./api-dotnet/"]
4  COPY ["./api-dotnet-tests/api-dotnet-tests.csproj", "./api-dotnet-tests/"]
5  RUN dotnet restore api-dotnet-tests
6  COPY ["./api-dotnet", "./api-dotnet"]
7  COPY ["./api-dotnet-tests", "./api-dotnet-tests"]
8  CMD dotnet test ./api-dotnet-tests
```

Figure 9.12: Dockerfile to create an environment to run automated tests

4. Build the image:

   ```
   docker image build \
       -f ./.docker/api-dotnet-tests/Dockerfile \
       -t api-dotnet-tests .
   ```

This command uses the **Dockerfile** in the **.docker/api-dotnet-tests** folder and calls the resulting image **api-dotnet-tests**. The build command uses the **demo-app** folder (from inside which we run it) as the build context (because of the "**.**" at the end of the command).

5. Run the tests inside the container with the following:

   ```
   docker container run api-dotnet-tests
   ```

   the output should look like the following:

```
gabriel@Solaris demo-app % docker container run api-dotnet-tests
  Determining projects to restore...
  Restored /src/api-dotnet/api-dotnet.csproj (in 179 ms).
  Restored /src/api-dotnet-tests/api-dotnet-tests.csproj (in 228 ms).
  api-dotnet -> /src/api-dotnet/bin/Debug/net5.0/api-dotnet.dll
  api-dotnet-tests -> /src/api-dotnet-tests/bin/Debug/net5.0/api-dotnet-tests.dll
Test run for /src/api-dotnet-tests/bin/Debug/net5.0/api-dotnet-tests.dll (.NETCoreApp,Version=v5.0)
Microsoft (R) Test Execution Command Line Tool Version 16.11.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed!  - Failed:     0, Passed:     1, Skipped:     0, Total:     1, Duration: < 1 ms - /src/api-dotnet-t
ests/bin/Debug/net5.0/api-dotnet-tests.dll (net5.0)
```

Figure 9.13: Running automated tests for api-dotnet inside a container

   Notice the test passed successfully!

Now, let us use all this when building the next workflow for GitHub:

1. Add a new file called **ci-api-dotnet.yml** to the **.github/workflows** folder and add this content to it:

```yaml
name: build-api-dotnet-image-workflow
on: [push]
jobs:
  build-image:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Build Docker image for automated tests of .NET API
        run: |
          docker image build \
            -f ./.docker/api-dotnet-tests/Dockerfile \
            -t api-dotnet-tests .

      - name: Run automated tests in api-dotnet-tests image
        run: docker container run api-dotnet-tests

      - name: Build the Docker image for .NET API
        run: |
          docker image build \
            -t dockerforjobseekers/api-dotnet:ch09 \
            api-dotnet
```

Figure 9.14: Complete CI workflow for the .NET API

This command uses the **Dockerfile** in the **.docker/api-dotnet-tests** folder and calls the resulting image **api-dotnet-tests**. The build command uses the **demo-app** folder (from inside which we run it) as the build context (because of the "**.**" at the end of the command).

5. Run the tests inside the container with the following:

   ```
   docker container run api-dotnet-tests
   ```

   the output should look like the following:

```
gabriel@Solaris demo-app % docker container run api-dotnet-tests
  Determining projects to restore...
  Restored /src/api-dotnet/api-dotnet.csproj (in 179 ms).
  Restored /src/api-dotnet-tests/api-dotnet-tests.csproj (in 228 ms).
  api-dotnet -> /src/api-dotnet/bin/Debug/net5.0/api-dotnet.dll
  api-dotnet-tests -> /src/api-dotnet-tests/bin/Debug/net5.0/api-dotnet-tests.dll
Test run for /src/api-dotnet-tests/bin/Debug/net5.0/api-dotnet-tests.dll (.NETCoreApp,Version=v5.0)
Microsoft (R) Test Execution Command Line Tool Version 16.11.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed!  - Failed:     0, Passed:     1, Skipped:     0, Total:     1, Duration: < 1 ms - /src/api-dotnet-t
ests/bin/Debug/net5.0/api-dotnet-tests.dll (net5.0)
```

Figure 9.13: Running automated tests for api-dotnet inside a container

   Notice the test passed successfully!

Now, let us use all this when building the next workflow for GitHub:

1. Add a new file called **ci-api-dotnet.yml** to the **.github/workflows** folder and add this content to it:

```yaml
name: build-api-dotnet-image-workflow
on: [push]
jobs:
  build-image:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Build Docker image for automated tests of .NET API
        run: |
          docker image build \
            -f ./.docker/api-dotnet-tests/Dockerfile \
            -t api-dotnet-tests .

      - name: Run automated tests in api-dotnet-tests image
        run: docker container run api-dotnet-tests

      - name: Build the Docker image for .NET API
        run: |
          docker image build \
            -t dockerforjobseekers/api-dotnet:ch09 \
            api-dotnet
```

Figure 9.14: Complete CI workflow for the .NET API

a. **Line 1:** Define the name of the new workflow.

b. **Line 2:** Workflow runs on every push to the code repository.

c. **Line 6:** This workflow has 4 steps that are all named.

d. **Lines 7 and 8:** First step checks out the code to the build agent of GitHub.

e. **Lines 10 to 14:** We build a container containing the code for the .NET API and also the code of the test project containing all automated tests. This container is called **`api-dotnet-tests`**.

f. **Lines 16 and 17:** We run the automated tests inside the container **`api-dotnet-tests`** we just built.

g. **Lines 19 to 23:** After the automated tests have passed, we build the final Docker image for the .NET API.

2. Commit the changes to Git and push the new feature branch to GitHub:

```
git add .
git commit -m "Adding CI workflow for .NET API"
git push -u origin add-tests-to-api-dotnet
```

3. Head over to GitHub and select the **Actions** tab. Note that you should now see a list of three workflows, including the one that you just pushed:

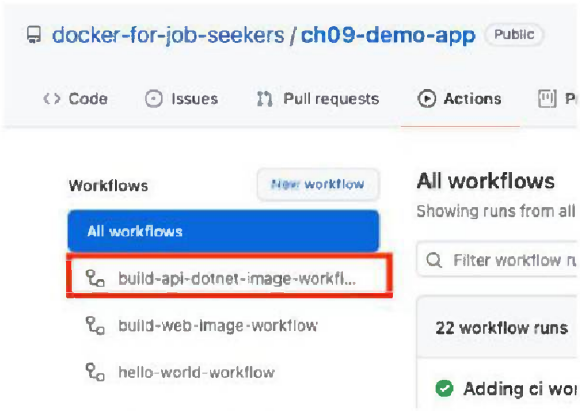


Figure 9.15: List of workflows on GitHub

a. **Line 1:** Define the name of the new workflow.

b. **Line 2:** Workflow runs on every push to the code repository.

c. **Line 6:** This workflow has 4 steps that are all named.

d. **Lines 7 and 8:** First step checks out the code to the build agent of GitHub.

e. **Lines 10 to 14:** We build a container containing the code for the .NET API and also the code of the test project containing all automated tests. This container is called **`api-dotnet-tests`**.

f. **Lines 16 and 17:** We run the automated tests inside the container **`api-dotnet-tests`** we just built.

g. **Lines 19 to 23:** After the automated tests have passed, we build the final Docker image for the .NET API.

2. Commit the changes to Git and push the new feature branch to GitHub:

```
git add .
git commit -m "Adding CI workflow for .NET API"
git push -u origin add-tests-to-api-dotnet
```

3. Head over to GitHub and select the **Actions** tab. Note that you should now see a list of three workflows, including the one that you just pushed:

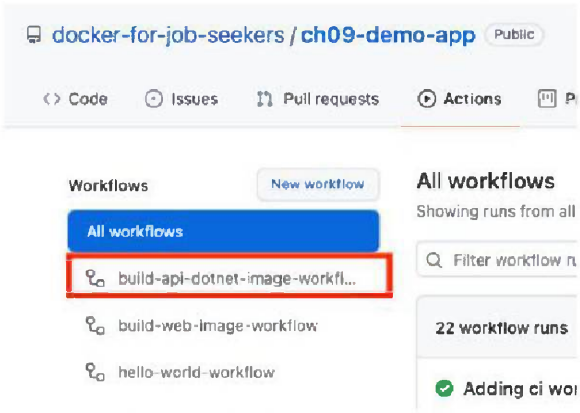


Figure 9.15: List of workflows on GitHub

4. Click on the new CI workflow and drill down further until you are able to observe that it has multiple steps. They all should succeed eventually signaled by the green check mark:

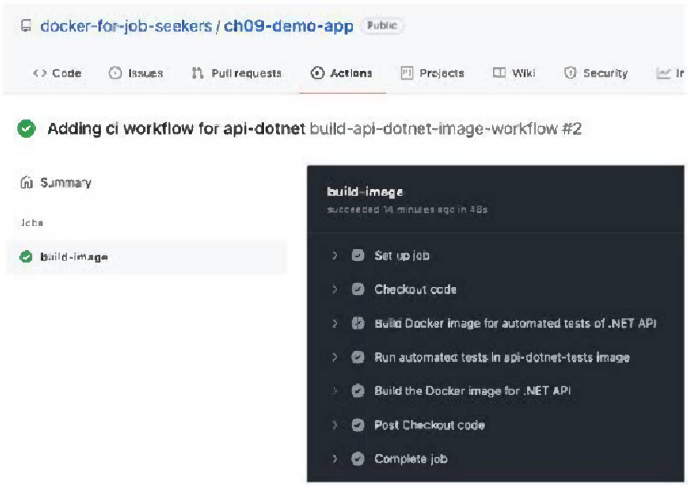


Figure 9.16: Details of the CI workflow for the .NET API

5. Once the workflow has succeeded, we can create a PR on GitHub for our feature branch against the `main` branch and then review the PR and merge it.

6. After having merged the PR, you will want to switch to the main branch on your local machine and pull the latest changes:

   ```
   git checkout main
   git pull
   ```

   With that, the new feature is available locally too in the `main` branch.

Now that we have learned how to build a complete CI pipeline using GitHub actions, let us have a look at an automation server that can do the same for us and much more. Let us learn how to use Jenkins.

Using Jenkins to setup a build pipeline

Before we discuss SaaS offerings, we want to get to know a product that you can use and manage yourself to automate your build pipelines. This product is called Jenkins. Jenkins (**https://www.jenkins.io**) is the leading open-source automation server. There are hundreds of plugins available to address any need when building up a fully automated CI/CD pipeline. An alternative product that is equally powerful but requires a license is TeamCity (**https://www.jetbrains.com/teamcity**).

4. Click on the new CI workflow and drill down further until you are able to observe that it has multiple steps. They all should succeed eventually signaled by the green check mark:

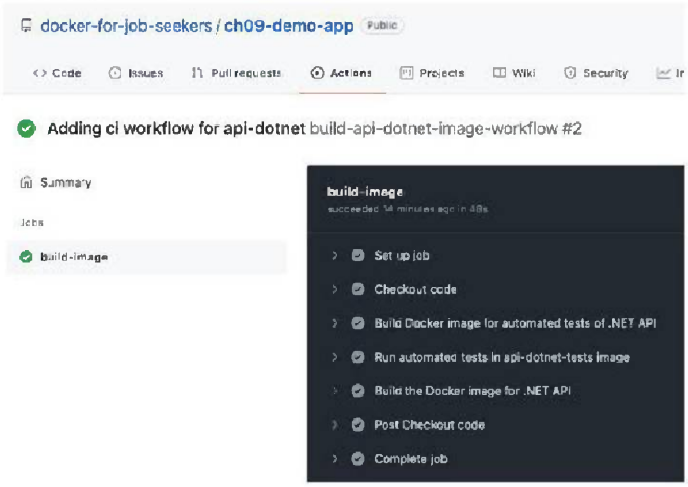


Figure 9.16: Details of the CI workflow for the .NET API

5. Once the workflow has succeeded, we can create a PR on GitHub for our feature branch against the **main** branch and then review the PR and merge it.

6. After having merged the PR, you will want to switch to the main branch on your local machine and pull the latest changes:

```
git checkout main
git pull
```

With that, the new feature is available locally too in the **main** branch.

Now that we have learned how to build a complete CI pipeline using GitHub actions, let us have a look at an automation server that can do the same for us and much more. Let us learn how to use Jenkins.

Using Jenkins to setup a build pipeline

Before we discuss SaaS offerings, we want to get to know a product that you can use and manage yourself to automate your build pipelines. This product is called Jenkins. Jenkins (**https://www.jenkins.io**) is the leading open-source automation server. There are hundreds of plugins available to address any need when building up a fully automated CI/CD pipeline. An alternative product that is equally powerful but requires a license is TeamCity (**https://www.jetbrains.com/teamcity**).

If you are more interested in a powerful product for deployment, then you may consider Octopus Deploy (**https://octopus.com**).

A complete CI/CD pipeline

In this section, we will try to implement a CI/CD pipeline using Jenkins that looks similar to the one depicted in following figure:

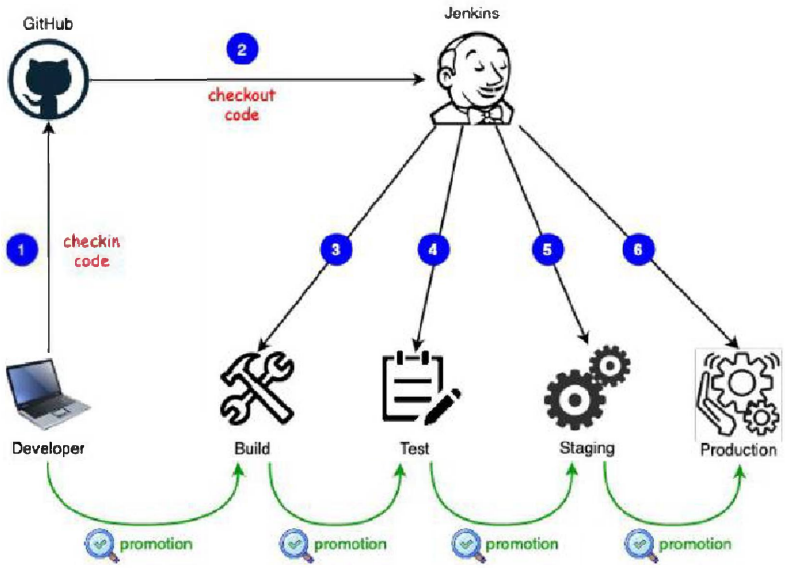


Figure 9.17: A simple CI/CD pipeline using Jenkins

In the preceding image, we have the following steps marked explicitly:

1. A developer writes code on their local laptop and checks it in with GitHub.
2. GitHub notifies Jenkins via WebHook, and Jenkins will checkout the code to a Jenkins agent.
3. Jenkins will run a build on the agent. If the build is successful, Jenkins moves to the next step.
4. Jenkins will run all automated tests. Again, if this succeeds, it will move to the next step.
5. Jenkins deploys the artifact to staging.
6. Finally, an authorized user may instruct Jenkins to deploy the artifacts to the production system.

If you are more interested in a powerful product for deployment, then you may consider Octopus Deploy (**https://octopus.com**).

A complete CI/CD pipeline

In this section, we will try to implement a CI/CD pipeline using Jenkins that looks similar to the one depicted in following figure:

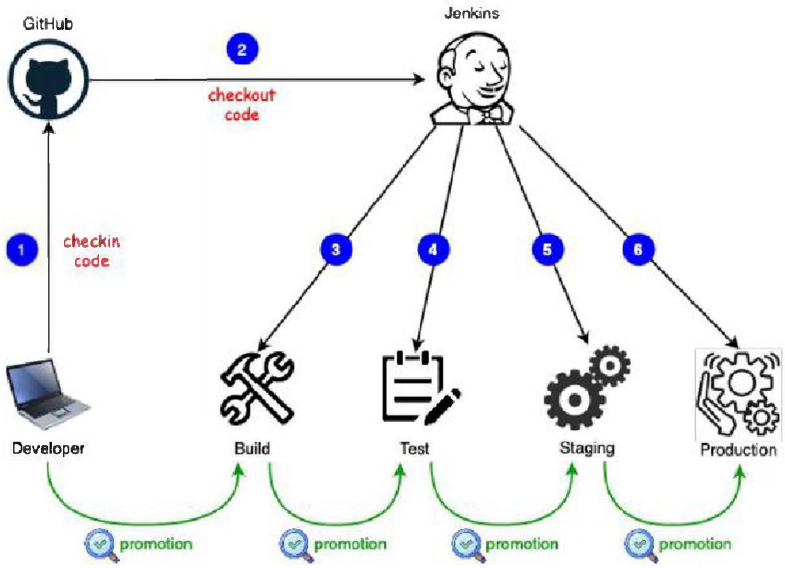


Figure 9.17: A simple CI/CD pipeline using Jenkins

In the preceding image, we have the following steps marked explicitly:

1. A developer writes code on their local laptop and checks it in with GitHub.
2. GitHub notifies Jenkins via WebHook, and Jenkins will checkout the code to a Jenkins agent.
3. Jenkins will run a build on the agent. If the build is successful, Jenkins moves to the next step.
4. Jenkins will run all automated tests. Again, if this succeeds, it will move to the next step.
5. Jenkins deploys the artifact to staging.
6. Finally, an authorized user may instruct Jenkins to deploy the artifacts to the production system.

We have the green arrows symbolizing that the next step is only executed ever by Jenkins if the previous step was successful. This is then called a promotion.

Jenkins uses a file called `Jenkinsfile` to configure pipelines. A simple example with `Build`, `Test`, `Deploy to Staging`, and `Deploy to Production` stages would look like the following:

Jenkinsfile

```
pipeline {
    agent any
    options {
        skipStagesAfterUnstable()
    }
    stages {
        stage('Build') {
            steps {
                echo 'Building'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing'
            }
        }
        stage('Deploy to Staging') {
            steps {
                echo 'Deploying to Staging'
            }
        }
        stage('Deploy to Production') {
            steps {
                echo 'Deploying to Production'
            }
        }
    }
}
```

Figure 9.18: Simple Jenkinsfile with four stages

The preceding file does nothing much other than output a message during each stage. It is still useful to showcase how a pipeline works.

Let us now look into how one can build such a pipeline.

Creating a CI pipeline

Now that we know how a complete and fully automated CI/CD pipeline may look like, let us use Jenkins to build a pure build pipeline, or differently said, let us build the CI part of the CI/CD pipeline using Jenkins. When using Jenkins, the central document is the `Jenkinsfile`, which will contain the definition of the pipeline with its multiple stages.

1. Open a new terminal window and navigate to the source code folder for *this chapter*:

   ```
   cd ~/src/ch09
   ```

We have the green arrows symbolizing that the next step is only executed ever by Jenkins if the previous step was successful. This is then called a promotion.

Jenkins uses a file called `Jenkinsfile` to configure pipelines. A simple example with `Build`, `Test`, `Deploy to Staging`, and `Deploy to Production` stages would look like the following:

Jenkinsfile

```
pipeline {
    agent any
    options {
        skipStagesAfterUnstable()
    }
    stages {
        stage('Build') {
            steps {
                echo 'Building'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing'
            }
        }
        stage('Deploy to Staging') {
            steps {
                echo 'Deploying to Staging'
            }
        }
        stage('Deploy to Production') {
            steps {
                echo 'Deploying to Production'
            }
        }
    }
}
```

Figure 9.18: Simple Jenkinsfile with four stages

The preceding file does nothing much other than output a message during each stage. It is still useful to showcase how a pipeline works.

Let us now look into how one can build such a pipeline.

Creating a CI pipeline

Now that we know how a complete and fully automated CI/CD pipeline may look like, let us use Jenkins to build a pure build pipeline, or differently said, let us build the CI part of the CI/CD pipeline using Jenkins. When using Jenkins, the central document is the `Jenkinsfile`, which will contain the definition of the pipeline with its multiple stages.

1. Open a new terminal window and navigate to the source code folder for *this chapter*:

   ```
   cd ~/src/ch09
   ```

2. Create a sub-folder named Jenkins-pipeline and navigate to it:

```
mkdir jenkins-pipeline && cd jenkins-pipeline
```

3. Next, we run an instance of Jenkins inside a container such as that we can access it via its Web UI through a mapped port:

```
$_ scripts.sh
1  docker run --rm -d \
2      --name jenkins \
3      -p 8080:8080 \
4      -p 50000:50000 \
5      -v jenkins_home:/var/jenkins_home \
6      -v /var/run/docker.sock:/var/run/docker.sock \
7      -v "$HOME":/home \
8      dockerforjobseekers/jenkins
```

Figure 9.19: Running Jenkins in a container

Note that:

a. Line 2: we are naming the container **jenkins**

b. Line 3: we are mapping the Jenkins port 8080 to the equivalent host port

c. Line 5: we are defining a volume (**jenkins_home**) where Jenkins will store its data locally so it is not lost if a restart is required.

d. Line 6: we are mounting the Docker socket into the container (**-v /var/run/docker.sock:/var/run/docker.sock**).

This allows Jenkins to execute Docker commands during the build process that we are going to establish.

e. We are mounting the local folder containing our source code into the **/home** folder inside the Jenkins container.

f. We are using the image called **dockerforjobseekers/jenkins** as our Jenkins image. This image is derived from the **jenkins/jenkins** image and in addition to that, has the Docker CLI installed. You can find the Dockerfile used to build this image in the folder **.docker/jenkins**.

4. Upon the first start, Jenkins generates an initial password that we can find by using the following command:

```
docker container exec jenkins \
    cat /var/jenkins_home/secrets/initialAdminPassword
```

2. Create a sub-folder named Jenkins-pipeline and navigate to it:

```
mkdir jenkins-pipeline && cd jenkins-pipeline
```

3. Next, we run an instance of Jenkins inside a container such as that we can access it via its Web UI through a mapped port:

```
$_ scripts.sh
1  docker run --rm -d \
2      --name jenkins \
3      -p 8080:8080 \
4      -p 50000:50000 \
5      -v jenkins_home:/var/jenkins_home \
6      -v /var/run/docker.sock:/var/run/docker.sock \
7      -v "$HOME":/home \
8      dockerforjobseekers/jenkins
```

Figure 9.19: Running Jenkins in a container

Note that:

a. Line 2: we are naming the container **jenkins**

b. Line 3: we are mapping the Jenkins port 8080 to the equivalent host port

c. Line 5: we are defining a volume (**jenkins_home**) where Jenkins will store its data locally so it is not lost if a restart is required.

d. Line 6: we are mounting the Docker socket into the container (**-v /var/run/docker.sock:/var/run/docker.sock**).

This allows Jenkins to execute Docker commands during the build process that we are going to establish.

e. We are mounting the local folder containing our source code into the **/home** folder inside the Jenkins container.

f. We are using the image called **dockerforjobseekers/jenkins** as our Jenkins image. This image is derived from the **jenkins/jenkins** image and in addition to that, has the Docker CLI installed. You can find the Dockerfile used to build this image in the folder **.docker/jenkins**.

4. Upon the first start, Jenkins generates an initial password that we can find by using the following command:

```
docker container exec jenkins \
    cat /var/jenkins_home/secrets/initialAdminPassword
```

The output should look similar to the following:

```
7f449293de5443a2bbcb0918c8558689
```

Please note down this value, as you will be using it in the next step when logging in to Jenkins

5. Open a new browser tab and navigate to `http://localhost:8080` to access the Web UI of Jenkins. You will be asked to unlock Jenkins by entering the password that you accessed in the previous step.

6. Next, choose **Install suggested plugins** to have Jenkins automatically install the most useful plugins for you. Plugins that will be installed include GitHub integration, an e-mail extension, and Maven and Gradle integration, among others.

   Alternatively, and if you are familiar with Jenkins, you can also manually install the plugins that are needed for this exercise.

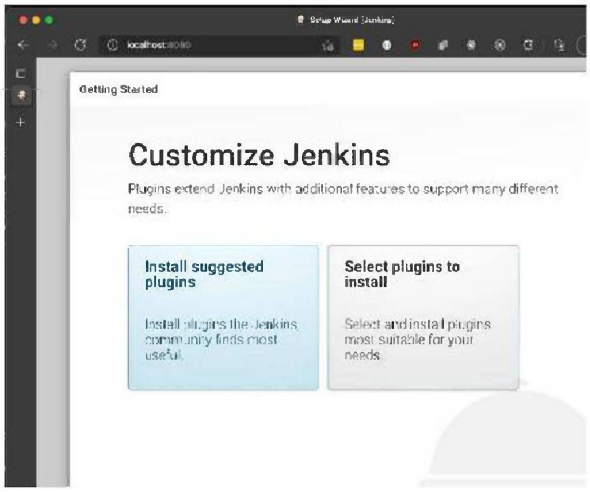


Figure 9.20: Customizing Jenkins upon first start

7. Once the plugins are installed, you are asked to create an Admin account. Please do so now. Select the following values:

   Username: Admin

   Password: Top-Secret

   E-mail: admin@example.com

8. When asked, leave the Jenkins URL at its default value `http://localhost:8080`.

The output should look similar to the following:

```
7f449293de5443a2bbcb0918c8558689
```

Please note down this value, as you will be using it in the next step when logging in to Jenkins

5. Open a new browser tab and navigate to **`http://localhost:8080`** to access the Web UI of Jenkins. You will be asked to unlock Jenkins by entering the password that you accessed in the previous step.

6. Next, choose **Install suggested plugins** to have Jenkins automatically install the most useful plugins for you. Plugins that will be installed include GitHub integration, an e-mail extension, and Maven and Gradle integration, among others.

   Alternatively, and if you are familiar with Jenkins, you can also manually install the plugins that are needed for this exercise.

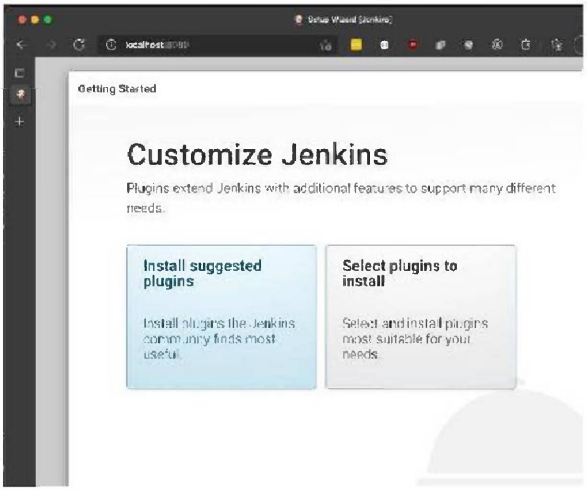


Figure 9.20: Customizing Jenkins upon first start

7. Once the plugins are installed, you are asked to create an Admin account. Please do so now. Select the following values:

   Username: Admin

   Password: Top-Secret

   E-mail: admin@example.com

8. When asked, leave the Jenkins URL at its default value **`http://localhost:8080`**.

9. Log out and log in again. This is what you should see the following:

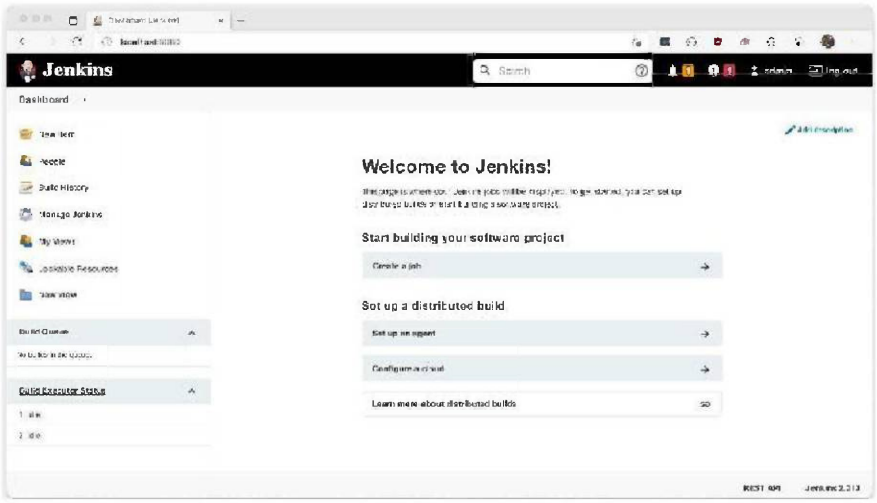


Figure 9.21: Welcome view of Jenkins

10. In addition to the plugins already installed, we have to install the **Docker** and the **Docker Pipeline** plugins. We need them later in this chapter when we create a Docker container-based agent. To install these plugins, select **Manage Jenkins** on the Dashboard and then **Manage Plugins**.

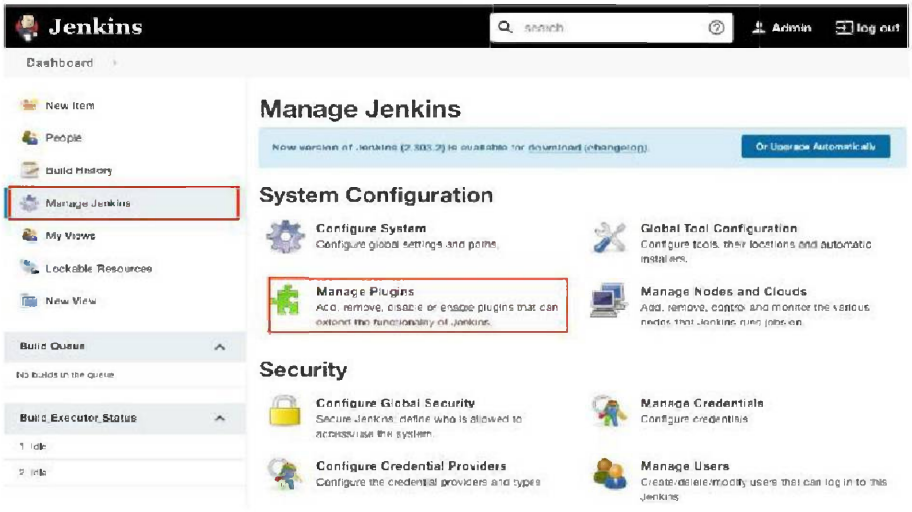


Figure 9.22: Installing additional plugins to Jenkins

9. Log out and log in again. This is what you should see the following:

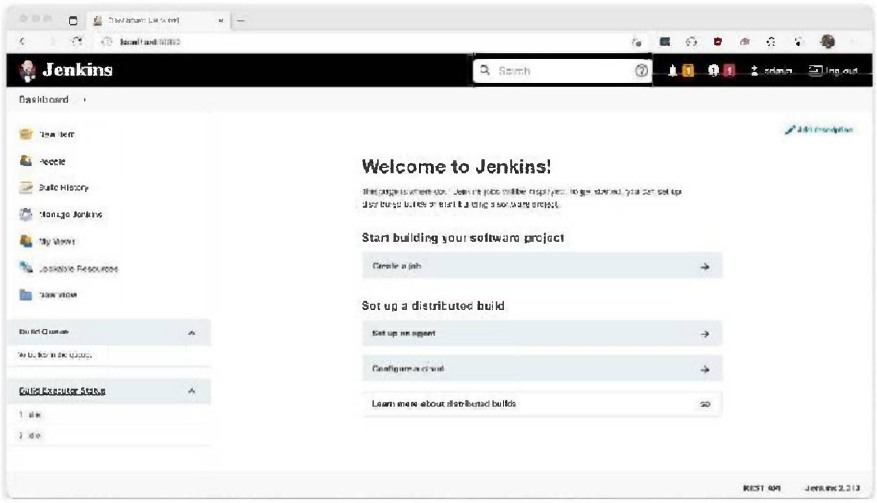


Figure 9.21: Welcome view of Jenkins

10. In addition to the plugins already installed, we have to install the **Docker** and the **Docker Pipeline** plugins. We need them later in this chapter when we create a Docker container-based agent. To install these plugins, select **Manage Jenkins** on the Dashboard and then **Manage Plugins**.

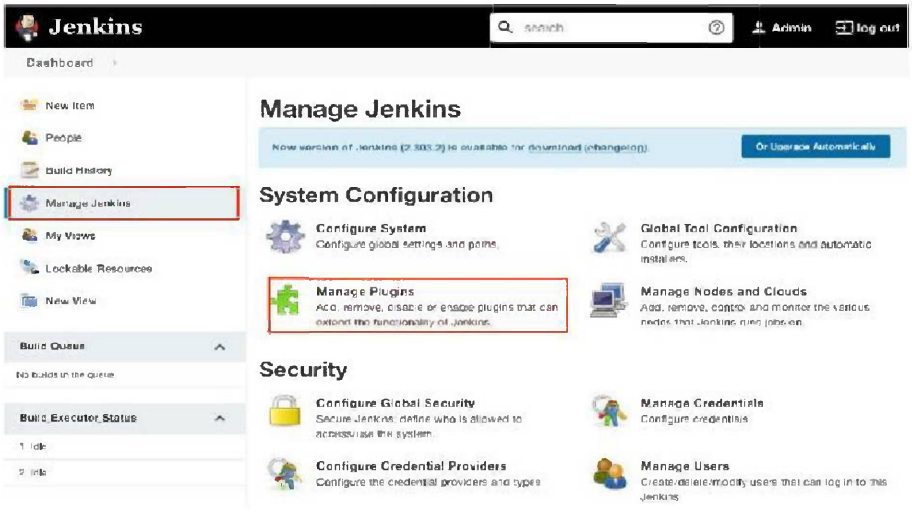


Figure 9.22: Installing additional plugins to Jenkins

11. Locate and install those two plugins **Docker** and the **Docker Pipeline** mentioned before.

12. Click **Create a job** and in the view displayed, enter **`sample-pipeline`** as the item name and select the **Pipeline** type, and finally, click the **OK** Button.

13. On the next view displayed, select the Pipeline tab and enter the simple Jenkins script from the preceding section into the **Script** textbox and then click the **Save** button:

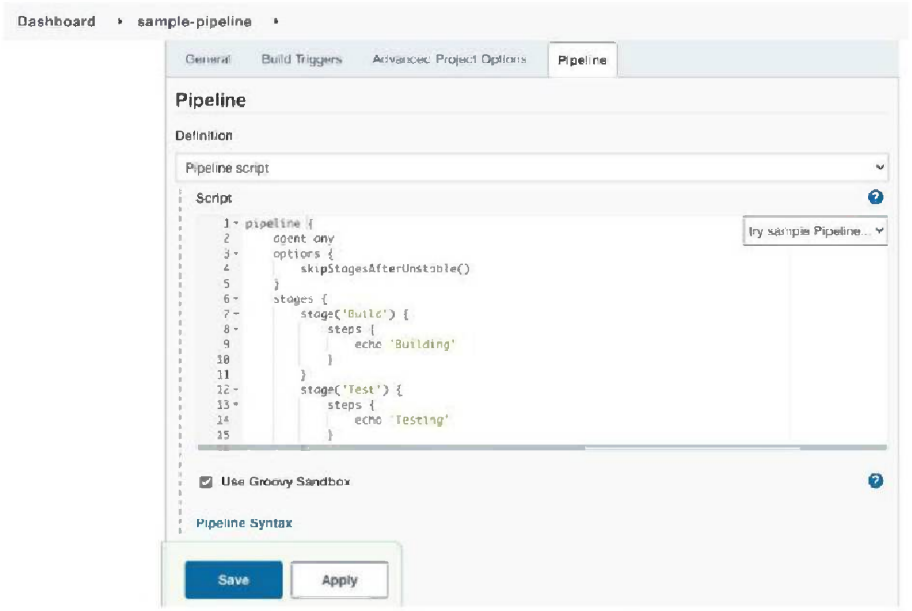


Figure 9.23: Defining the first sample pipeline in Jenkins

14. Click **Build Now** in the left sidebar and observe how the pipeline consisting of the four stages is executed:

11. Locate and install those two plugins **Docker** and the **Docker Pipeline** mentioned before.

12. Click **Create a job** and in the view displayed, enter **`sample-pipeline`** as the item name and select the **Pipeline** type, and finally, click the **OK** Button.

13. On the next view displayed, select the Pipeline tab and enter the simple Jenkins script from the preceding section into the **Script** textbox and then click the **Save** button:

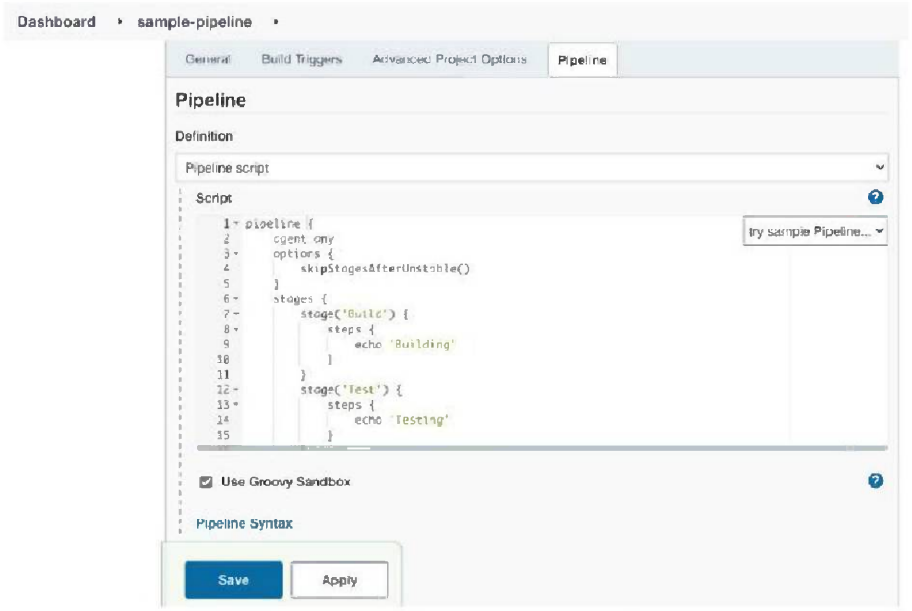


Figure 9.23: Defining the first sample pipeline in Jenkins

14. Click **Build Now** in the left sidebar and observe how the pipeline consisting of the four stages is executed:

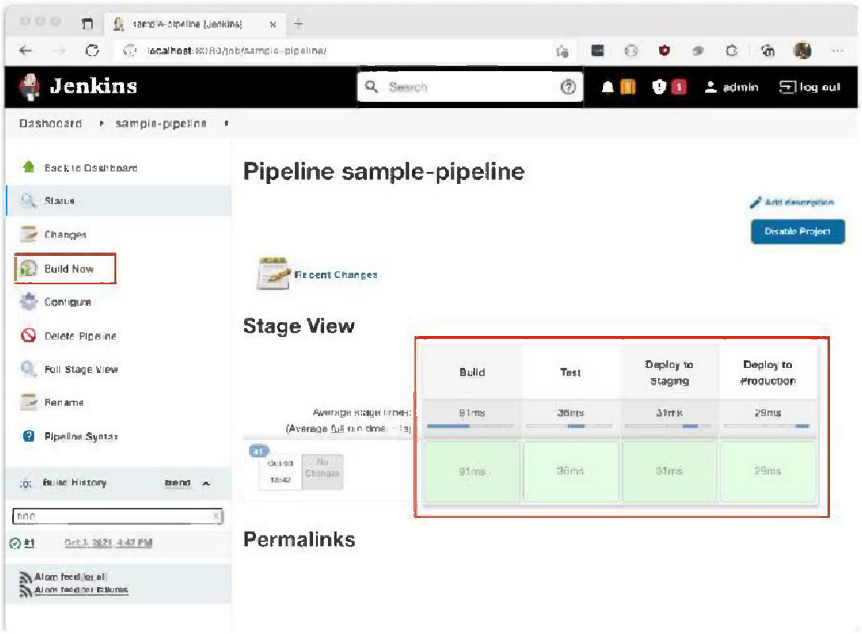


***Figure 9.24:** Building the sample-pipeline in Jenkins*

15. Hover the mouse pointer over any of the four stages and click the **Logs** button in the small popup appearing to see the output of each stage.

We have successfully created and run a simple pipeline consisting of four stages in Jenkins. It is now time that we try to create a pipeline that does something meaningful such as building and testing our `api-dotnet` project.

A complete CI pipeline in Jenkins

To create a pipeline that:

- Checks out code from GitHub,
- Builds a Docker image with the `web` project,
- Runs the tests in the built test image and finally
- Build a Docker image for the api-dotnet project

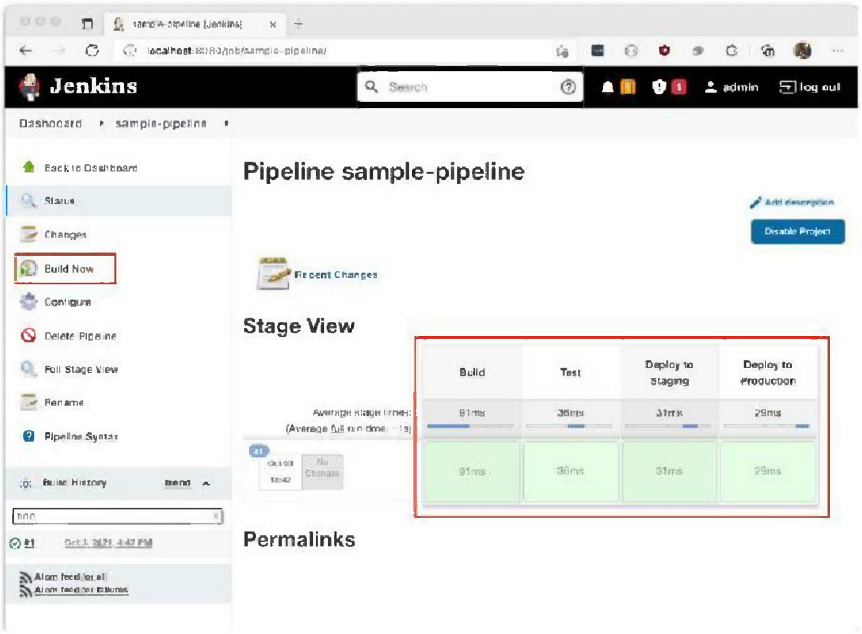


***Figure 9.24:** Building the sample-pipeline in Jenkins*

15. Hover the mouse pointer over any of the four stages and click the **Logs** button in the small popup appearing to see the output of each stage.

We have successfully created and run a simple pipeline consisting of four stages in Jenkins. It is now time that we try to create a pipeline that does something meaningful such as building and testing our `api-dotnet` project.

A complete CI pipeline in Jenkins

To create a pipeline that:

- Checks out code from GitHub,
- Builds a Docker image with the `web` project,
- Runs the tests in the built test image and finally
- Build a Docker image for the api-dotnet project

Follow the following step-by-step guide:

1. The first step is to commit and push the **`Jenkinsfile`** to the GitHub repository:

```
git add .
git commit -m "Adding Jenkinsfile"
git push
```

2. Then, we need to connect the GitHub repository of our demo application with Jenkins. For this, select **Configure** on the left sidebar of the simple-pipeline project in Jenkins. Then, select the Pipeline tab and modify the configuration to look like the following:

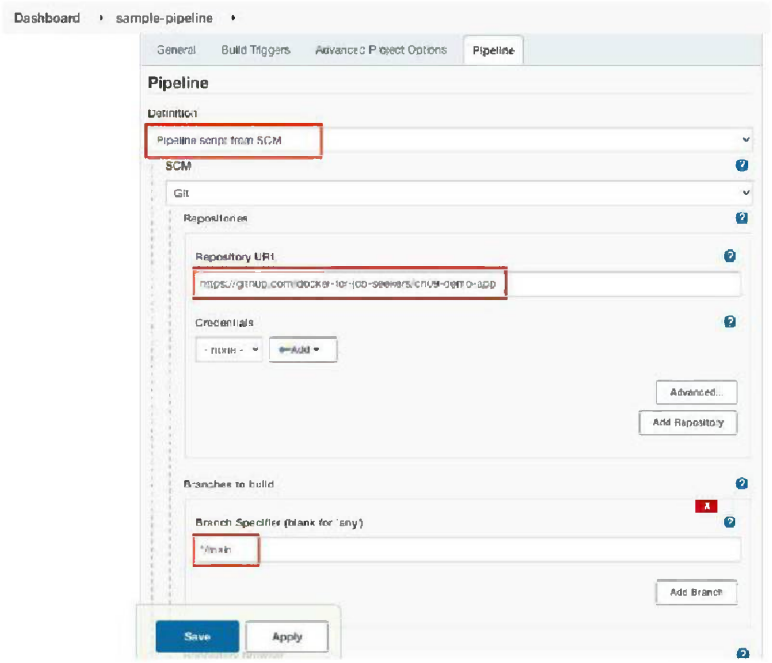


Figure 9.25: Connecting the pipeline in Jenkins with the GitHub repository

   a. Make sure to select **Pipeline script from SCM** in the drop-down.

   b. Enter the correct repository URL where your **`demo-app`** is located in GitHub.

   c. Make sure under Branches to build to replace **`*/master`** with **`*/main`** since our main branch is called **`main`**. Optionally you can leave this textbox blank.

Follow the following step-by-step guide:

1. The first step is to commit and push the **`Jenkinsfile`** to the GitHub repository:

```
git add .
git commit -m "Adding Jenkinsfile"
git push
```

2. Then, we need to connect the GitHub repository of our demo application with Jenkins. For this, select **Configure** on the left sidebar of the simple-pipeline project in Jenkins. Then, select the Pipeline tab and modify the configuration to look like the following:

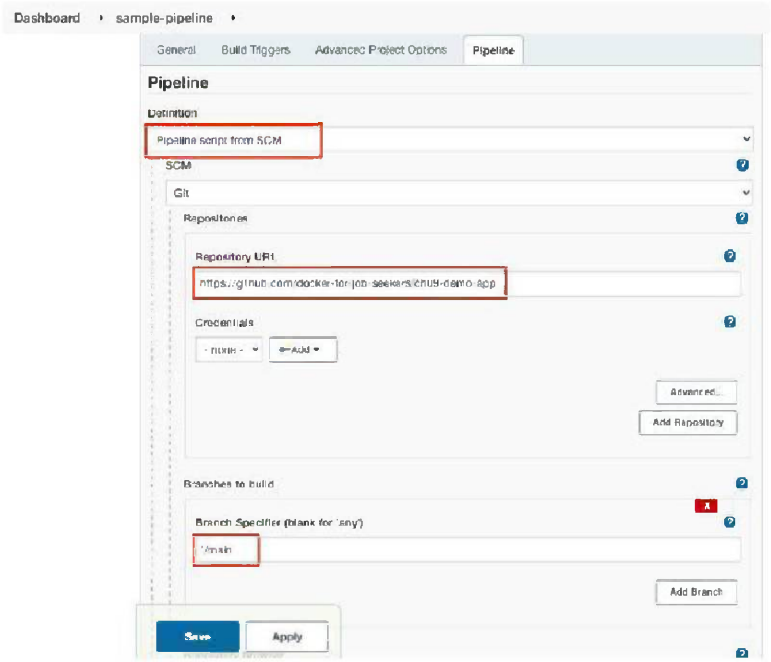


Figure 9.25: Connecting the pipeline in Jenkins with the GitHub repository

a. Make sure to select **Pipeline script from SCM** in the drop-down.

b. Enter the correct repository URL where your **`demo-app`** is located in GitHub.

c. Make sure under Branches to build to replace **`*/master`** with **`*/main`** since our main branch is called **`main`**. Optionally you can leave this textbox blank.

3. Click the **Save** button when done.

4. Now, select the **Build now** command in the left sidebar again and observe how the pipeline is built. Notice that the pipeline has now an additional stage called **Declarative: Checkout SCM**, which is actually checking out the source code from GitHub to the local build agent of Jenkins running in the container. Jenkins is now using the **Jenkinsfile** found in the root of the checked-out code. This is the default location where the Jenkins file is expected.

5. When you click on **Logs** in the **Checkout** stage, you should get a popup with content similar to the following:

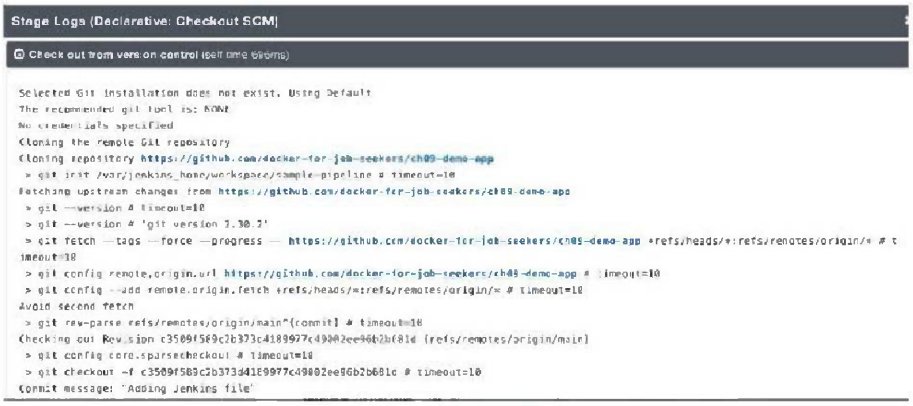


Figure 9.26: Log of the GitHub checkout performed by Jenkins

Now, let us extend the web project a bit and add a test to it.

1. In the terminal window, navigate to the **web** folder of the **demo-app**

2. Install the two npm modules **jasmine** and **request**:

```
npm i jasmine request --save-dev
```

3. In VS Code, open the **package.json** file and modify the scripts section to look like the following:

```
"scripts": {
  "start": "node index.js",
  "test-init": "node_modules/jasmine/bin/jasmine.js init",
  "test": "node_modules/jasmine/bin/jasmine.js"
},
```

3. Click the **Save** button when done.

4. Now, select the **Build now** command in the left sidebar again and observe how the pipeline is built. Notice that the pipeline has now an additional stage called **Declarative: Checkout SCM**, which is actually checking out the source code from GitHub to the local build agent of Jenkins running in the container. Jenkins is now using the **Jenkinsfile** found in the root of the checked-out code. This is the default location where the Jenkins file is expected.

5. When you click on **Logs** in the **Checkout** stage, you should get a popup with content similar to the following:

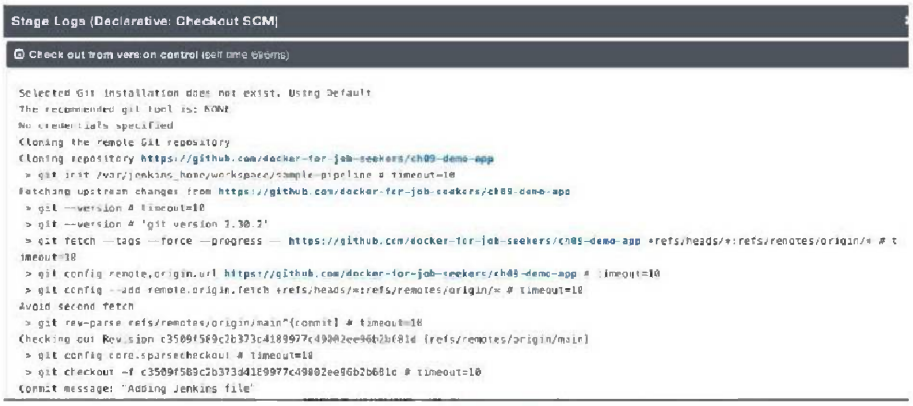


Figure 9.26: Log of the GitHub checkout performed by Jenkins

Now, let us extend the web project a bit and add a test to it.

1. In the terminal window, navigate to the **web** folder of the **demo-app**

2. Install the two npm modules **jasmine** and **request**:

```
npm i jasmine request --save-dev
```

3. In VS Code, open the **package.json** file and modify the scripts section to look like the following:

```
"scripts": {
  "start": "node index.js",
  "test-init": "node_modules/jasmine/bin/jasmine.js init",
  "test": "node_modules/jasmine/bin/jasmine.js"
},
```

4. Open the file **index.js** and at the end of the file, add a **/health** endpoint and also modify the startup of express as follows:

```
37  app.get('/health', (req, res) => {
38    res.sendStatus(200);
39  })
40
41  var server = app.listen(port, () => {
42    console.log(`Demo Web application listening at http://localhost:${port}`)
43  })
44
45  module.exports = server;
```

Figure 9.27: Modified index.js with added /health endpoint and exported server

Note how we export the **server** variable in Line 45.

5. Add a folder **spec** to the project.
6. To the **spec** folder, add a file **index-spec.js** containing one test:

```
JS index-spec.js U  X
web > spec > JS index-spec.js > describe('Checking /health endpoint') callback
1   const request = require('request');
2   const baseUrl = 'http://localhost:3000';
3
4   describe('Checking /health endpoint', () => {
5       var server;
6       beforeAll(() => {
7           server = require('../index.js');
8       })
9       afterAll(() => {
10          server.close();
11      })
12      it('return status code 200', (done) => {
13          request.get(`${baseUrl}/health`, (err, response, body) => {
14              expect(response.statusCode).toBe(200);
15              done();
16          })
17      })
18  })
```

Figure 9.28: Testing /health endpoint in Web project

4. Open the file **index.js** and at the end of the file, add a **/health** endpoint and also modify the startup of express as follows:

```
37  app.get('/health', (req, res) => {
38    res.sendStatus(200);
39  })
40
41  var server = app.listen(port, () => {
42    console.log(`Demo Web application listening at http://localhost:${port}`)
43  })
44
45  module.exports = server;
```

Figure 9.27: Modified index.js with added /health endpoint and exported server

Note how we export the **server** variable in Line 45.

5. Add a folder **spec** to the project.
6. To the **spec** folder, add a file **index-spec.js** containing one test:

```
JS index-spec.js U  X
web > spec > JS index-spec.js > describe('Checking /health endpoint') callback
1   const request = require('request');
2   const baseUrl = 'http://localhost:3000';
3
4   describe('Checking /health endpoint', () => {
5       var server;
6       beforeAll(() => {
7           server = require('../index.js');
8       })
9       afterAll(() => {
10          server.close();
11      })
12      it('return status code 200', (done) => {
13          request.get(`${baseUrl}/health`, (err, response, body) => {
14              expect(response.statusCode).toBe(200);
15              done();
16          })
17      })
18  })
```

Figure 9.28: Testing /health endpoint in Web project

This test uses the Jasmine syntax and the `request` module to execute the REST call to the `/health` endpoint.

7. Prepare the web project for testing with the following:

   ```
   npm test-init
   ```

   This will add a folder `/support` with a file `jasmine.json` into the `spec` folder. This file is used to configure Jasmine.

8. And now, try to run the tests:

   ```
   npm test
   ```

   The output should look similar to the following:

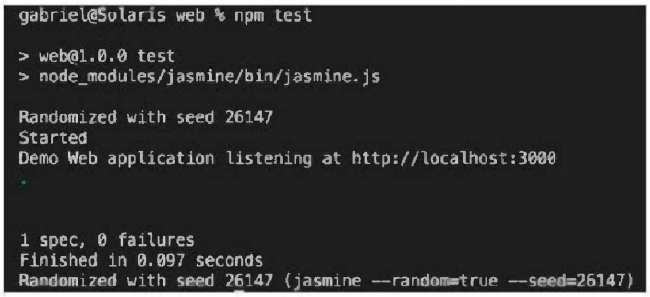


Figure 9.29: Output of Jasmine test run

   As you can see, we have one passing test.

9. Commit your changes to Git and push them to GitHub:

   ```
   git add .
   git commit -m "Adding tests to the web project"
   git push -u origin jenkins-ci-for-web
   ```

This test uses the Jasmine syntax and the `request` module to execute the REST call to the `/health` endpoint.

7. Prepare the web project for testing with the following:

   ```
   npm test-init
   ```

   This will add a folder `/support` with a file `jasmine.json` into the `spec` folder. This file is used to configure Jasmine.

8. And now, try to run the tests:

   ```
   npm test
   ```

   The output should look similar to the following:

```
gabriel@Solaris web % npm test

> web@1.0.0 test
> node_modules/jasmine/bin/jasmine.js

Randomized with seed 26147
Started
Demo Web application listening at http://localhost:3000
.


1 spec, 0 failures
Finished in 0.097 seconds
Randomized with seed 26147 (jasmine --random=true --seed=26147)
```

Figure 9.29: Output of Jasmine test run

   As you can see, we have one passing test.

9. Commit your changes to Git and push them to GitHub:

   ```
   git add .
   git commit -m "Adding tests to the web project"
   git push -u origin jenkins-ci-for-web
   ```

Modifying the pipeline for Jenkins:

1. Modify the content of the **Jenkinsfile** in the **demo-app** folder to look like the following:

```
1  pipeline {
2      environment {
3          registry = "dockerforjobseekers/web"
4          DOCKER_PWD = credentials('docker-login-pwd')
5      }
6      agent {
7          docker {
8              image 'dockerforjobseekers/node-docker'
9              args '-p 3000:3000'
10             args '-w /app'
11             args '-v /var/run/docker.sock:/var/run/docker.sock'
12         }
13     }
14     options {
15         skipStagesAfterUnstable()
16     }
17     stages {
18         stage('Build') {
19             steps {
20                 echo('Building...')
21                 sh 'npm install'
22             }
23         }
24         stage('Test') {
25             steps {
26                 echo 'Testing...'
27                 sh 'npm test'
28             }
29         }
30         stage('Build & push Docker image') {
31             steps {
32                 echo 'Building & pushing Docker image...'
33                 sh 'docker image build -t $registry:$BUILD_NUMBER .'
34                 sh 'docker login -u dockerforjobseekers -p $DOCKER_PWD'
35                 sh 'docker image push $registry:$BUILD_NUMBER'
36                 sh 'docker image rm $registry:$BUILD_NUMBER'
37             }
38         }
39     }
40 }
```

Figure 9.30: Modified Jenkins pipeline

Let us explain in detail the different parts of this file:

a. **Lines 2–5:** We define some environment variables that can be used later in the pipeline. The first variable, **registry**, denotes the name of the target Docker image for the web project. Please make sure to replace **dockerforjobseekers** with your own Docker username. The second variable, **docker_PWD**, will contain the password to login to the Docker account (**dockerforjobseekers** in our case and your username in your case). Since we do not want to hard-code a

Modifying the pipeline for Jenkins:

1. Modify the content of the **`Jenkinsfile`** in the **`demo-app`** folder to look like the following:

```
1  pipeline {
2      environment {
3          registry = "dockerforjobseekers/web"
4          DOCKER_PWD = credentials('docker-login-pwd')
5      }
6      agent {
7          docker {
8              image 'dockerforjobseekers/node-docker'
9              args '-p 3000:3000'
10             args '-w /app'
11             args '-v /var/run/docker.sock:/var/run/docker.sock'
12         }
13     }
14     options {
15         skipStagesAfterUnstable()
16     }
17     stages {
18         stage('Build') {
19             steps {
20                 echo('Building...')
21                 sh 'npm install'
22             }
23         }
24         stage('Test') {
25             steps {
26                 echo 'Testing...'
27                 sh 'npm test'
28             }
29         }
30         stage('Build & push Docker image') {
31             steps {
32                 echo 'Building & pushing Docker image...'
33                 sh 'docker image build -t $registry:$BUILD_NUMBER .'
34                 sh 'docker login -u dockerforjobseekers -p $DOCKER_PWD'
35                 sh 'docker image push $registry:$BUILD_NUMBER'
36                 sh 'docker image rm $registry:$BUILD_NUMBER'
37             }
38         }
39     }
40 }
```

Figure 9.30: Modified Jenkins pipeline

Let us explain in detail the different parts of this file:

a. **Lines 2–5:** We define some environment variables that can be used later in the pipeline. The first variable, **`registry`**, denotes the name of the target Docker image for the web project. Please make sure to replace **`dockerforjobseekers`** with your own Docker username. The second variable, **`docker_PWD`**, will contain the password to login to the Docker account (**`dockerforjobseekers`** in our case and your username in your case). Since we do not want to hard-code a

password here in this file, we use the credentials function to retrieve the value from Jenkin's secrets store, where I will place it under the name `docker-login-pwd`.

b. **Lines 6–13**: Here, we define the agent on which Jenkins will run the pipeline. We are using a special Docker image called `dockerforjobseekers/node-docker` that we built. It is based on `node:16-alpine` and has additionally `curl` and the Docker CLI installed. In Line 11, we mount the Docker socket into the container such as that we can run Docker command from withing the container.

c. **Optional:** If you want to build your own image, here are the necessary commands to prepare the image:

```
docker pull node:16-alpine
docker run --name node -it node:16-alpine sh
apk update && apk add curl docker
exit
docker commit node dockerforjobseekers/node-docker
```

d. **Line 21:** In the build step, we just run the `npm install` command.

e. **Line 27:** In the test step, we run the `npm test` command.

f. **Line 33:** Here, we build the Docker image using the `registry` environment variable to name the image and the Jenkins build variable `$BUILD_NUMBER` to give it a tag.

g. **Line 34:** We login to our Docker account using the password retrieved from the Jenkins secret store.

h. **Line 35:** We push the image to Docker.

i. **Line 36:** We remove the image from the local image cache. This step is optional.

2. Push the changes to GitHub:

```
git add .
git commit -m "Modifying Jenkinsfile to a CI pipeline for web"
git push
```

3. Add the secret to Jenkins:

   a. In Jenkins, navigate to the **Dashboard**.

   b. Click **Manage Jenkins** and then click **Manage Credentials**.

   c. On the Credentials view, click on the **Jenkins** store

password here in this file, we use the credentials function to retrieve the value from Jenkin's secrets store, where I will place it under the name `docker-login-pwd`.

b. **Lines 6–13**: Here, we define the agent on which Jenkins will run the pipeline. We are using a special Docker image called `dockerforjobseekers/node-docker` that we built. It is based on `node:16-alpine` and has additionally `curl` and the Docker CLI installed. In Line 11, we mount the Docker socket into the container such as that we can run Docker command from withing the container.

c. **Optional**: If you want to build your own image, here are the necessary commands to prepare the image:

```
docker pull node:16-alpine
docker run --name node -it node:16-alpine sh
apk update && apk add curl docker
exit
docker commit node dockerforjobseekers/node-docker
```

d. **Line 21:** In the build step, we just run the `npm install` command.

e. **Line 27:** In the test step, we run the `npm test` command.

f. **Line 33:** Here, we build the Docker image using the `registry` environment variable to name the image and the Jenkins build variable `$BUILD_NUMBER` to give it a tag.

g. **Line 34:** We login to our Docker account using the password retrieved from the Jenkins secret store.

h. **Line 35:** We push the image to Docker.

i. **Line 36:** We remove the image from the local image cache. This step is optional.

2. Push the changes to GitHub:

```
git add .
git commit -m "Modifying Jenkinsfile to a CI pipeline for web"
git push
```

3. Add the secret to Jenkins:

   a. In Jenkins, navigate to the **Dashboard**.

   b. Click **Manage Jenkins** and then click **Manage Credentials**.

   c. On the Credentials view, click on the **Jenkins** store

d. On the System view, click on **Global Credentials (unrestricted)**.

e. On the **Global Credentials (unrestricted)** view, click on **Add Credentials** in the left side bar.

f. Fill in the details as follows:

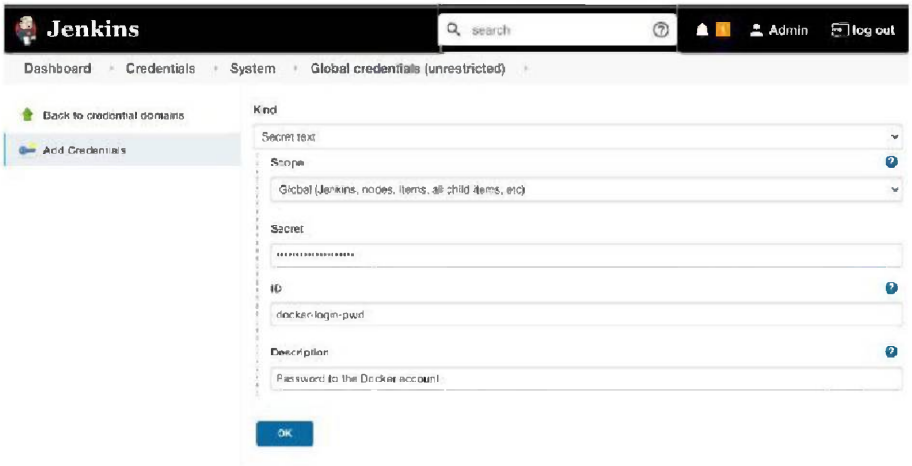


Figure 9.31: Adding a secret text to Jenkins

Add your Docker password as the **Secret** text box, and then hit **Save**.

g. Navigate back to the dashboard and select the `sample-pipeline`. Click **Build Now** to trigger a new build.

h. Wait until the build is completed. The build should succeed!

If, for any reason, the build errors out click on the red "x" icon left to the build number to open the full log. You can now easily find out what went wrong.

Remember that all steps run inside the Docker container, which is our agent. But since we have installed the Docker CLI and mapped the Docker socket into the container, we can use Docker commands.

Let us now, as a last exercise, add continuous delivery (CD) to the pipeline.

Adding continuous delivery

We are now adding two additional steps to the CI/CD pipeline.

1. Add the following step right at the bottom of the Jenkins file:

d. On the System view, click on **Global Credentials (unrestricted)**.

e. On the **Global Credentials (unrestricted)** view, click on **Add Credentials** in the left side bar.

f. Fill in the details as follows:

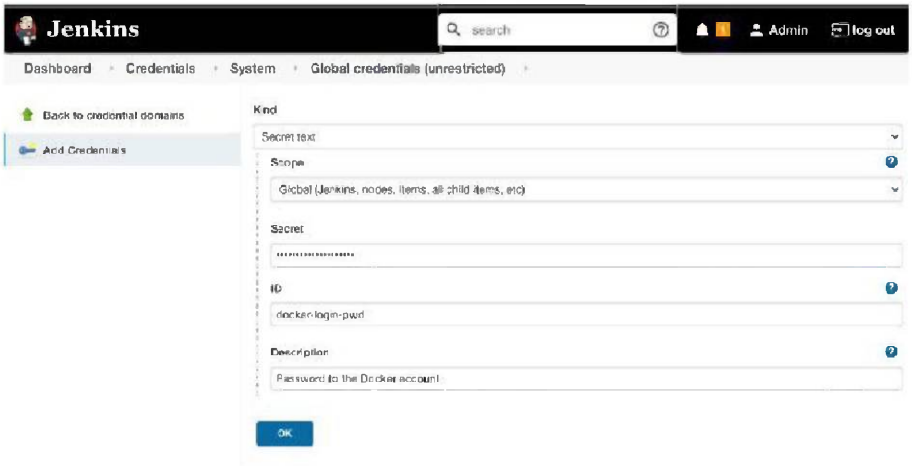


Figure 9.31: Adding a secret text to Jenkins

Add your Docker password as the **Secret** text box, and then hit **Save**.

g. Navigate back to the dashboard and select the `sample-pipeline`. Click **Build Now** to trigger a new build.

h. Wait until the build is completed. The build should succeed!

If, for any reason, the build errors out click on the red "x" icon left to the build number to open the full log. You can now easily find out what went wrong.

Remember that all steps run inside the Docker container, which is our agent. But since we have installed the Docker CLI and mapped the Docker socket into the container, we can use Docker commands.

Let us now, as a last exercise, add continuous delivery (CD) to the pipeline.

Adding continuous delivery

We are now adding two additional steps to the CI/CD pipeline.

1. Add the following step right at the bottom of the Jenkins file:

```
stage('Deploy and smoke test') {
    steps {
        sh './jenkins/scripts/deploy.sh'
        sh './jenkins/scripts/smoke-test.sh'
    }
}
```

2. Next, add a new subfolder **`jenkins/scripts`**, to the root folder of the sample application.

3. To this, new subfolder, add a file **`deploy.sh`** with the following content:

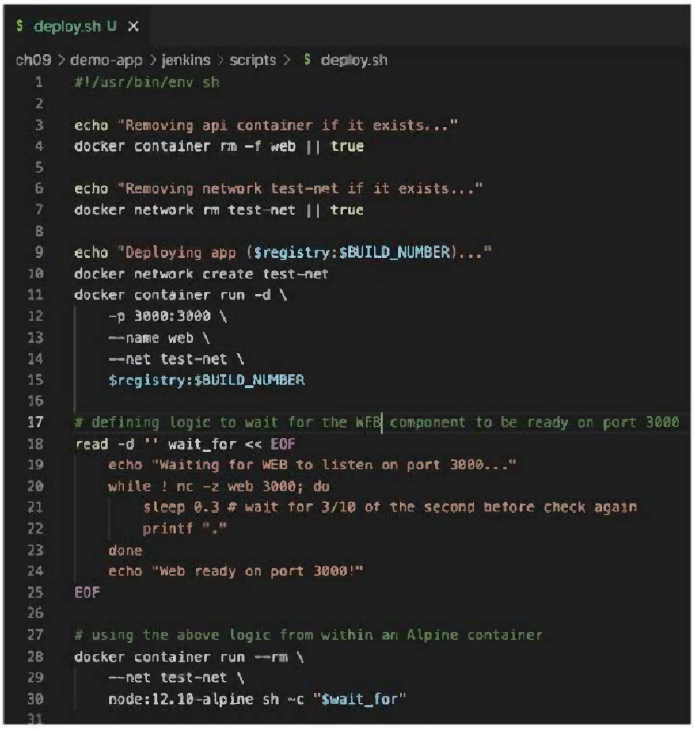


Figure 9.32: Script used by Jenkins to deploy the web Docker image

   a. In Line 4, we make sure any previous container called **`web`** is removed from the system.

   b. In Line 7, we do the same for the Docker network **`test-net`**.

```
stage('Deploy and smoke test') {
    steps {
        sh './jenkins/scripts/deploy.sh'
        sh './jenkins/scripts/smoke-test.sh'
    }
}
```

2. Next, add a new subfolder **`jenkins/scripts`**, to the root folder of the sample application.

3. To this, new subfolder, add a file **`deploy.sh`** with the following content:

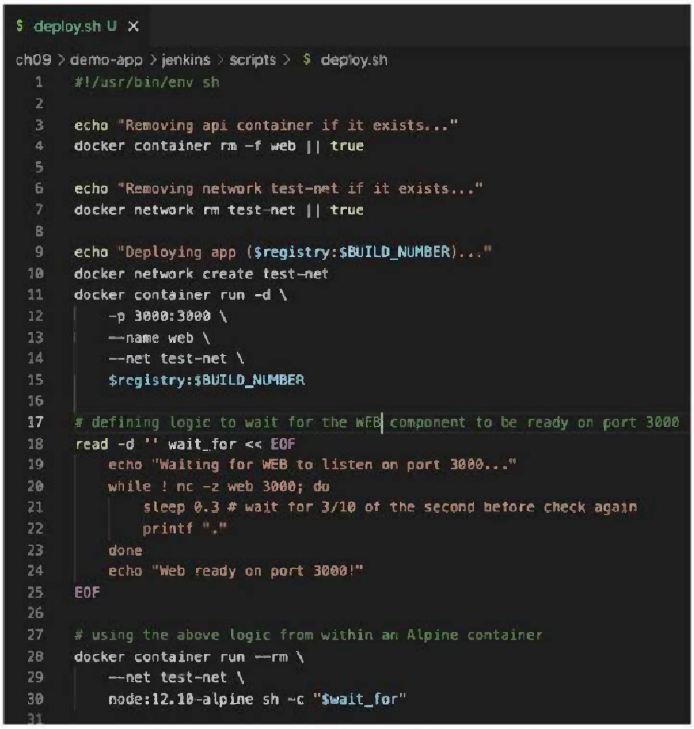


Figure 9.32: Script used by Jenkins to deploy the web Docker image

   a. In Line 4, we make sure any previous container called **`web`** is removed from the system.

   b. In Line 7, we do the same for the Docker network **`test-net`**.

c. In Line 10, we create a network **test-net**.

d. In Line 11, we deploy the newly build Docker image for the web project to the network just created in the previous step. We also make sure to expose container port 3000 to the equivalent host port.

e. In Lines 18 to 25, we define a script that uses the **netcat** (short **nc**) Linux tool to probe port 3000. We store the script in a variable called **wait_for**.

f. In Line 28, we use an Alpine container, attach it to the same network as the **web** container and execute the preceding script from within it.

4. Add a file **smoke-test.sh** to the same folder with this content:



Figure 9.33: Script used by Jenkins to execute smoke tests

Here, we are using our **dockerforjobseekers/node-docker** image to run a container named **tester**, and from within this tester container, we curl the **web:3000** endpoint. Alternatively, since we expose port 3000 to the host, we could also just run the following command directly:

```
curl localhost:3000
```

5. Make the two files executable with the following:

```
chmod +x jenkins/scripts/deploy.sh
chmod +x jenkins/scripts/smoke-test.sh
```

6. As a last stage, we will add a cleanup step as follows:

```
stage('Cleanup') {
    steps {
        sh './jenkins/scripts/cleanup.sh'
    }
}
```

This step just uses the script **cleanup.sh** in the **jenkins/scripts** directory

c. In Line 10, we create a network **test-net**.

d. In Line 11, we deploy the newly build Docker image for the web project to the network just created in the previous step. We also make sure to expose container port 3000 to the equivalent host port.

e. In Lines 18 to 25, we define a script that uses the **netcat** (short **nc**) Linux tool to probe port 3000. We store the script in a variable called **wait_for**.

f. In Line 28, we use an Alpine container, attach it to the same network as the **web** container and execute the preceding script from within it.

4. Add a file **smoke-test.sh** to the same folder with this content:

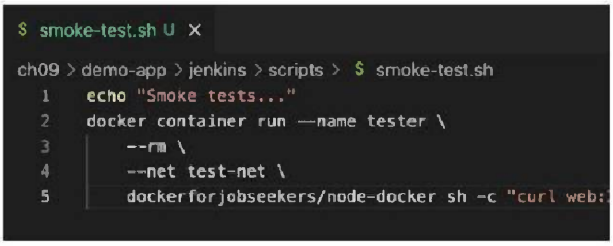


Figure 9.33: Script used by Jenkins to execute smoke tests

Here, we are using our **dockerforjobseekers/node-docker** image to run a container named **tester**, and from within this tester container, we curl the **web:3000** endpoint. Alternatively, since we expose port 3000 to the host, we could also just run the following command directly:

```
curl localhost:3000
```

5. Make the two files executable with the following:

```
chmod +x jenkins/scripts/deploy.sh
chmod +x jenkins/scripts/smoke-test.sh
```

6. As a last stage, we will add a cleanup step as follows:

```
stage('Cleanup') {
    steps {
        sh './jenkins/scripts/cleanup.sh'
    }
}
```

This step just uses the script **cleanup.sh** in the **jenkins/scripts** directory

7. Add a file **`cleanup.sh`** to the **`jenkins/scripts`** folder with the following content:

```
$ cleanup.sh U ×

ch09 > demo-app > jenkins > scripts > $ cleanup.sh
  1  #!/usr/bin/env sh
  2  docker rm -f web
  3  docker network rm test-net
```

Figure 9.34: *Cleanup script for Jenkins*

8. Make the cleanup file executable with the following:

```
chmod +x jenkins/scripts/cleanup.sh
```

9. Push your code changes to GitHub with the following:

```
git -a && git commit -m "Defined the CD part of CI/CD"
git push origin main
```

10. Back in Jenkins and from its dashboard, select the **`sample-pipeline`**. Click **Build Now** to trigger a new build. Jenkins will start to build the pipeline. If everything goes well, you should see something similar to this:

Stage View



Figure 9.35: *Complete build pipeline in Jenkins*

With this, you have learned how to build a complete CI/CD pipeline for Jenkins.

Next, we will discuss some popular cloud offerings in the area of DevOps and CI/CD.

Discussing some cloud-based SaaS offering for your CI/CD pipeline

In this section, we discuss three popular SaaS offerings for building CI/CD pipelines for your software projects and/or products. Let us start with the Microsoft offering.

7. Add a file **`cleanup.sh`** to the **`jenkins/scripts`** folder with the following content:

```
$ cleanup.sh U ×

ch09 > demo-app > jenkins > scripts > $ cleanup.sh
  1  #!/usr/bin/env sh
  2  docker rm -f web
  3  docker network rm test-net
```

Figure 9.34: *Cleanup script for Jenkins*

8. Make the cleanup file executable with the following:

```
chmod +x jenkins/scripts/cleanup.sh
```

9. Push your code changes to GitHub with the following:

```
git -a && git commit -m "Defined the CD part of CI/CD"
git push origin main
```

10. Back in Jenkins and from its dashboard, select the **`sample-pipeline`**. Click **Build Now** to trigger a new build. Jenkins will start to build the pipeline. If everything goes well, you should see something similar to this:

Stage View



Figure 9.35: *Complete build pipeline in Jenkins*

With this, you have learned how to build a complete CI/CD pipeline for Jenkins.

Next, we will discuss some popular cloud offerings in the area of DevOps and CI/CD.

Discussing some cloud-based SaaS offering for your CI/CD pipeline

In this section, we discuss three popular SaaS offerings for building CI/CD pipelines for your software projects and/or products. Let us start with the Microsoft offering.

Azure DevOps

Azure DevOps is an offering by Microsoft and provides full integration CI/CD with all the other services offered on Azure, such as issue management and source control management. With Azure DevOps, you can create pipelines that build, test, and deploy your code. This works with any of the popular languages, frameworks, or platforms. It is straightforward to manage, e.g., NodeJS, Python, PHP, Ruby, C/C++, .NET, Android, or iOS applications with Azure DevOps pipelines. To create container images, push them to any container registry, and deploy them to Kubernetes is easy and supported by powerful templates or plugins that free you from having to write and maintain complicated scripts.

Code can reside in a Git repository on Azure (Azure Repos) or other popular hosters of Git-compatible repositories such as GitHub or BitBucket. **Continuous deployment** (CD) integrates with any cloud, such as Azure, AWS, or GCP. Targets can be Linux or Windows-based container host such as the hosted Kubernetes service of Azure (AKS), virtual machines, or Azure functions, to just name a few.

As an added bonus, the service is free for open-source projects.

A typical template for a simple project looks like the following:



Figure 9.36: Sample Azure DevOps CI/CD pipeline

Azure DevOps

Azure DevOps is an offering by Microsoft and provides full integration CI/CD with all the other services offered on Azure, such as issue management and source control management. With Azure DevOps, you can create pipelines that build, test, and deploy your code. This works with any of the popular languages, frameworks, or platforms. It is straightforward to manage, e.g., NodeJS, Python, PHP, Ruby, C/C++, .NET, Android, or iOS applications with Azure DevOps pipelines. To create container images, push them to any container registry, and deploy them to Kubernetes is easy and supported by powerful templates or plugins that free you from having to write and maintain complicated scripts.

Code can reside in a Git repository on Azure (Azure Repos) or other popular hosters of Git-compatible repositories such as GitHub or BitBucket. **Continuous deployment (CD)** integrates with any cloud, such as Azure, AWS, or GCP. Targets can be Linux or Windows-based container host such as the hosted Kubernetes service of Azure (AKS), virtual machines, or Azure functions, to just name a few.

As an added bonus, the service is free for open-source projects.

A typical template for a simple project looks like the following:

azure-pipelines.yml

ch09 > Azure > azure-pipelines.yml > {} pool > vmImage

Pipeline schema (service-schema.json)

```
trigger:
- main

pool:
  vmImage: 'ubuntu-20.04'

variables:
  buildConfiguration: 'Release'

steps:
- task: UseDotNet@2
  inputs:
    version: '5.0.x'

- task: DotNetCoreCLI@2
  inputs:
    command: 'restore'
  displayName: 'dotnet restore $(buildConfiguration)'

- task: DotNetCoreCLI@2
  inputs:
    command: 'build'
    projects: '**/*.csproj'
    arguments: '--configuration $(buildConfiguration)'
  displayName: 'dotnet build $(buildConfiguration)'

- task: DotNetCoreCLI@2
  inputs:
    command: test
    projects: '**/*Tests/*.csproj'
    arguments: '--configuration $(buildConfiguration)'
  displayName: 'dotnet test $(buildConfiguration)'
```

Figure 9.36: *Sample Azure DevOps CI/CD pipeline*

An Azure pipeline is usually described in a file called `azure-pipelines.yml` that sits in the root folder of the software project. The pipeline consists of a list of steps. Each step is either a task or a script. In the preceding example, we have defined a pipeline for a .NET project. The pipeline contains the following details:

- Lines 1–2: Defines the trigger that executes the pipeline. Here it is any commit to the branch `main` in the source code repository.
- Lines 4–5: Defines the image that the build agent uses, to execute the subsequent steps. In this case we are using Ubuntu 20.04 for our build agent.
- Lines 7–8: In the pipeline, one can use variables. Here, we only define a single variable `buildConfiguration` that is then used further down in the pipeline, e.g., on Line 18.
- Lines 10–32: a series of four steps. Each step consists of a task:
    1. Lines 11–13: The first task is declaring which version of the .NET framework to use.
    2. Lines 15–18: This task executes a `dotnet restore` for the project at hand. The display name defined on Line 18 is useful when analyzing the log output of the build process.
    3. Lines 20–25: This task builds the project using the command `dotnet build`. It uses the build configuration defined in the variable `buildConfiguration`.
    4. Lines 27–32: After a successful build, the pipeline runs all the automated tests against the application using the command `dotnet test`.

The author has been using Azure DevOps to manage mission-critical applications end-to-end. The services were easy to use, powerful, and covered all of his needs.

After peeking into the Microsoft offering, let us have a look at what the number one cloud provider AWS has in the sleeves for us.

AWS DevOps

Another major contender in the space is of course AWS. The CI/CD service on AWS is called AWS CodePipeline. Similar to Azure pipelines, it offers services to build, test, and deploy your applications. Pipelines are triggered each time there is a code change. The whole release process can be fully configured to fit your specific needs. This can be done either via the console interface, the well-known AWS CLI, AWS CloudFormation, or the AWS SDK.

An Azure pipeline is usually described in a file called `azure-pipelines.yml` that sits in the root folder of the software project. The pipeline consists of a list of steps. Each step is either a task or a script. In the preceding example, we have defined a pipeline for a .NET project. The pipeline contains the following details:

- Lines 1–2: Defines the trigger that executes the pipeline. Here it is any commit to the branch `main` in the source code repository.
- Lines 4–5: Defines the image that the build agent uses, to execute the subsequent steps. In this case we are using Ubuntu 20.04 for our build agent.
- Lines 7–8: In the pipeline, one can use variables. Here, we only define a single variable `buildConfiguration` that is then used further down in the pipeline, e.g., on Line 18.
- Lines 10–32: a series of four steps. Each step consists of a task:
    1. Lines 11–13: The first task is declaring which version of the .NET framework to use.
    2. Lines 15–18: This task executes a `dotnet restore` for the project at hand. The display name defined on Line 18 is useful when analyzing the log output of the build process.
    3. Lines 20–25: This task builds the project using the command `dotnet build`. It uses the build configuration defined in the variable `buildConfiguration`.
    4. Lines 27–32: After a successful build, the pipeline runs all the automated tests against the application using the command `dotnet test`.

The author has been using Azure DevOps to manage mission-critical applications end-to-end. The services were easy to use, powerful, and covered all of his needs.

After peeking into the Microsoft offering, let us have a look at what the number one cloud provider AWS has in the sleeves for us.

AWS DevOps

Another major contender in the space is of course AWS. The CI/CD service on AWS is called AWS CodePipeline. Similar to Azure pipelines, it offers services to build, test, and deploy your applications. Pipelines are triggered each time there is a code change. The whole release process can be fully configured to fit your specific needs. This can be done either via the console interface, the well-known AWS CLI, AWS CloudFormation, or the AWS SDK.

Cost is determined on a usage base. You only pay for what you need. CodePipeline is fully managed, and it can be connected to your existing tools and systems.

To define a pipeline, we can use a build specification file, which is a collection of build commands and related settings that is written in YAML. The file is often called **`buildspec.yml`** and is added to the root of the project.

A typical template for a simple project looks like the following:

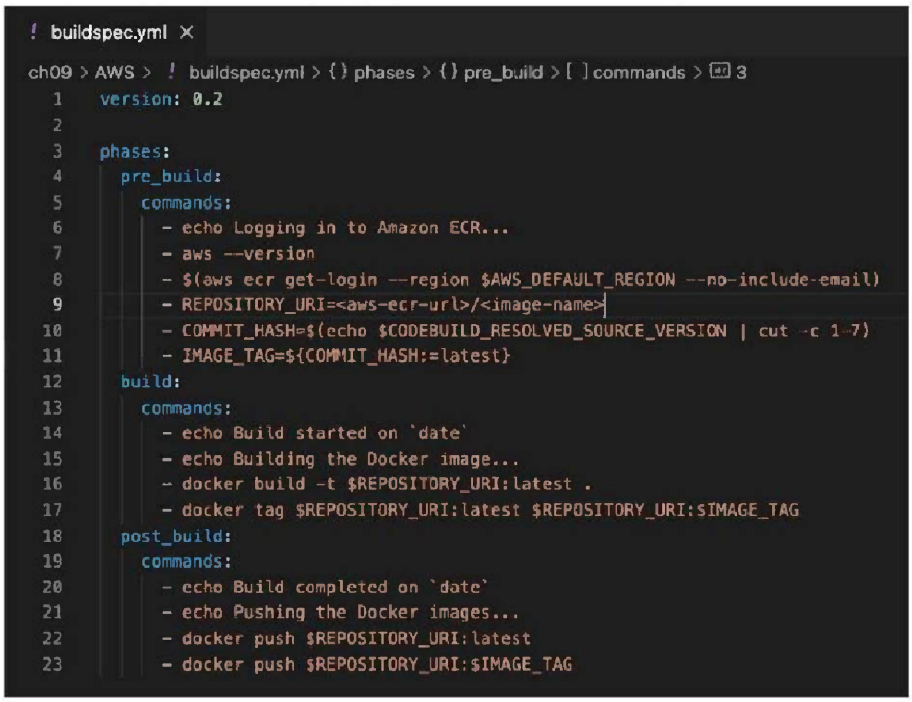


Figure 9.37: Sample AWS DevOps CI/CD pipeline

AWS CodePipeline organizes the pipelines into phases. Each phase then consists of a series of commands. As an example, the preceding pipeline is consisting of three phases **`pre_build`**, **`build`**, and **`post_build`**:

- **`pre_build`** commands:
    - **Line 8:** Login to AWS container service (ECR)
    - **Line 9:** Define the full name of the Docker image to build, where **`<aws-ecr-url>`** is the URL to your container repository on ECR (such as **`012345678910.dkr.ecr.us-east-1.amazonaws.com`**)

Cost is determined on a usage base. You only pay for what you need. CodePipeline is fully managed, and it can be connected to your existing tools and systems.

To define a pipeline, we can use a build specification file, which is a collection of build commands and related settings that is written in YAML. The file is often called **`buildspec.yml`** and is added to the root of the project.

A typical template for a simple project looks like the following:

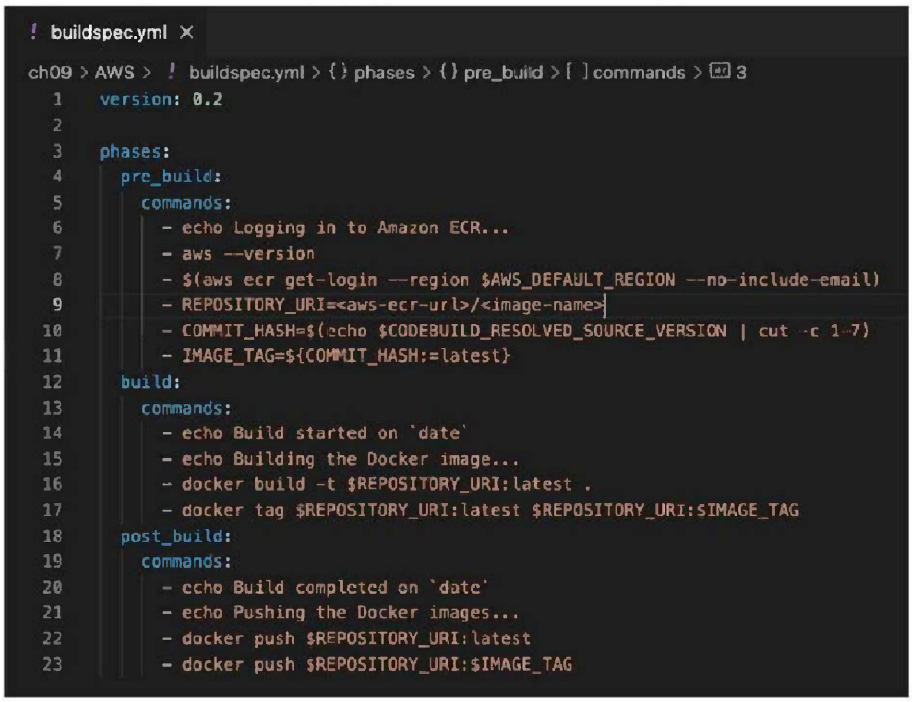


Figure 9.37: Sample AWS DevOps CI/CD pipeline

AWS CodePipeline organizes the pipelines into phases. Each phase then consists of a series of commands. As an example, the preceding pipeline is consisting of three phases **`pre_build`**, **`build`**, and **`post_build`**:

- **`pre_build`** commands:
  - **Line 8:** Login to AWS container service (ECR)
  - **Line 9:** Define the full name of the Docker image to build, where **`<aws-ecr-url>`** is the URL to your container repository on ECR (such as **`012345678910.dkr.ecr.us-east-1.amazonaws.com`**)

and `<image-name>` is the name of the image you are going to build such as `web`.

- **Line 10:** Get the first seven characters of the last Git commit. It will be used to tag the container image
- **Line 11:** Define the image tag using the commit hash from previously

- `build` commands:
  - **Line 16:** Build the Docker image using the tag `latest`
  - **Line 17:** Tag the preceding image also with the tag stored in the variable `$IMAGE_TAG` (defined on Line 11)

- `post_build` commands:
  - **Line 22:** Push the image with tag `latest` to ECR
  - **Line 23:** Push the image with the tag derived from the Git commit to ECR

As you can see, we can use any valid shell command in the list of commands of a phase in the pipeline.

Now that we have briefly covered the AWS CodePipeline product let us have a glance at another popular offering from GitLab.

GitLab

GitLab is a DevOps platform and is offered as a SaaS solution or can also be self-hosted. To avoid additional management overhead, the author recommends the former.

GitLab does offer a free trial option for its paid plans. The free trial allows you to test the features and functionalities of the paid plans before committing to a subscription. The free trial is usually valid for 30 days, and it allows you to explore the full capabilities of the platform, including features such as GitLab's built-in continuous integration and delivery (CI/CD), access to the GitLab Container Registry, and many more.

During the free trial, you will have access to the same features and functionalities as paid customers. You will also have access to support, documentation, and community resources.

To start a free trial, you can sign up for a new account on **https://gitlab.com**, or you can contact the GitLab sales team, and they will assist you with the process.

and `<image-name>` is the name of the image you are going to build such as `web`.

- **Line 10:** Get the first seven characters of the last Git commit. It will be used to tag the container image
- **Line 11:** Define the image tag using the commit hash from previously

- `build` commands:
  - **Line 16:** Build the Docker image using the tag `latest`
  - **Line 17:** Tag the preceding image also with the tag stored in the variable `$IMAGE_TAG` (defined on Line 11)

- `post_build` commands:
  - **Line 22:** Push the image with tag `latest` to ECR
  - **Line 23:** Push the image with the tag derived from the Git commit to ECR

As you can see, we can use any valid shell command in the list of commands of a phase in the pipeline.

Now that we have briefly covered the AWS CodePipeline product let us have a glance at another popular offering from GitLab.

GitLab

GitLab is a DevOps platform and is offered as a SaaS solution or can also be self-hosted. To avoid additional management overhead, the author recommends the former.

GitLab does offer a free trial option for its paid plans. The free trial allows you to test the features and functionalities of the paid plans before committing to a subscription. The free trial is usually valid for 30 days, and it allows you to explore the full capabilities of the platform, including features such as GitLab's built-in continuous integration and delivery (CI/CD), access to the GitLab Container Registry, and many more.

During the free trial, you will have access to the same features and functionalities as paid customers. You will also have access to support, documentation, and community resources.

To start a free trial, you can sign up for a new account on **https://gitlab.com**, or you can contact the GitLab sales team, and they will assist you with the process.

It is important to note that GitLab also has a free community edition, which is a limited version of the software that provides basic functionality; this free edition does not require a free trial and can be used for free for an unlimited time.

GitLab can be used to manage and optimize the software delivery lifecycle, plan progress through epics, groups, and milestones, help teams to design, develop and securely manage code and project data, verify quality with automated testing and reporting, and finally, package and deploy applications and their dependencies.

GitLab is used by many renowned companies such as Nvidia, Siemens, and Goldman Sachs.

For the CI/CD part, GitLab—as do Azure and AWS DevOps—uses pipelines that are written in YAML and stored and version controlled with the code.

A simple pipeline on GitLab looks like the following:

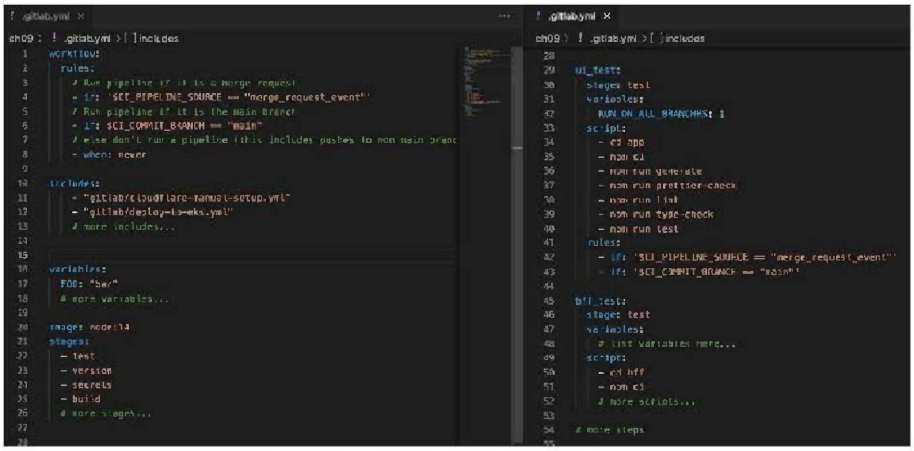


Figure 9.38: Sample GitLab CI/CD pipeline

As mentioned previously, GitLab pipelines are defined using YAML. By default, the pipeline is stored in the root folder of a project, and the name of the file is **`.gitlab.yml`**.

Let us explain in detail what the preceding sample pipeline consists of.

- In Lines 1 to 8, we define the workflow rules that apply for this pipeline. In this example, the pipeline is only triggered if either there was a merge request event discovered or a commit happened to the "main" branch of the repository. Any other scenario is ignored.

It is important to note that GitLab also has a free community edition, which is a limited version of the software that provides basic functionality; this free edition does not require a free trial and can be used for free for an unlimited time.

GitLab can be used to manage and optimize the software delivery lifecycle, plan progress through epics, groups, and milestones, help teams to design, develop and securely manage code and project data, verify quality with automated testing and reporting, and finally, package and deploy applications and their dependencies.

GitLab is used by many renowned companies such as Nvidia, Siemens, and Goldman Sachs.

For the CI/CD part, GitLab—as do Azure and AWS DevOps—uses pipelines that are written in YAML and stored and version controlled with the code.

A simple pipeline on GitLab looks like the following:

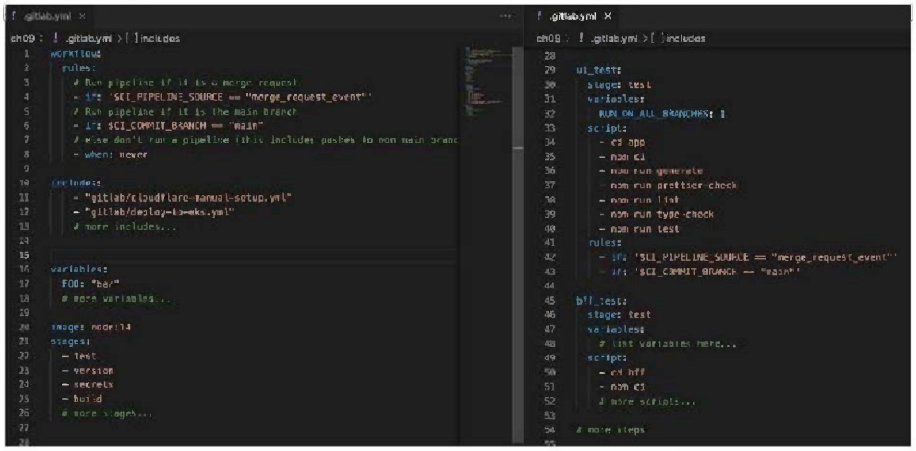


Figure 9.38: Sample GitLab CI/CD pipeline

As mentioned previously, GitLab pipelines are defined using YAML. By default, the pipeline is stored in the root folder of a project, and the name of the file is **`.gitlab.yml`**.

Let us explain in detail what the preceding sample pipeline consists of.

- In Lines 1 to 8, we define the workflow rules that apply for this pipeline. In this example, the pipeline is only triggered if either there was a merge request event discovered or a commit happened to the "main" branch of the repository. Any other scenario is ignored.

- The pipeline can be split into several (reusable) files. In Lines 10 to 13, we can see how the content of other files is included in this pipeline. Apparently, we have some templates called `cloudflare.manual.setup.yml` and `deploy-to-eks.yml` in a subfolder `gitlab` that we want to use in this example.
- On Lines 16 to 18, we show how one can define variables that can then be used in the remains of the file. Here, as an example, we define a variable called `FOO` and assign it the value `bar`. GitLab has a whole list of predefined variables that all start with `CI_`, such as `CI_COMMIT_BRANCH` used in Line 4. GitLab is populating the values of those variables.
- In Line 20, we define what image the build agent of GitLab shall be using for all the subsequent steps. In this case, we use the NodeJS version 14 image since our application is a React JS application.
- In Lines 21 to 26, we define the stages that this pipeline consists of. In our example, we have, e.g., a test, a version, a secrets, and a build stage.
- From Lines 29 to 43, we define a section `ui_test`. In this section, we define what shall happen in the stage test. The list of scripts that are run through by the build agent is shown on Lines 33 to 40. It is what you typically do when initializing, building, linting, type checking, and testing your React JS application.
- From Lines 45 to 52, we define a section `bff_test`. In our case, the React JS UI is interfacing with a **backend for frontend** (**BFF**) written in Node JS. The structure is very similar to the one of the `ui_test` section.

The author is using GitLab in his current product, a SaaS solution for insurances. GitLab is very powerful, flexible, and easy to manage. It supports all the steps in the products' CI/CD requirements, such as building, linting, and testing the code, mainly consisting of Java Spring Boot microservices and React SPA applications. The pipeline integrates with Nexus repository, Split.io for feature toggles, SonarQube for code quality checks, GitLab repositories, AWS EKS as the deployment target, AWS Secrets Manager to manage and store secrets, and many more services.

OK, you should notice that each offering has a similar approach on how to define a build pipeline. It is always a YAML file, by default, located at the root of the project. A pipeline consists of a series of stages or steps, which in turn are made up of a series of commands. Once you have mastered one provider, it should be fairly trivial for you to familiarize yourself with another one.

The list of SaaS offerings discussed preceding is by far not complete. We have only presented some of the most popular offerings. Others worth noticing are Circle CI and Travis CI.

With this, we are at the end of this chapter. Let us have a look back on what we have learned.

- The pipeline can be split into several (reusable) files. In Lines 10 to 13, we can see how the content of other files is included in this pipeline. Apparently, we have some templates called `cloudflare.manual.setup.yml` and `deploy-to-eks.yml` in a subfolder `gitlab` that we want to use in this example.
- On Lines 16 to 18, we show how one can define variables that can then be used in the remains of the file. Here, as an example, we define a variable called `FOO` and assign it the value `bar`. GitLab has a whole list of predefined variables that all start with `CI_`, such as `CI_COMMIT_BRANCH` used in Line 4. GitLab is populating the values of those variables.
- In Line 20, we define what image the build agent of GitLab shall be using for all the subsequent steps. In this case, we use the NodeJS version 14 image since our application is a React JS application.
- In Lines 21 to 26, we define the stages that this pipeline consists of. In our example, we have, e.g., a test, a version, a secrets, and a build stage.
- From Lines 29 to 43, we define a section `ui_test`. In this section, we define what shall happen in the stage test. The list of scripts that are run through by the build agent is shown on Lines 33 to 40. It is what you typically do when initializing, building, linting, type checking, and testing your React JS application.
- From Lines 45 to 52, we define a section `bff_test`. In our case, the React JS UI is interfacing with a **backend for frontend** (**BFF**) written in Node JS. The structure is very similar to the one of the `ui_test` section.

The author is using GitLab in his current product, a SaaS solution for insurances. GitLab is very powerful, flexible, and easy to manage. It supports all the steps in the products' CI/CD requirements, such as building, linting, and testing the code, mainly consisting of Java Spring Boot microservices and React SPA applications. The pipeline integrates with Nexus repository, Split.io for feature toggles, SonarQube for code quality checks, GitLab repositories, AWS EKS as the deployment target, AWS Secrets Manager to manage and store secrets, and many more services.

OK, you should notice that each offering has a similar approach on how to define a build pipeline. It is always a YAML file, by default, located at the root of the project. A pipeline consists of a series of stages or steps, which in turn are made up of a series of commands. Once you have mastered one provider, it should be fairly trivial for you to familiarize yourself with another one.

The list of SaaS offerings discussed preceding is by far not complete. We have only presented some of the most popular offerings. Others worth noticing are Circle CI and Travis CI.

With this, we are at the end of this chapter. Let us have a look back on what we have learned.

Conclusion

In this chapter, we have explained what a continuous integration and continuous deployment or delivery (CI/CD) pipeline is. We then have built a simple pipeline using GitHub actions. Next, we have used the very popular automation server Jenkins to implement a simple pipeline. Finally, we have discussed a few of the most popular SaaS offerings for DevOps in general and for CI/CD in particular.

Questions

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

1. Name the steps required to have GitHub automatically build a Docker image from your code.
2. What are the steps required to have Jenkins build and test your code and, if successful, build a Docker image?
3. There are many SaaS offerings available for you to choose to build your CI/CD pipelines. What are some criteria that can help you to do an informed selection of one offering for your own purposes?

Answers

Here are sample answers to the preceding questions.

1. The following steps are needed:
    a. Create a `.github` folder at the root of your project
    b. Add a YAML file to the preceding folder containing the definition of a workflow
    c. In the YAML file, define the name of the workflow, the event (or action) that triggers a workflow, and the jobs that are executed during the workflow
    d. For each job, define the type of agent on which it is running and the list of steps that need to be executed during this step. A step can be a script or a predefined GitHub action. In our case, we need at least the following two steps:
        i. Use a step with a GitHub action to check out the code (`actions/checkout@v2`)
        ii. Use an additional step with a script that builds the Docker image from the code using the Docker CLI (e.g., `docker image build -t ...`)
    e. Push the workflow to GitHub. It will be executed automatically.

Conclusion

In this chapter, we have explained what a continuous integration and continuous deployment or delivery (CI/CD) pipeline is. We then have built a simple pipeline using GitHub actions. Next, we have used the very popular automation server Jenkins to implement a simple pipeline. Finally, we have discussed a few of the most popular SaaS offerings for DevOps in general and for CI/CD in particular.

Questions

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

1. Name the steps required to have GitHub automatically build a Docker image from your code.
2. What are the steps required to have Jenkins build and test your code and, if successful, build a Docker image?
3. There are many SaaS offerings available for you to choose to build your CI/CD pipelines. What are some criteria that can help you to do an informed selection of one offering for your own purposes?

Answers

Here are sample answers to the preceding questions.

1. The following steps are needed:
    a. Create a `.github` folder at the root of your project
    b. Add a YAML file to the preceding folder containing the definition of a workflow
    c. In the YAML file, define the name of the workflow, the event (or action) that triggers a workflow, and the jobs that are executed during the workflow
    d. For each job, define the type of agent on which it is running and the list of steps that need to be executed during this step. A step can be a script or a predefined GitHub action. In our case, we need at least the following two steps:
        i. Use a step with a GitHub action to check out the code (`actions/checkout@v2`)
        ii. Use an additional step with a script that builds the Docker image from the code using the Docker CLI (e.g., `docker image build -t ...`)
    e. Push the workflow to GitHub. It will be executed automatically.

2. Add a file called Jenkinsfile to the root of your project, which contains the definition of the environment, the agent, the stages, and the steps per stage. The agent needs to be defined such as that it is able to build, push, and deploy a Docker image. One can use an existing Docker image or a customized one as the agents' runtime environment. Then, one needs to define the stages of the pipeline, such as test, build image, push image, and deploy image. Each stage can consist of one-to-many steps in which we define what exactly has to happen. Each step is basically a script that the agent will execute. It can be a shell script or a Docker command to just name a few.
3. An (incomplete) list of criteria that should be used when selecting the SaaS offering for your DevOps, specifically CI/CD:
    a. Cost
    b. Does it provide all the features you need, such as building, pushing, and deploying Docker images
    c. Does the DevOps offering integrate well with the other tools or offerings you are using in your software development lifecycle?
    d. Is the service scalable, and can it be extended with custom-built features?

Job Interview sample questions

Today is your first interview for the role as DevOps engineer. The job is offered by a mid-size company that is tipping its toes into DevOps. The job profile states that one of your new responsibilities would be to fully automate the company's CI/CD pipelines. The interviewer faces you with the following questions:

1. Some of our developers do not see the reason why we have to fully automate our build process. They are happy to build the artifacts manually in their IDEs and deploy them to production from there when the time is ready for a new release. Can you explain why it is so important to automate the build and release process?
2. We are pondering about the pros and cons of using a SaaS solution to build our CI/CD pipelines or to self-host and manage a build server such as Jenkins or TeamCity. Tell us your opinion. On what criteria do you base your judgment?
3. On the whiteboard (or the back of a napkin), draw the essential elements in a CI/CD pipeline, concentrating on the CI part.

2. Add a file called Jenkinsfile to the root of your project, which contains the definition of the environment, the agent, the stages, and the steps per stage. The agent needs to be defined such as that it is able to build, push, and deploy a Docker image. One can use an existing Docker image or a customized one as the agents' runtime environment. Then, one needs to define the stages of the pipeline, such as test, build image, push image, and deploy image. Each stage can consist of one-to-many steps in which we define what exactly has to happen. Each step is basically a script that the agent will execute. It can be a shell script or a Docker command to just name a few.
3. An (incomplete) list of criteria that should be used when selecting the SaaS offering for your DevOps, specifically CI/CD:
    a. Cost
    b. Does it provide all the features you need, such as building, pushing, and deploying Docker images
    c. Does the DevOps offering integrate well with the other tools or offerings you are using in your software development lifecycle?
    d. Is the service scalable, and can it be extended with custom-built features?

Job Interview sample questions

Today is your first interview for the role as DevOps engineer. The job is offered by a mid-size company that is tipping its toes into DevOps. The job profile states that one of your new responsibilities would be to fully automate the company's CI/CD pipelines. The interviewer faces you with the following questions:

1. Some of our developers do not see the reason why we have to fully automate our build process. They are happy to build the artifacts manually in their IDEs and deploy them to production from there when the time is ready for a new release. Can you explain why it is so important to automate the build and release process?
2. We are pondering about the pros and cons of using a SaaS solution to build our CI/CD pipelines or to self-host and manage a build server such as Jenkins or TeamCity. Tell us your opinion. On what criteria do you base your judgment?
3. On the whiteboard (or the back of a napkin), draw the essential elements in a CI/CD pipeline, concentrating on the CI part.

Answers

Sample answers to the interview questions:

1. Whenever humans are executing a series of complex tasks manually, then the chance of introducing an error is high. Specifically, under pressure, humans tend to be unreliable. It is thus important to automate every step of a DevOps CI/CD pipeline and, with it, make it reliable, repeatable, and auditable. A new build and release should be possible by pushing the new feature to the code repository or alternatively with a simple click of a button. Developers are very expensive resources, and thus, should concentrate on implementing features that generate value for the company and not on performing repetitive and tedious tasks.
2. CI/CD is something that every software project—independent of the sector of the industry —should use. Thus CI/CD is not a differentiating factor for a company compared to its competitors. In that regard, the company should outsource the management of the DevOps platform. Only companies with very exotic requirements in the area of CI/CD that are not typically covered by hosted DevOps offerings should consider self-host a CI/CD platform.
3. We have the following parts in the picture:
    a. The developer with their laptop pushing code to.
    b. A source code repository such as GitHub, GitLab or BitBucket.
    c. One (or several, depending on the requirement) reviewers that review a pull request of the developer against the main branch of the code repository.
    d. An automation server—ideally a hosted one—that:
        i. Checks out code to an agent
        ii. Builds the code on the agent
        iii. Lints and quality check the code
        iv. Runs the automated tests against this code

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



Answers

Sample answers to the interview questions:

1. Whenever humans are executing a series of complex tasks manually, then the chance of introducing an error is high. Specifically, under pressure, humans tend to be unreliable. It is thus important to automate every step of a DevOps CI/CD pipeline and, with it, make it reliable, repeatable, and auditable. A new build and release should be possible by pushing the new feature to the code repository or alternatively with a simple click of a button. Developers are very expensive resources, and thus, should concentrate on implementing features that generate value for the company and not on performing repetitive and tedious tasks.
2. CI/CD is something that every software project—independent of the sector of the industry —should use. Thus CI/CD is not a differentiating factor for a company compared to its competitors. In that regard, the company should outsource the management of the DevOps platform. Only companies with very exotic requirements in the area of CI/CD that are not typically covered by hosted DevOps offerings should consider self-host a CI/CD platform.
3. We have the following parts in the picture:
    a. The developer with their laptop pushing code to.
    b. A source code repository such as GitHub, GitLab or BitBucket.
    c. One (or several, depending on the requirement) reviewers that review a pull request of the developer against the main branch of the code repository.
    d. An automation server—ideally a hosted one—that:
        i. Checks out code to an agent
        ii. Builds the code on the agent
        iii. Lints and quality check the code
        iv. Runs the automated tests against this code

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 10
Orchestrating Containers

Introduction

In this chapter, we are going to introduce 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 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.

Structure

In this chapter, we will discuss the following topics:

- Introducing container orchestration
- Understanding Kubernetes
- Leveraging managed Kubernetes offerings in the cloud

Chapter 10
Orchestrating Containers

Introduction

In this chapter, we are going to introduce 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 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.

Structure

In this chapter, we will discuss the following topics:

- Introducing container orchestration
- Understanding Kubernetes
- Leveraging managed Kubernetes offerings in the cloud

- Discussing containers in the context of serverless

Objectives

After reading through the chapter, the reader will be able to understand key concepts of a distributed application architecture and why Kubernetes is such an important piece of software in this context, as it alleviates many of the problems we developers are facing when dealing with highly distributed and mission-critical applications. The reader will be familiar with the key concepts of Kubernetes and will leverage this know-how to deploy their own containerized applications to Kubernetes, be it a self-hosted, simple Kubernetes cluster, or a more production-like cluster provided by one of the fully managed Kubernetes offerings available today.

Let us start.

A Kubernetes cluster consists of a set of servers. So far, we have learned a lot about individual containers and how we can build, ship, and use them. We have not really talked so much about putting our containerized applications into production yet. But the goal of every business application built out there is to eventually run in production. What does that mean for us? We will address this question in detail in this section.

Distributed application architecture

In this section, we are going to discuss what we mean when we are talking about a distributed application architecture. Not everyone may be familiar with the underlying concepts. Thus, let us define the most important parts of it. In such an architecture, we will encounter the following elements:

- **VM**: Virtual machine; this corresponds to a virtual computer. We can run many VMs on a single physical computer (often called a server).
- **Node**: An individual server used to run applications. It can be either a physical or a virtual server. The former often is called a bare-metal server, and the latter a VM. Nodes can be anything from a Raspberry PI to a super computer. Nodes can run in the cloud, in a company's own data-center, or even under your desk.
- **Cluster**: A cluster is a group of nodes connected by a network. A cluster is used to run distributed applications.
- **Network**: Physical or software-defined paths between nodes over which the latter and the programs that run on them communicate. The nodes of a cluster are connected to each other by one (or several) network(s).

- Discussing containers in the context of serverless

Objectives

After reading through the chapter, the reader will be able to understand key concepts of a distributed application architecture and why Kubernetes is such an important piece of software in this context, as it alleviates many of the problems we developers are facing when dealing with highly distributed and mission-critical applications. The reader will be familiar with the key concepts of Kubernetes and will leverage this know-how to deploy their own containerized applications to Kubernetes, be it a self-hosted, simple Kubernetes cluster, or a more production-like cluster provided by one of the fully managed Kubernetes offerings available today.

Let us start.

A Kubernetes cluster consists of a set of servers. So far, we have learned a lot about individual containers and how we can build, ship, and use them. We have not really talked so much about putting our containerized applications into production yet. But the goal of every business application built out there is to eventually run in production. What does that mean for us? We will address this question in detail in this section.

Distributed application architecture

In this section, we are going to discuss what we mean when we are talking about a distributed application architecture. Not everyone may be familiar with the underlying concepts. Thus, let us define the most important parts of it. In such an architecture, we will encounter the following elements:

- **VM**: Virtual machine; this corresponds to a virtual computer. We can run many VMs on a single physical computer (often called a server).
- **Node**: An individual server used to run applications. It can be either a physical or a virtual server. The former often is called a bare-metal server, and the latter a VM. Nodes can be anything from a Raspberry PI to a super computer. Nodes can run in the cloud, in a company's own data-center, or even under your desk.
- **Cluster**: A cluster is a group of nodes connected by a network. A cluster is used to run distributed applications.
- **Network**: Physical or software-defined paths between nodes over which the latter and the programs that run on them communicate. The nodes of a cluster are connected to each other by one (or several) network(s).

- **Port:** This is a channel on which an application, such as a Web server or TCP host, is listening for incoming requests.
- **Service**: Unfortunately, the word service is a very overloaded term, and without any further context, given it is not really clear what the meaning of service is. If we use the term service in the context of an application (we may then call it an application service), it normally means that it is a piece of software that offers a limited set of functionalities to other parts of the application. But be aware that in the remains of this chapter, we will use service in other contexts too, where it will have a different meaning.

A distributed application architecture is the opposite of a monolithic application architecture. In the latter, the whole application consists of a single monolithic block that has always to be deployed and run as a single piece. Traditionally, businesses have written monolithic applications. The result of this was then deployed to a named server running somewhere in a data-center. All the code of this application has been compiled into a single binary or a few highly coupled binaries that needed to be collocated when running the application. It is important to note that the server on which the application runs has a well-defined name or static IP address. Let us illustrate this with a little diagram:

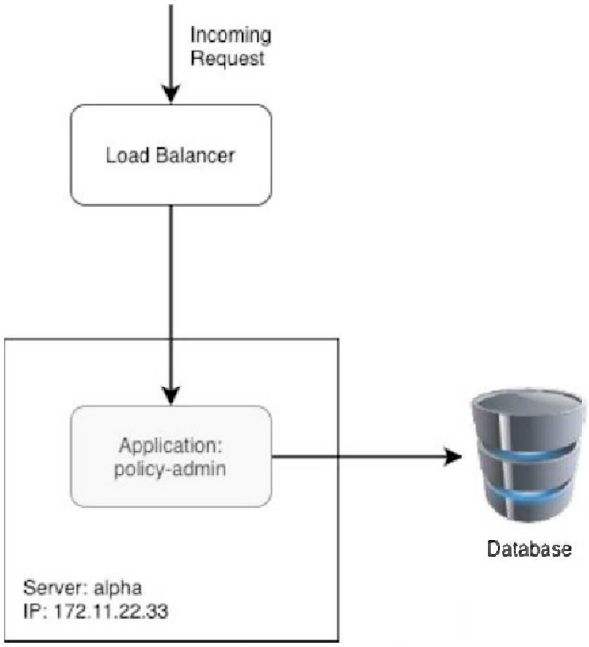


Figure 10.1: Monolithic application architecture

- **Port:** This is a channel on which an application, such as a Web server or TCP host, is listening for incoming requests.
- **Service**: Unfortunately, the word service is a very overloaded term, and without any further context, given it is not really clear what the meaning of service is. If we use the term service in the context of an application (we may then call it an application service), it normally means that it is a piece of software that offers a limited set of functionalities to other parts of the application. But be aware that in the remains of this chapter, we will use service in other contexts too, where it will have a different meaning.

A distributed application architecture is the opposite of a monolithic application architecture. In the latter, the whole application consists of a single monolithic block that has always to be deployed and run as a single piece. Traditionally, businesses have written monolithic applications. The result of this was then deployed to a named server running somewhere in a data-center. All the code of this application has been compiled into a single binary or a few highly coupled binaries that needed to be collocated when running the application. It is important to note that the server on which the application runs has a well-defined name or static IP address. Let us illustrate this with a little diagram:

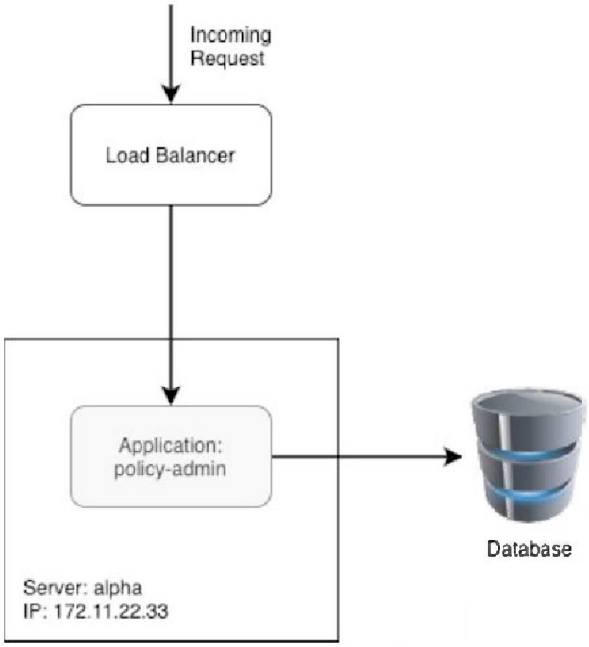


Figure 10.1: Monolithic application architecture

In the preceding graphic, we can identify a server named `alpha` with an IP address of `172.11.22.33` running an application called `policy-admin`, which is a monolith consisting of a main module and a few tightly coupled libraries.

Now, let us look at this next graphic:

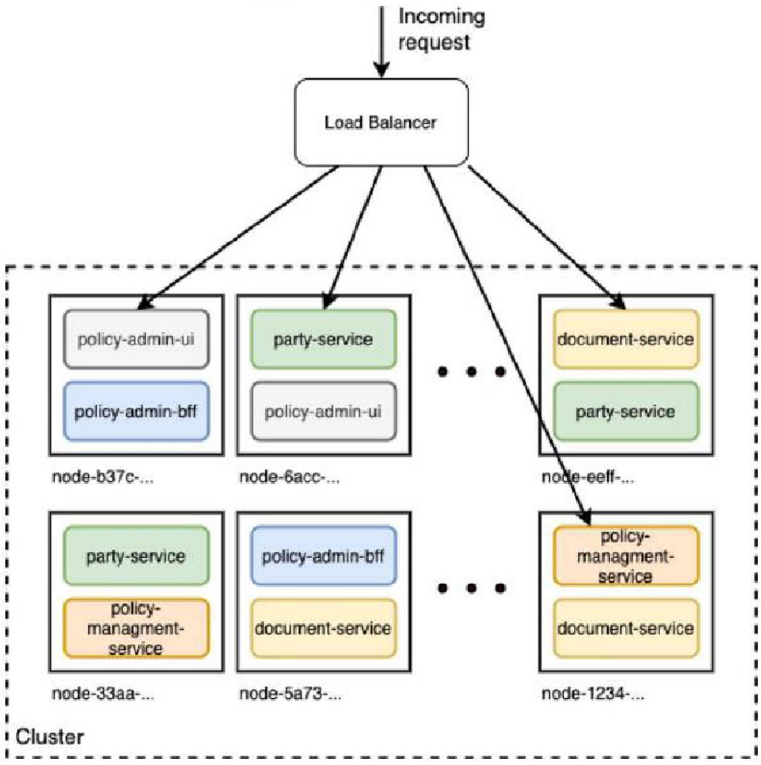


Figure 10.2: Distributed application architecture

In the preceding graphic, we can see that we do not have a single server anymore. Instead, we have a lot of them. Also note that the servers do not have human-friendly names anymore, but instead, some unique identifiers that may be something like a **universal unique identifier** (UUID). The policy-admin application also does not consist of a single monolithic bloc anymore but of a set of interacting yet loosely coupled services such as `policy-admin-ui`, `policy-admin-bff`, `policy-management-service`, `party-service`, `document-service`, and so on. We can also see that each of those services runs on multiple instances in this cluster of servers.

In the preceding graphic, we can identify a server named `alpha` with an IP address of `172.11.22.33` running an application called `policy-admin`, which is a monolith consisting of a main module and a few tightly coupled libraries.

Now, let us look at this next graphic:

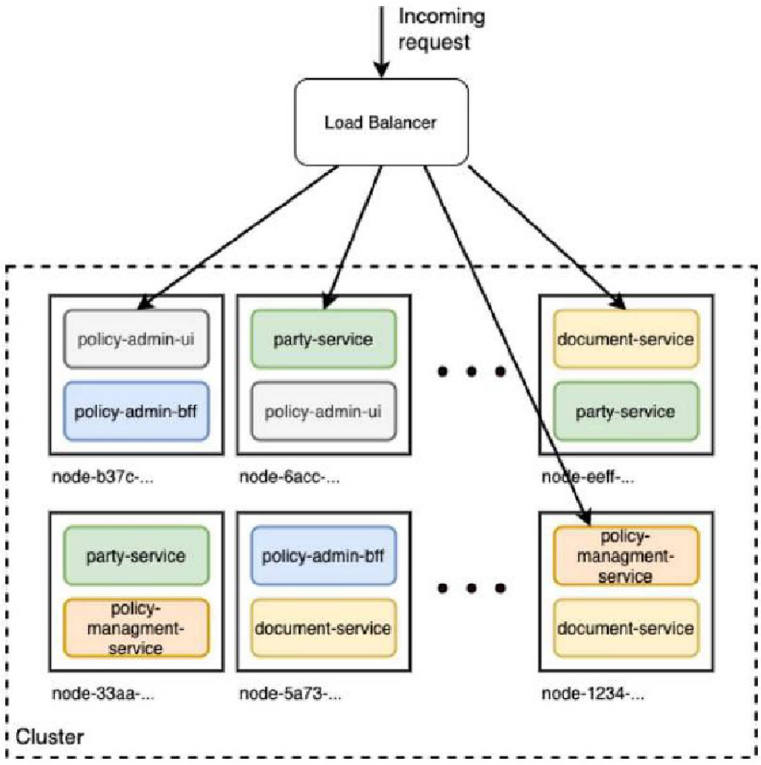


Figure 10.2: Distributed application architecture

In the preceding graphic, we can see that we do not have a single server anymore. Instead, we have a lot of them. Also note that the servers do not have human-friendly names anymore, but instead, some unique identifiers that may be something like a **universal unique identifier** (UUID). The policy-admin application also does not consist of a single monolithic bloc anymore but of a set of interacting yet loosely coupled services such as `policy-admin-ui`, `policy-admin-bff`, `policy-management-service`, `party-service`, `document-service`, and so on. We can also see that each of those services runs on multiple instances in this cluster of servers.

If you are now confused and are wondering why we discuss this in a book about Docker, well, let me say this, it is good that you asked. While all the topics we are discussing here apply to a world where containers did not (yet) exist, it is important to realize that the introduction of containers and, with it, of container orchestrators has helped to address all the problems mentioned in a more efficient and straight forward manner. Most problems of a distributed application architecture that used to be very hard to solve in a world without containers are now quite simple to address, as you will see.

Patterns and best practices

A distributed application architecture offers many benefits compared to a monolithic application architecture, but it also has a few important drawbacks compared to the latter. The most important drawback is that the distributed application architecture is way more complex. To address this complexity, the community and industry have developed a few best practices and patterns to tackle this beast. Let us discuss a few of them.

Loosely coupling 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 rocket, such as the SpaceX Starship, in one single step. It is much easier to build the rockt 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 only loosely coupled to 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.

Preferring stateless over stateful components

Every business application creates, modifies, and displays some form of data. This can be data about a customer, a product, and more. In IT, data is also called state.

If you are now confused and are wondering why we discuss this in a book about Docker, well, let me say this, it is good that you asked. While all the topics we are discussing here apply to a world where containers did not (yet) exist, it is important to realize that the introduction of containers and, with it, of container orchestrators has helped to address all the problems mentioned in a more efficient and straight forward manner. Most problems of a distributed application architecture that used to be very hard to solve in a world without containers are now quite simple to address, as you will see.

Patterns and best practices

A distributed application architecture offers many benefits compared to a monolithic application architecture, but it also has a few important drawbacks compared to the latter. The most important drawback is that the distributed application architecture is way more complex. To address this complexity, the community and industry have developed a few best practices and patterns to tackle this beast. Let us discuss a few of them.

Loosely coupling 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 rocket, such as the SpaceX Starship, in one single step. It is much easier to build the rockt 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 only loosely coupled to 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.

Preferring stateless over stateful components

Every business application creates, modifies, and displays some form of data. This can be data about a customer, a product, and more. In IT, data is also called state.

In that sense, an application produces and manages the state. A component of the overall application that creates or modifies persistent data is called a stateful component. Typical stateful components or services are the classical databases (SQL or no-SQL) or services that create and manage files.

Components that do not create or modify data are called stateless components.

Stateless components are very easy to deal with in a distributed application architecture. Those components can be easily scaled up or down, or they can be torn down and restarted on a different node of the cluster. This is all because they do not have any associated state which would have to be taken care of.

Stateful components are much harder to manage in a distributed system. The state has to be managed, and the more state we have or the more quickly state changes happen, the harder it is to cope with failing components.

Considering that, we recommend to design your systems in a way that most application services are stateless. Push the stateful components to the boundary of the system and limit their number. Try to use stateful services off the shelf, such as databases. Let us state it once again: managing stateful components is hard.

Using service discovery

When we build a system consisting of many individual parts that have to communicate with each other in some or the other way, then we need a mechanism that allows the individual components in the system to find each other. Since the components are distributed in a cluster, this normally means that this mechanism needs to provide the calling component with the address of the target component. This address will consist of the node on which the target component runs as well as the port on which it is listening. Normally, we identify nodes by an IP address. The port is just a number in a well-defined range.

In theory, each service could be configured with a list of addresses of all other components running in the system. This could, for example, happen by the use of a configuration file as shown in the example as follows:

In that sense, an application produces and manages the state. A component of the overall application that creates or modifies persistent data is called a stateful component. Typical stateful components or services are the classical databases (SQL or no-SQL) or services that create and manage files.

Components that do not create or modify data are called stateless components.

Stateless components are very easy to deal with in a distributed application architecture. Those components can be easily scaled up or down, or they can be torn down and restarted on a different node of the cluster. This is all because they do not have any associated state which would have to be taken care of.

Stateful components are much harder to manage in a distributed system. The state has to be managed, and the more state we have or the more quickly state changes happen, the harder it is to cope with failing components.

Considering that, we recommend to design your systems in a way that most application services are stateless. Push the stateful components to the boundary of the system and limit their number. Try to use stateful services off the shelf, such as databases. Let us state it once again: managing stateful components is hard.

Using service discovery

When we build a system consisting of many individual parts that have to communicate with each other in some or the other way, then we need a mechanism that allows the individual components in the system to find each other. Since the components are distributed in a cluster, this normally means that this mechanism needs to provide the calling component with the address of the target component. This address will consist of the node on which the target component runs as well as the port on which it is listening. Normally, we identify nodes by an IP address. The port is just a number in a well-defined range.

In theory, each service could be configured with a list of addresses of all other components running in the system. This could, for example, happen by the use of a configuration file as shown in the example as follows:

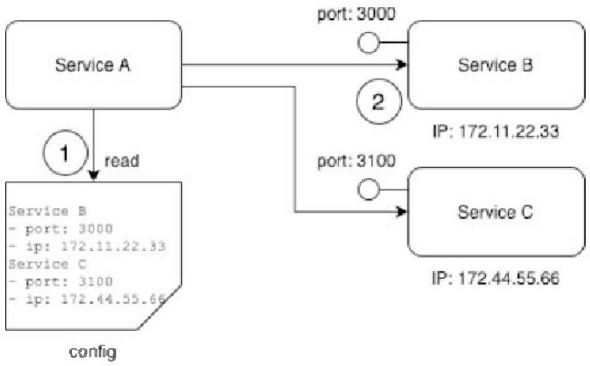


Figure 10.3: Components are hand wired

This may work well in the context of a monolithic application that runs on one or a handful of well-known and curated servers, but it miserably fails in highly distributed applications architecture with its many moving parts. To keep track of the ever-changing addresses of components quickly becomes a maintenance nightmare. Thus, we need some external mechanism or service that specializes in this. This external service discovery service does nothing else than keeping track of the whereabout of each service in the system. Often such a kind of service is called a **domain name system (DNS)**. Docker, for example, provides such a DNS service out of the box, and also Kubernetes offers a DNS service to facilitate communication between the individual components running in the system.

Routing calls between components

Routing is the mechanism of sending data packets from a source component to a target component. From the **Open Systems Interconnection (OSI)** model, we can derive different types or levels of routing. In our context, routing at Levels 2, 3, 4, and 7 are important. Layer 2 routing connects two MAC addresses to each other whilst Layer 7 routing—also called application-level routing—uses URLs such as **https://example.com/icsm/policy-admin** to route a request from a party to the appropriate target component in our system.

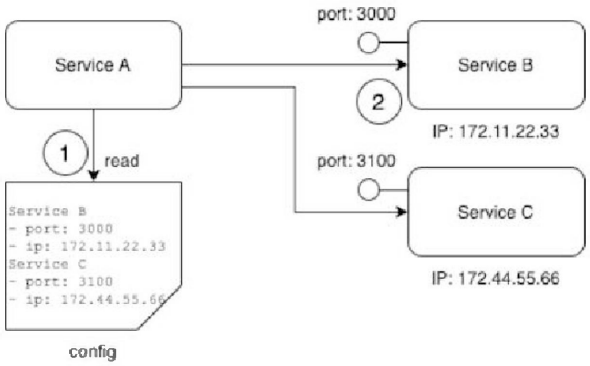


Figure 10.3: Components are hand wired

This may work well in the context of a monolithic application that runs on one or a handful of well-known and curated servers, but it miserably fails in highly distributed applications architecture with its many moving parts. To keep track of the ever-changing addresses of components quickly becomes a maintenance nightmare. Thus, we need some external mechanism or service that specializes in this. This external service discovery service does nothing else than keeping track of the whereabout of each service in the system. Often such a kind of service is called a **domain name system (DNS)**. Docker, for example, provides such a DNS service out of the box, and also Kubernetes offers a DNS service to facilitate communication between the individual components running in the system.

Routing calls between components

Routing is the mechanism of sending data packets from a source component to a target component. From the **Open Systems Interconnection (OSI)** model, we can derive different types or levels of routing. In our context, routing at Levels 2, 3, 4, and 7 are important. Layer 2 routing connects two MAC addresses to each other whilst Layer 7 routing—also called application-level routing—uses URLs such as **https://example.com/icsm/policy-admin** to route a request from a party to the appropriate target component in our system.

Load balancing traffic

We need load balancing every time when a service called `foo` needs to communicate with a service called `bar` in a request-response pattern and where the service `bar` runs in more than one instance. This can be better illustrated with the following graphic:

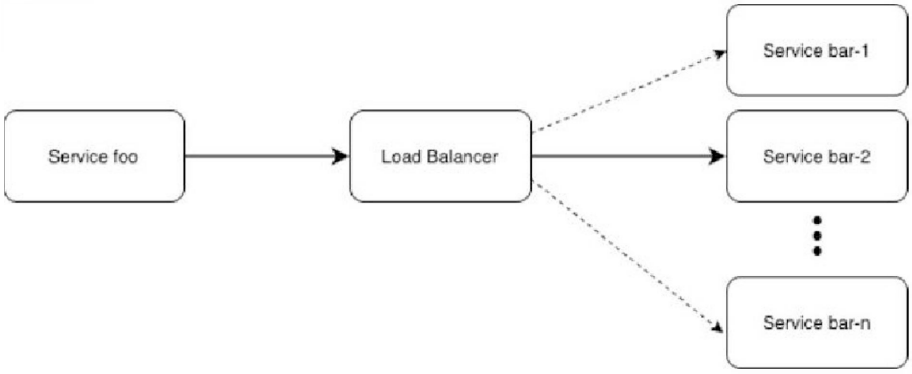


Figure 10.4: Requests of service foo are load balanced to the service bar

As you can see previously, we have multiple instances `bar-1`, `bar-2`, ... `bar-n` of the service `bar` running. We want to make sure that each of these instances is receiving the same load, and thus, we need to somehow distribute the work load evenly among them. For this, we use a load balancer. The load balancer knows how many instances of the service `bar` there are and whether they are healthy or not. The load balancer can use different algorithms to decide how to route the traffic. One of the most popular and simple algorithms is the so-called round-robin method, where each call is forwarded to the next instance in the set in a cyclic way. First instance, `bar-1` is served, then `bar-2`, then `bar-3`, and so on. After instance, *n* has been served, it is again the turn-on instance `bar-1`.

In the sample, we have illustrated the load balancer also facilitates high availability since a request from service `foo` will only be forwarded to a healthy instance of the service `bar`. The load balancer periodically checks the health of each instance of the service `bar`.

Programming defensively

In a highly distributed application architecture, we always need to be prepared for the worst. And thus, we have to implement code accordingly. We need to be very cautious and expect unexpected results and behavior all the time. To tackle this, here are a few recommendations:

Load balancing traffic

We need load balancing every time when a service called `foo` needs to communicate with a service called `bar` in a request-response pattern and where the service `bar` runs in more than one instance. This can be better illustrated with the following graphic:

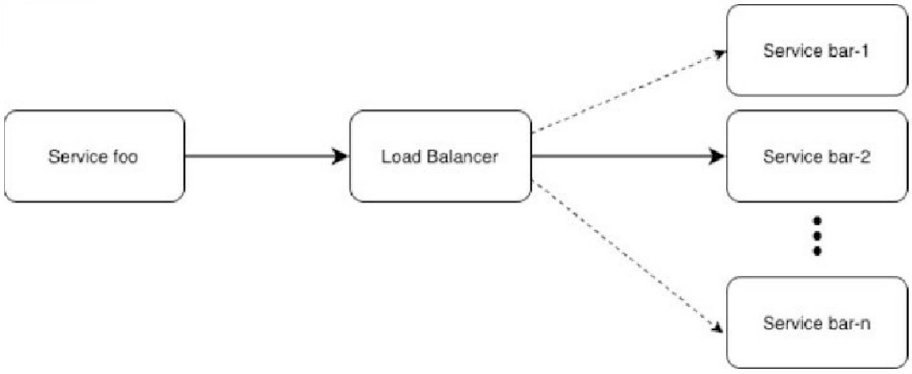


Figure 10.4: Requests of service foo are load balanced to the service bar

As you can see previously, we have multiple instances `bar-1`, `bar-2`, ... `bar-n` of the service `bar` running. We want to make sure that each of these instances is receiving the same load, and thus, we need to somehow distribute the work load evenly among them. For this, we use a load balancer. The load balancer knows how many instances of the service `bar` there are and whether they are healthy or not. The load balancer can use different algorithms to decide how to route the traffic. One of the most popular and simple algorithms is the so-called round-robin method, where each call is forwarded to the next instance in the set in a cyclic way. First instance, `bar-1` is served, then `bar-2`, then `bar-3`, and so on. After instance, *n* has been served, it is again the turn-on instance `bar-1`.

In the sample, we have illustrated the load balancer also facilitates high availability since a request from service `foo` will only be forwarded to a healthy instance of the service `bar`. The load balancer periodically checks the health of each instance of the service `bar`.

Programming defensively

In a highly distributed application architecture, we always need to be prepared for the worst. And thus, we have to implement code accordingly. We need to be very cautious and expect unexpected results and behavior all the time. To tackle this, here are a few recommendations:

- **Retries**: When there is even a slight possibility that an external service might become temporarily or permanently unavailable, then we should react accordingly. For this, we can use the following approach: each call to this external service should be repeated multiple times in case it fails. How many times it should be repeated needs to be discussed with the business and potentially the owner of the external service. But make sure you are not looping forever. Start with a reasonable maximum number of retries, say 10. Also, make sure not to repeat the call without an ever-increasing wait time between each call.
- **Logging**: For each important operation that your code performs, specifically when accessing external services over which you have no control, make sure to produce logging information. Though this logging information needs to be categorized, e.g., with log levels such as debug, info, warning, and error. Collect all the log output produced centrally where you can then easily filter and parse for relevant information. This information is crucial when you want to quickly pinpoint the root cause of unexpected behavior or even failures of the application running in production.
- **Error handling**: One rule is to always fail fast. If you discover an irregularity, such as missing yet required input data, your code should fail immediately and not try to somehow continue (and hope for the best ...). Do not forget to log crucial information in case of such a failure because it may help you and your colleagues to quickly pinpoint the root cause of the failure.

Guaranteeing redundancy

Business applications are often mission-critical. What this means is that the application has to be available all the time, around the clock, seven days a week. No downtime is acceptable, and if there is downtime, it often results in reputational and financial damage. In an application that is highly distributed, there is a big chance that one or the other component of the application is failing. This can be a component of the application itself or a piece of the infrastructure it runs on. The question in such an architecture is never if it will fail but rather when it will fail.

To avoid downtime in case of a component failing, each component in the system (hardware or software) has to be redundant. The easiest way of guaranteeing this is by running each component redundantly. Once again, we want to point out that this applies to software components equally that to hardware components, say, an edge router, a server on which a node runs, or a physical network cable. To avoid downtime, everything needs to be redundant.

Now that we have a rough understanding of what a distributed application architecture is and what its challenges are, we are ready to ask ourselves how

- **Retries**: When there is even a slight possibility that an external service might become temporarily or permanently unavailable, then we should react accordingly. For this, we can use the following approach: each call to this external service should be repeated multiple times in case it fails. How many times it should be repeated needs to be discussed with the business and potentially the owner of the external service. But make sure you are not looping forever. Start with a reasonable maximum number of retries, say 10. Also, make sure not to repeat the call without an ever-increasing wait time between each call.
- **Logging**: For each important operation that your code performs, specifically when accessing external services over which you have no control, make sure to produce logging information. Though this logging information needs to be categorized, e.g., with log levels such as debug, info, warning, and error. Collect all the log output produced centrally where you can then easily filter and parse for relevant information. This information is crucial when you want to quickly pinpoint the root cause of unexpected behavior or even failures of the application running in production.
- **Error handling**: One rule is to always fail fast. If you discover an irregularity, such as missing yet required input data, your code should fail immediately and not try to somehow continue (and hope for the best ...). Do not forget to log crucial information in case of such a failure because it may help you and your colleagues to quickly pinpoint the root cause of the failure.

Guaranteeing redundancy

Business applications are often mission-critical. What this means is that the application has to be available all the time, around the clock, seven days a week. No downtime is acceptable, and if there is downtime, it often results in reputational and financial damage. In an application that is highly distributed, there is a big chance that one or the other component of the application is failing. This can be a component of the application itself or a piece of the infrastructure it runs on. The question in such an architecture is never if it will fail but rather when it will fail.

To avoid downtime in case of a component failing, each component in the system (hardware or software) has to be redundant. The easiest way of guaranteeing this is by running each component redundantly. Once again, we want to point out that this applies to software components equally that to hardware components, say, an edge router, a server on which a node runs, or a physical network cable. To avoid downtime, everything needs to be redundant.

Now that we have a rough understanding of what a distributed application architecture is and what its challenges are, we are ready to ask ourselves how

the challenges mentioned are solved in a world using containers. Let us start by discussing what a container orchestrator is.

What is an orchestrator?

In the section about preceding distributed application architecture, we learned which patterns and best practices are commonly used to build, ship, and run a highly distributed and mission-critical application. If we containerize the components of our distributed application, we are facing the same problems that the non-containerized version faces. Thus, we would really benefit if we had a tool of infrastructure that helped us with all those challenges mentioned, such as service discovery, load balancing, and scaling, to just name a few.

Such a piece of software exists and is commonly called an orchestration engine. The orchestration engine has similarity to a conductor in an orchestra. Although every single artist in the orchestra knows very well how to play his or her instrument, they all need someone that coordinates their individual efforts, such as that the total of all instruments and voices results in that harmonious music that we love to listen to. Without a conductor, we would probably rather hear a cacophony of individual sounds.

Analogously, the orchestration engine or short, the orchestrator has to execute very specific tasks to make sure all containers in a cluster play with each other in harmony.

The responsibilities of an orchestrator

What are the tasks that the orchestrator has to execute to keep all containers at bay? Let us look at them in detail. The following list discusses the most important ones.

Reconciling the desired state

When using an orchestrator, you tell it, in a declarative way, how you want it to run a given application or application service. We learned what declarative versus imperative means in *Chapter 7, Managing Complex Apps with Docker Compose*. Part of this declarative way of describing the application service that we want to run includes elements such as which container image to use, how many instances of this service to run, which ports to open, and more. This declaration of the properties of our application service is what we call **the desired state**.

So, when we now tell the orchestrator for the first time to create such a new application service based on the declaration, then the orchestrator makes sure to schedule as many containers in the cluster as requested. If the container image is not yet available on the target nodes of the cluster where the containers are supposed to run, then the scheduler makes sure that they are first downloaded from the image

the challenges mentioned are solved in a world using containers. Let us start by discussing what a container orchestrator is.

What is an orchestrator?

In the section about preceding distributed application architecture, we learned which patterns and best practices are commonly used to build, ship, and run a highly distributed and mission-critical application. If we containerize the components of our distributed application, we are facing the same problems that the non-containerized version faces. Thus, we would really benefit if we had a tool of infrastructure that helped us with all those challenges mentioned, such as service discovery, load balancing, and scaling, to just name a few.

Such a piece of software exists and is commonly called an orchestration engine. The orchestration engine has similarity to a conductor in an orchestra. Although every single artist in the orchestra knows very well how to play his or her instrument, they all need someone that coordinates their individual efforts, such as that the total of all instruments and voices results in that harmonious music that we love to listen to. Without a conductor, we would probably rather hear a cacophony of individual sounds.

Analogously, the orchestration engine or short, the orchestrator has to execute very specific tasks to make sure all containers in a cluster play with each other in harmony.

The responsibilities of an orchestrator

What are the tasks that the orchestrator has to execute to keep all containers at bay? Let us look at them in detail. The following list discusses the most important ones.

Reconciling the desired state

When using an orchestrator, you tell it, in a declarative way, how you want it to run a given application or application service. We learned what declarative versus imperative means in *Chapter 7, Managing Complex Apps with Docker Compose*. Part of this declarative way of describing the application service that we want to run includes elements such as which container image to use, how many instances of this service to run, which ports to open, and more. This declaration of the properties of our application service is what we call **the desired state**.

So, when we now tell the orchestrator for the first time to create such a new application service based on the declaration, then the orchestrator makes sure to schedule as many containers in the cluster as requested. If the container image is not yet available on the target nodes of the cluster where the containers are supposed to run, then the scheduler makes sure that they are first downloaded from the image

registry. Next, the containers are started with all the settings, such as networks to which to attach or ports to expose. The orchestrator works as hard as it can to exactly match the cluster state to the declaration.

Once our service is up and running as requested, that is, it is running in the desired state, then the orchestrator continues to monitor it. Each time the orchestrator discovers a discrepancy between the actual state of the service and its desired state, it again tries its best to reconcile the desired state.

What could such a discrepancy between the actual and desired states of an application service be? Well, let us say one of the replicas of the service, that is, one of the containers, crashes due to, say, a bug, then the orchestrator will discover that the actual state differs from the desired state in the number of replicas: there is one replica missing. The orchestrator will immediately schedule a new instance to another cluster node, which replaces the crashed instance. Another discrepancy could be that there are too many instances of the application service running if the service has been scaled down. In this case, the orchestrator will just randomly kill as many instances as needed in order to achieve parity between the actual and the desired number of instances. Yet another discrepancy could be when the orchestrator discovers that there is an instance of the application service running a wrong (maybe old) version of the underlying container image. By now, you should get the picture, right?

Thus, instead of us actively monitoring our application's services that are running in the cluster and correcting any deviation from the desired state, we delegate this tedious task to the orchestrator. This works very well, provided we use a declarative and not an imperative way of describing the desired state of our application services.

Replicating and global services

There are two quite different types of services that we might want to run in a cluster that is managed by an orchestrator. They are replicated and global services. A replicated service is a service that is required to run in a specific number of instances, say 10. A global service, in turn, is a service that is required to have exactly one instance running on every single worker node of the cluster. I have used the term worker node here. In a cluster that is managed by an orchestrator, we typically have two types of nodes, managers, and workers. A manager node is usually exclusively used by the orchestrator to manage the cluster and does not run any other workload. Worker nodes, in turn, run the actual applications.

So, the orchestrator makes sure that, for a global service, an instance of it is running on every single worker node, no matter how many there are. We do not need to care about the number of instances, but only that on each node, it is guaranteed to run a single instance of the service.

registry. Next, the containers are started with all the settings, such as networks to which to attach or ports to expose. The orchestrator works as hard as it can to exactly match the cluster state to the declaration.

Once our service is up and running as requested, that is, it is running in the desired state, then the orchestrator continues to monitor it. Each time the orchestrator discovers a discrepancy between the actual state of the service and its desired state, it again tries its best to reconcile the desired state.

What could such a discrepancy between the actual and desired states of an application service be? Well, let us say one of the replicas of the service, that is, one of the containers, crashes due to, say, a bug, then the orchestrator will discover that the actual state differs from the desired state in the number of replicas: there is one replica missing. The orchestrator will immediately schedule a new instance to another cluster node, which replaces the crashed instance. Another discrepancy could be that there are too many instances of the application service running if the service has been scaled down. In this case, the orchestrator will just randomly kill as many instances as needed in order to achieve parity between the actual and the desired number of instances. Yet another discrepancy could be when the orchestrator discovers that there is an instance of the application service running a wrong (maybe old) version of the underlying container image. By now, you should get the picture, right?

Thus, instead of us actively monitoring our application's services that are running in the cluster and correcting any deviation from the desired state, we delegate this tedious task to the orchestrator. This works very well, provided we use a declarative and not an imperative way of describing the desired state of our application services.

Replicating and global services

There are two quite different types of services that we might want to run in a cluster that is managed by an orchestrator. They are replicated and global services. A replicated service is a service that is required to run in a specific number of instances, say 10. A global service, in turn, is a service that is required to have exactly one instance running on every single worker node of the cluster. I have used the term worker node here. In a cluster that is managed by an orchestrator, we typically have two types of nodes, managers, and workers. A manager node is usually exclusively used by the orchestrator to manage the cluster and does not run any other workload. Worker nodes, in turn, run the actual applications.

So, the orchestrator makes sure that, for a global service, an instance of it is running on every single worker node, no matter how many there are. We do not need to care about the number of instances, but only that on each node, it is guaranteed to run a single instance of the service.

Enabling service discovery

When we describe an application service in a declarative way, we are never supposed to tell the orchestrator on which cluster nodes the different instances of the service have to run. We leave it up to the orchestrator to decide which nodes best fit this task.

It is, of course, technically possible to instruct the orchestrator to use very deterministic placement rules, but this would be an anti-pattern and is not recommended at all other than in very special edge cases.

So, if we now assume that the orchestration engine has complete and free will as to where to place individual instances of the application service and, furthermore, that instances can crash and be rescheduled by the orchestrator to different nodes, then we will realize that it is a futile task for us to keep track of where the individual instances are running at any given time. Even better, we should not even try to know this because it is not important.

OK, you might say, but what about if I have two services, A and B, and Service A relies on Service B; should not any given instance of Service A know where it can find an instance of Service B?

Here, I have to say loudly and clearly—no, it should not. This kind of knowledge is not desirable in a highly distributed and scalable application. Rather, we should rely on the orchestrator to provide us with the information that we need in order to reach the other service instances that we depend on. It is a bit like in the old days of telephony when we could not directly call our friends but had to call the phone company's central office, where some operator would then route us to the correct destination. In our case, the orchestrator plays the role of the operator, routing a request coming from an instance of Service A to an available instance of Service B. This whole process is called service discovery.

Providing routing services

We have learned so far that in a distributed application, we have many interacting services. When Service A interacts with Service B, it happens through the exchange of data packets. These data packets need to somehow be funneled from Service A to Service B. This process of funneling the data packets from a source to a destination is also called routing. Routing can happen on different levels. It is like in real life. Suppose you are working in a big company in one of their office buildings. Now, you have a document that needs to be forwarded to another employee of the company. The internal post service will pick up the document from your outbox and take it to the post office located in the same building. If the target person works in the same building, the document can then be directly forwarded to that person. If, on the other hand, the person works in another building of the same block, the document

Enabling service discovery

When we describe an application service in a declarative way, we are never supposed to tell the orchestrator on which cluster nodes the different instances of the service have to run. We leave it up to the orchestrator to decide which nodes best fit this task.

It is, of course, technically possible to instruct the orchestrator to use very deterministic placement rules, but this would be an anti-pattern and is not recommended at all other than in very special edge cases.

So, if we now assume that the orchestration engine has complete and free will as to where to place individual instances of the application service and, furthermore, that instances can crash and be rescheduled by the orchestrator to different nodes, then we will realize that it is a futile task for us to keep track of where the individual instances are running at any given time. Even better, we should not even try to know this because it is not important.

OK, you might say, but what about if I have two services, A and B, and Service A relies on Service B; should not any given instance of Service A know where it can find an instance of Service B?

Here, I have to say loudly and clearly—no, it should not. This kind of knowledge is not desirable in a highly distributed and scalable application. Rather, we should rely on the orchestrator to provide us with the information that we need in order to reach the other service instances that we depend on. It is a bit like in the old days of telephony when we could not directly call our friends but had to call the phone company's central office, where some operator would then route us to the correct destination. In our case, the orchestrator plays the role of the operator, routing a request coming from an instance of Service A to an available instance of Service B. This whole process is called service discovery.

Providing routing services

We have learned so far that in a distributed application, we have many interacting services. When Service A interacts with Service B, it happens through the exchange of data packets. These data packets need to somehow be funneled from Service A to Service B. This process of funneling the data packets from a source to a destination is also called routing. Routing can happen on different levels. It is like in real life. Suppose you are working in a big company in one of their office buildings. Now, you have a document that needs to be forwarded to another employee of the company. The internal post service will pick up the document from your outbox and take it to the post office located in the same building. If the target person works in the same building, the document can then be directly forwarded to that person. If, on the other hand, the person works in another building of the same block, the document

will be forwarded to the post office in that target building, from where it is then distributed to the receiver through the internal post service. Third, if the document is targeted at an employee working in another branch of the company that is located in a different city or even country, then, the document is forwarded to an external postal service such as UPS, which will transport it to the target location, from where, once again, the internal post service takes over and delivers it to the recipient.

Similar things happen when routing data packets between application services that are running in containers. The source and target containers can be located on the same cluster node, which corresponds to the situation where both employees work in the same building. The target container can be running on a different cluster node, which corresponds to the situation where the two employees work in different buildings of the same block. Finally, the third situation is when a data packet comes from outside of the cluster and has to be routed to the target container that is running inside the cluster. All these situations, and more, have to be handled by the orchestrator.

Distributing work by load balancing

In a highly available distributed application, all components have to be redundant. That means that every application service has to be run in multiple instances so that if one instance fails, the service as a whole is still operational.

To make sure that all instances of a service are actually doing work and are not just sitting around idle, you have to make sure that the requests for service are distributed equally to all the instances. This process of distributing the workload among service instances is

called load balancing. Various algorithms exist for how the workload can be distributed. Usually, a load balancer works using the so-called round-robin algorithm, which makes sure that the workload is distributed equally to the instances using a cyclic algorithm.

Once again, we expect the orchestrator to take care of the load-balancing requests from one service to another or from external sources to internal services.

Dealing with fluctuating load by scaling

When running our containerized, distributed application in a cluster that is managed by an orchestrator, we additionally want an easy way to handle expected or unexpected increases in workload. To handle an increased workload, we usually just schedule additional instances of a service that is experiencing this increased load. Load balancers will then automatically be configured to distribute the workload over more available target instances.

will be forwarded to the post office in that target building, from where it is then distributed to the receiver through the internal post service. Third, if the document is targeted at an employee working in another branch of the company that is located in a different city or even country, then, the document is forwarded to an external postal service such as UPS, which will transport it to the target location, from where, once again, the internal post service takes over and delivers it to the recipient.

Similar things happen when routing data packets between application services that are running in containers. The source and target containers can be located on the same cluster node, which corresponds to the situation where both employees work in the same building. The target container can be running on a different cluster node, which corresponds to the situation where the two employees work in different buildings of the same block. Finally, the third situation is when a data packet comes from outside of the cluster and has to be routed to the target container that is running inside the cluster. All these situations, and more, have to be handled by the orchestrator.

Distributing work by load balancing

In a highly available distributed application, all components have to be redundant. That means that every application service has to be run in multiple instances so that if one instance fails, the service as a whole is still operational.

To make sure that all instances of a service are actually doing work and are not just sitting around idle, you have to make sure that the requests for service are distributed equally to all the instances. This process of distributing the workload among service instances is

called load balancing. Various algorithms exist for how the workload can be distributed. Usually, a load balancer works using the so-called round-robin algorithm, which makes sure that the workload is distributed equally to the instances using a cyclic algorithm.

Once again, we expect the orchestrator to take care of the load-balancing requests from one service to another or from external sources to internal services.

Dealing with fluctuating load by scaling

When running our containerized, distributed application in a cluster that is managed by an orchestrator, we additionally want an easy way to handle expected or unexpected increases in workload. To handle an increased workload, we usually just schedule additional instances of a service that is experiencing this increased load. Load balancers will then automatically be configured to distribute the workload over more available target instances.

But in real-life scenarios, the workload varies over time. If we look at a shopping site such as Amazon, it might have a high load during peak hours in the evening, when everyone is at home and shopping online; it may experience extreme loads during special days such as Black Friday; and it may experience very little traffic early in the morning. Thus, services need to not just be able to scale up but also to scale down when the workload goes down.

We also expect orchestrators to distribute the instances of service in a meaningful way when scaling up or down. It would not be wise to schedule all instances of the service on the same cluster node since if that node goes down, then the whole service goes down. The scheduler of the orchestrator, which is responsible for the placement of the containers, needs to also consider not placing all instances into the same rack of computers since if the power supply of the rack fails, again, the whole service is affected. Furthermore, service instances of critical services should even be distributed across data centers in order to avoid outages. All these decisions, and many more, are the responsibility of the orchestrator.

Please note that in the cloud, instead of computer racks, the term **availability zones** is often used.

Facilitating self-healing

These days, orchestrators are very sophisticated and can do a lot for us to maintain a healthy system. Orchestrators monitor all containers that are running in the cluster, and they automatically replace crashed or unresponsive containers with new instances. Orchestrators monitor the health of cluster nodes and take a node out of the scheduler loop if it becomes unhealthy or is down. A workload that was located on those nodes is automatically rescheduled to different available nodes.

All these activities, where the orchestrator monitors the current state and automatically repairs the damage or reconciles the desired state, resulting in a so-called self-healing system. We do not, in most cases, have to actively engage and repair the damage. The orchestrator will do this for us automatically.

However, there are a few situations that the orchestrator cannot handle without our help. Imagine a situation where we have a service instance running in a container. The container is up and running and, from the outside, looks perfectly healthy. But the application running inside it is in an unhealthy state. The application did not crash, and it just is not able to work anymore as it was originally designed. How could the orchestrator possibly know about this without us giving it a hint? It cannot! Being in an unhealthy or invalid state means something completely different for each application service. In other words, the health status is service dependent. Only the authors of the service, or its operators, know what health means in the context of a service.

But in real-life scenarios, the workload varies over time. If we look at a shopping site such as Amazon, it might have a high load during peak hours in the evening, when everyone is at home and shopping online; it may experience extreme loads during special days such as Black Friday; and it may experience very little traffic early in the morning. Thus, services need to not just be able to scale up but also to scale down when the workload goes down.

We also expect orchestrators to distribute the instances of service in a meaningful way when scaling up or down. It would not be wise to schedule all instances of the service on the same cluster node since if that node goes down, then the whole service goes down. The scheduler of the orchestrator, which is responsible for the placement of the containers, needs to also consider not placing all instances into the same rack of computers since if the power supply of the rack fails, again, the whole service is affected. Furthermore, service instances of critical services should even be distributed across data centers in order to avoid outages. All these decisions, and many more, are the responsibility of the orchestrator.

Please note that in the cloud, instead of computer racks, the term **availability zones** is often used.

Facilitating self-healing

These days, orchestrators are very sophisticated and can do a lot for us to maintain a healthy system. Orchestrators monitor all containers that are running in the cluster, and they automatically replace crashed or unresponsive containers with new instances. Orchestrators monitor the health of cluster nodes and take a node out of the scheduler loop if it becomes unhealthy or is down. A workload that was located on those nodes is automatically rescheduled to different available nodes.

All these activities, where the orchestrator monitors the current state and automatically repairs the damage or reconciles the desired state, resulting in a so-called self-healing system. We do not, in most cases, have to actively engage and repair the damage. The orchestrator will do this for us automatically.

However, there are a few situations that the orchestrator cannot handle without our help. Imagine a situation where we have a service instance running in a container. The container is up and running and, from the outside, looks perfectly healthy. But the application running inside it is in an unhealthy state. The application did not crash, and it just is not able to work anymore as it was originally designed. How could the orchestrator possibly know about this without us giving it a hint? It cannot! Being in an unhealthy or invalid state means something completely different for each application service. In other words, the health status is service dependent. Only the authors of the service, or its operators, know what health means in the context of a service.

Now, orchestrators define seams or probes over which an application service can communicate to the orchestrator about what state it is in. Two fundamental types of probes exist:

- The service can tell the orchestrator whether it is healthy or not
- The service can tell the orchestrator that it is ready or temporarily unavailable

How the service determines either of the preceding answers are totally up to the service. The orchestrator only defines how it is going to ask, for example, through an HTTP GET request, or what type of answers it is expecting, for example, OK or NOT OK.

If our services implement logic in order to answer the preceding health or availability questions, then we have a truly self-healing system since the orchestrator can kill unhealthy service instances and replace them with new healthy ones, and it can take service instances that are temporarily unavailable out of the load balancer's round robin.

Enabling zero-downtime deployments

These days, it gets harder and harder to justify a complete downtime for a mission-critical application that needs to be updated. Not only does that mean missed opportunities, but also it can also result in a damaged reputation for the company. Customers using the application are no longer prepared to accept such an inconvenience and will turn away quickly. Furthermore, our release cycles get shorter and shorter. Where, in the past, we would have one or two new releases per year, these days, a lot of companies update their applications multiple times a week or even multiple times per day.

The solution to that problem is to come up with a zero-downtime application update strategy. The orchestrator needs to be able to update individual application services batch-wise. This is also called rolling updates. At any given time, only one or a few of the total number of instances of a given service are taken down and replaced by the new version of the service. Only if the new instances are operational and do not produce any unexpected errors or show any misbehavior will the next batch of instances be updated. This is repeated until all instances are replaced with their new version. If, for some reason, the update fails, then we expect the orchestrator to automatically roll the updated instances back to their previous version.

Other possible zero downtime deployments are blue-green deployments and canary releases. In both cases, the new version of a service is installed in parallel with the current, active version. But initially, the new version is only accessible internally. Operations can then run smoke tests against the new version, and when the new version seems to be running just fine, then, in the case of a blue-green deployment, the router is switched from the current blue version to the new green version. For some

Now, orchestrators define seams or probes over which an application service can communicate to the orchestrator about what state it is in. Two fundamental types of probes exist:

- The service can tell the orchestrator whether it is healthy or not
- The service can tell the orchestrator that it is ready or temporarily unavailable

How the service determines either of the preceding answers are totally up to the service. The orchestrator only defines how it is going to ask, for example, through an HTTP GET request, or what type of answers it is expecting, for example, OK or NOT OK.

If our services implement logic in order to answer the preceding health or availability questions, then we have a truly self-healing system since the orchestrator can kill unhealthy service instances and replace them with new healthy ones, and it can take service instances that are temporarily unavailable out of the load balancer's round robin.

Enabling zero-downtime deployments

These days, it gets harder and harder to justify a complete downtime for a mission-critical application that needs to be updated. Not only does that mean missed opportunities, but also it can also result in a damaged reputation for the company. Customers using the application are no longer prepared to accept such an inconvenience and will turn away quickly. Furthermore, our release cycles get shorter and shorter. Where, in the past, we would have one or two new releases per year, these days, a lot of companies update their applications multiple times a week or even multiple times per day.

The solution to that problem is to come up with a zero-downtime application update strategy. The orchestrator needs to be able to update individual application services batch-wise. This is also called rolling updates. At any given time, only one or a few of the total number of instances of a given service are taken down and replaced by the new version of the service. Only if the new instances are operational and do not produce any unexpected errors or show any misbehavior will the next batch of instances be updated. This is repeated until all instances are replaced with their new version. If, for some reason, the update fails, then we expect the orchestrator to automatically roll the updated instances back to their previous version.

Other possible zero downtime deployments are blue-green deployments and canary releases. In both cases, the new version of a service is installed in parallel with the current, active version. But initially, the new version is only accessible internally. Operations can then run smoke tests against the new version, and when the new version seems to be running just fine, then, in the case of a blue-green deployment, the router is switched from the current blue version to the new green version. For some

time, the new green version of the service is closely monitored, and if everything is fine, the old blue version can be decommissioned. If, on the other hand, the new green version does not work as expected, then it is only a matter of setting the router back to the old blue version in order to achieve a complete rollback.

In the case of a canary release, the router is configured in such a way that it funnels a tiny percentage, say 1%, of the overall traffic through the new version of the service, whereas 99% of the traffic is still routed through the old version. The behavior of the new version is closely monitored and compared to the behavior of the old version. If everything looks good, then the percentage of the traffic that is funneled through the new service is slightly increased. This process is repeated until 100% of the traffic is routed through the new service. If the new service has run for a while and everything looks good, then the old service can be decommissioned.

Most orchestrators support at least the rolling update type of zero downtime deployment out of the box. Blue-green deployments and canary releases are often quite easy to implement.

Providing affinity and location awareness

Sometimes, certain application services require the availability of dedicated hardware on the nodes on which they run. For example, I/O-bound services require cluster nodes with an attached high-performance **solid-state drive (SSD)**, or some services that are used for machine learning or similar require an **Arithmetic Processing Unit (APU)**.

Orchestrators allow us to define node affinities per application service. The orchestrator will then make sure that its scheduler only schedules containers on cluster nodes that fulfill the required criteria.

Defining an affinity to a particular node should be avoided; this would introduce a single point of failure and thus compromise high availability. Always define a set of multiple cluster nodes as the target for an application service.

Some orchestration engines also support what is called location awareness or geo awareness. What this means is that you can ask the orchestrator to equally distribute instances of service over a set of different locations. You could, for example, define a data-center label with the possible west, center, and east values and apply the label to all of the cluster nodes with the value that corresponds to the geographical region in which the respective node is located. Then, you instruct the orchestrator to use this label for the geo awareness of a certain application service. In this case, if you request nine replicas of the service, then the orchestrator would make sure that three instances are deployed to the nodes in each of the three data-centers, west, center, and east.

time, the new green version of the service is closely monitored, and if everything is fine, the old blue version can be decommissioned. If, on the other hand, the new green version does not work as expected, then it is only a matter of setting the router back to the old blue version in order to achieve a complete rollback.

In the case of a canary release, the router is configured in such a way that it funnels a tiny percentage, say 1%, of the overall traffic through the new version of the service, whereas 99% of the traffic is still routed through the old version. The behavior of the new version is closely monitored and compared to the behavior of the old version. If everything looks good, then the percentage of the traffic that is funneled through the new service is slightly increased. This process is repeated until 100% of the traffic is routed through the new service. If the new service has run for a while and everything looks good, then the old service can be decommissioned.

Most orchestrators support at least the rolling update type of zero downtime deployment out of the box. Blue-green deployments and canary releases are often quite easy to implement.

Providing affinity and location awareness

Sometimes, certain application services require the availability of dedicated hardware on the nodes on which they run. For example, I/O-bound services require cluster nodes with an attached high-performance **solid-state drive (SSD)**, or some services that are used for machine learning or similar require an **Arithmetic Processing Unit (APU)**.

Orchestrators allow us to define node affinities per application service. The orchestrator will then make sure that its scheduler only schedules containers on cluster nodes that fulfill the required criteria.

Defining an affinity to a particular node should be avoided; this would introduce a single point of failure and thus compromise high availability. Always define a set of multiple cluster nodes as the target for an application service.

Some orchestration engines also support what is called location awareness or geo awareness. What this means is that you can ask the orchestrator to equally distribute instances of service over a set of different locations. You could, for example, define a data-center label with the possible west, center, and east values and apply the label to all of the cluster nodes with the value that corresponds to the geographical region in which the respective node is located. Then, you instruct the orchestrator to use this label for the geo awareness of a certain application service. In this case, if you request nine replicas of the service, then the orchestrator would make sure that three instances are deployed to the nodes in each of the three data-centers, west, center, and east.

Geo awareness can even be defined hierarchically; for example, you can have a data-center as the top-level discriminator, followed by the availability zone.

Geo awareness, or location awareness, is used to decrease the probability of outages due to power supply failures or data-center outages. If the application instances are distributed across nodes, availability zones, or even data-centers, it is extremely unlikely that everything will go down at once. One region will always be available.

Security

Security is always in focus. Cybercriminals are doing their evil work everywhere and all the time. No company is secure from them. Cybercriminals operate alone, in groups, or are even state-sponsored. North Korea or Iran have been known to be such state sponsors.

It is a nightmare for every CEO, CIO, or CTO to wake up one morning just to hear from the news that their company has been hacked and **personal identifiable information (PII)** of their customers or financial data or company secrets have been stolen.

To avoid such threats, we need to establish a secure software supply chain and enforce a multi-layer security system—also called security in depth.

Here are some measures commonly taken to make a system more secure:

Secure communication and cryptographic node identity

First and foremost, we want to make sure that our cluster, which is managed by the orchestrator, is secure. Only trusted nodes can join the cluster. Each node that joins the cluster gets a cryptographic node identity, and all communication between the nodes must be encrypted. For this, nodes can use **Mutual Transport Layer Security (MTLS)**. In order to authenticate nodes of the cluster with each other, certificates are used. These certificates are automatically rotated periodically or on request to protect the system in case a certificate is leaked.

The communication that happens in a cluster can be separated into three types. You talk about communication planes—management, control, and data planes:

- The management plane is used by the cluster managers, for example, to schedule service instances, execute health checks, or create and modify any other resources in the cluster, such as data volumes, secrets, or networks.
- The control plane is used to exchange important state information between all nodes of the cluster. This kind of information is, for example, used to update the local IP tables on clusters, which are used for routing purposes.

Geo awareness can even be defined hierarchically; for example, you can have a data-center as the top-level discriminator, followed by the availability zone.

Geo awareness, or location awareness, is used to decrease the probability of outages due to power supply failures or data-center outages. If the application instances are distributed across nodes, availability zones, or even data-centers, it is extremely unlikely that everything will go down at once. One region will always be available.

Security

Security is always in focus. Cybercriminals are doing their evil work everywhere and all the time. No company is secure from them. Cybercriminals operate alone, in groups, or are even state-sponsored. North Korea or Iran have been known to be such state sponsors.

It is a nightmare for every CEO, CIO, or CTO to wake up one morning just to hear from the news that their company has been hacked and **personal identifiable information (PII)** of their customers or financial data or company secrets have been stolen.

To avoid such threats, we need to establish a secure software supply chain and enforce a multi-layer security system—also called security in depth.

Here are some measures commonly taken to make a system more secure:

Secure communication and cryptographic node identity

First and foremost, we want to make sure that our cluster, which is managed by the orchestrator, is secure. Only trusted nodes can join the cluster. Each node that joins the cluster gets a cryptographic node identity, and all communication between the nodes must be encrypted. For this, nodes can use **Mutual Transport Layer Security (MTLS)**. In order to authenticate nodes of the cluster with each other, certificates are used. These certificates are automatically rotated periodically or on request to protect the system in case a certificate is leaked.

The communication that happens in a cluster can be separated into three types. You talk about communication planes—management, control, and data planes:

- The management plane is used by the cluster managers, for example, to schedule service instances, execute health checks, or create and modify any other resources in the cluster, such as data volumes, secrets, or networks.
- The control plane is used to exchange important state information between all nodes of the cluster. This kind of information is, for example, used to update the local IP tables on clusters, which are used for routing purposes.

- The data plane is where the actual application services communicate with each other and exchange data.

Normally, orchestrators mainly care about securing the management and control plane. Securing the data plane is left to the user, although the orchestrator may facilitate this task.

Secure networks and network policies

When running application services, not every service needs to communicate with every other service in the cluster. Thus, we want the ability to sandbox services from each other and only run those services in the same networking sandbox that absolutely needs to communicate with each other. All other services and all network traffic coming from outside of the cluster should have no possibility of accessing the sandboxed services.

There are at least two ways in which this network-based sandboxing can happen. Either we can use a **software-defined network** (**SDN**) to group application services, or we can have one flat network and use network policies to control who does and does not have access to a particular service or group of services.

Role base access control (RBAC)

One of the most important tasks (next to security) that an orchestrator must fulfill in order to make it enterprise-ready is to provide role-based access to the cluster and its resources. RBAC defines how subjects, users, or groups of users of the system, organized into teams, and so on, can access and manipulate the system. It makes sure that unauthorized personnel cannot do any harm to the system, nor can they see any of the available resources in the system that they're not supposed to know of or see.

One way of implementing RBAC is through the definition of grants. A grant is an association between a subject, a role, and a resource collection. Here, a role is comprised of a set of access permissions to a resource. Such permissions can be to create, stop, remove, list, or view containers; to deploy a new application service; to list cluster nodes or view the details of a cluster node; and many more.

A resource collection is a group of logically related resources of the cluster, such as application services, secrets, data volumes, or containers.

Secrets

In our daily life, we have loads of secrets. Secrets are information that is not meant to be publicly known, such as the username and password combination that you use to access your online bank account or the code to your cell phone or your locker at the school or office.

- The data plane is where the actual application services communicate with each other and exchange data.

Normally, orchestrators mainly care about securing the management and control plane. Securing the data plane is left to the user, although the orchestrator may facilitate this task.

Secure networks and network policies

When running application services, not every service needs to communicate with every other service in the cluster. Thus, we want the ability to sandbox services from each other and only run those services in the same networking sandbox that absolutely needs to communicate with each other. All other services and all network traffic coming from outside of the cluster should have no possibility of accessing the sandboxed services.

There are at least two ways in which this network-based sandboxing can happen. Either we can use a **software-defined network** (**SDN**) to group application services, or we can have one flat network and use network policies to control who does and does not have access to a particular service or group of services.

Role base access control (RBAC)

One of the most important tasks (next to security) that an orchestrator must fulfill in order to make it enterprise-ready is to provide role-based access to the cluster and its resources. RBAC defines how subjects, users, or groups of users of the system, organized into teams, and so on, can access and manipulate the system. It makes sure that unauthorized personnel cannot do any harm to the system, nor can they see any of the available resources in the system that they're not supposed to know of or see.

One way of implementing RBAC is through the definition of grants. A grant is an association between a subject, a role, and a resource collection. Here, a role is comprised of a set of access permissions to a resource. Such permissions can be to create, stop, remove, list, or view containers; to deploy a new application service; to list cluster nodes or view the details of a cluster node; and many more.

A resource collection is a group of logically related resources of the cluster, such as application services, secrets, data volumes, or containers.

Secrets

In our daily life, we have loads of secrets. Secrets are information that is not meant to be publicly known, such as the username and password combination that you use to access your online bank account or the code to your cell phone or your locker at the school or office.

When writing software, we often need to use secrets, too. For example, we need a certificate to authenticate our application service with the external service that we want to access, or we need a token to authenticate and authorize our service when accessing some other API. In the past, developers, for convenience, have just hardcoded those values or put them in clear text in some external configuration files. There, this very sensitive information has been accessible to a broad audience, which, in reality, should never have had the opportunity to see those secrets.

Luckily, these days, orchestrators offer what is called secrets to deal with such sensitive information in a highly secure way. Secrets can be created by authorized or trusted personnel. The values of those secrets are then encrypted and stored in the highly available cluster state database. The secrets, since they are encrypted, are now secure at rest. Once a secret is requested by an authorized application service, the secret is only forwarded to the cluster nodes that actually run an instance of that particular service, and the secret value is never stored on the node but mounted into the container in a `tmpfs` RAM-based volume. Only inside the respective container is the secret value available in clear text.

We already mentioned that the secrets are secure at rest. Once they are requested by a service, the cluster manager, or master, decrypts the secret and sends it over the wire to the target nodes. So, what about the secrets being secure in transit? Well, we learned earlier that the cluster nodes use MTLS for their communication, and so the secret, although transmitted in clear text, is still secure since data packets will be encrypted by MTLS. Thus, secrets are secure both at rest and in transit. Only services that are authorized to use secrets will ever have access to those secret values.

Content trust

For added security, we want to make sure that only trusted images run in our production cluster. Some orchestrators allow us to configure a cluster so that it can only ever run signed images. Content trust and signing images are all about making sure that the authors of the image are the ones that we expect them to be, namely, our trusted developers or, even better, our trusted CI server. Furthermore, with content trust, we want to guarantee that the image that we get is fresh and is not an old and maybe vulnerable image. And finally, we want to make sure that the image cannot be compromised by malicious hackers in transit. The latter is often called a **man-in-the-middle (MITM)** attack.

By signing images at the source and validating the signature at the target, we can guarantee that the images that we want to run are not compromised.

Reverse uptime

The last point I want to discuss in the context of security is reverse uptime. What do we mean by that? Imagine that you have configured and secured a production

When writing software, we often need to use secrets, too. For example, we need a certificate to authenticate our application service with the external service that we want to access, or we need a token to authenticate and authorize our service when accessing some other API. In the past, developers, for convenience, have just hardcoded those values or put them in clear text in some external configuration files. There, this very sensitive information has been accessible to a broad audience, which, in reality, should never have had the opportunity to see those secrets.

Luckily, these days, orchestrators offer what is called secrets to deal with such sensitive information in a highly secure way. Secrets can be created by authorized or trusted personnel. The values of those secrets are then encrypted and stored in the highly available cluster state database. The secrets, since they are encrypted, are now secure at rest. Once a secret is requested by an authorized application service, the secret is only forwarded to the cluster nodes that actually run an instance of that particular service, and the secret value is never stored on the node but mounted into the container in a `tmpfs` RAM-based volume. Only inside the respective container is the secret value available in clear text.

We already mentioned that the secrets are secure at rest. Once they are requested by a service, the cluster manager, or master, decrypts the secret and sends it over the wire to the target nodes. So, what about the secrets being secure in transit? Well, we learned earlier that the cluster nodes use MTLS for their communication, and so the secret, although transmitted in clear text, is still secure since data packets will be encrypted by MTLS. Thus, secrets are secure both at rest and in transit. Only services that are authorized to use secrets will ever have access to those secret values.

Content trust

For added security, we want to make sure that only trusted images run in our production cluster. Some orchestrators allow us to configure a cluster so that it can only ever run signed images. Content trust and signing images are all about making sure that the authors of the image are the ones that we expect them to be, namely, our trusted developers or, even better, our trusted CI server. Furthermore, with content trust, we want to guarantee that the image that we get is fresh and is not an old and maybe vulnerable image. And finally, we want to make sure that the image cannot be compromised by malicious hackers in transit. The latter is often called a **man-in-the-middle (MITM)** attack.

By signing images at the source and validating the signature at the target, we can guarantee that the images that we want to run are not compromised.

Reverse uptime

The last point I want to discuss in the context of security is reverse uptime. What do we mean by that? Imagine that you have configured and secured a production

cluster. On this cluster, you're running a few mission-critical applications of your company. Now, a hacker has managed to find a security hole in one of your software stacks and has gained root access to one of your cluster nodes. That alone is already bad enough, but even worse, this hacker could now mask their presence on this node they are root on the machine, after all, and then use it as a base to attack other nodes in your cluster.

But what if we leverage the fact that containers are ephemeral and cluster nodes are quickly provisioned, usually in a matter of minutes if fully automated? We just kill each cluster node after a certain uptime of, say, one day. The orchestrator is instructed to drain the node and then exclude it from the cluster. Once the node is out of the cluster, it is torn down and replaced by a freshly provisioned node.

That way, the hacker has lost their base, and the problem has been eliminated. This concept is not yet broadly available, though, but to me, it seems to be a huge step toward increased security, and, as far as I have discussed it with engineers who are working in this area, it is not difficult to implement.

Facilitating introspection

So far, we have discussed a lot of tasks for which the orchestrator is responsible and that it can execute in a completely autonomous way. But there is also the need for human operators to be able to see and analyze what is currently running on the cluster and in what state or health the individual applications are. For all this, we need the possibility of introspection. The orchestrator needs to surface crucial information in a way that is easily consumable and understandable.

The orchestrator should collect system metrics from all the cluster nodes and make them accessible to the operators. Metrics include CPU, memory, disk usage, network bandwidth consumption, and more. The information should be easily available on a node-per-node basis, as well as in an aggregated form.

We also want the orchestrator to give us access to logs that are produced by service instances or containers. Even more, the orchestrator should provide us with exec access to each and every container if we have the correct authorization to do so. With exec access to containers, we can then debug misbehaving containers.

In highly distributed applications, where each request to the application goes through numerous services until it is completely handled, tracing requests is a really important task. Ideally, the orchestrator supports us in implementing a tracing strategy or gives us some good guidelines to follow.

Finally, human operators can best monitor a system when working with a graphical representation of all the collected metrics and logging and tracing information. Here, we are speaking about dashboards. Every decent orchestrator should offer at least

cluster. On this cluster, you're running a few mission-critical applications of your company. Now, a hacker has managed to find a security hole in one of your software stacks and has gained root access to one of your cluster nodes. That alone is already bad enough, but even worse, this hacker could now mask their presence on this node they are root on the machine, after all, and then use it as a base to attack other nodes in your cluster.

But what if we leverage the fact that containers are ephemeral and cluster nodes are quickly provisioned, usually in a matter of minutes if fully automated? We just kill each cluster node after a certain uptime of, say, one day. The orchestrator is instructed to drain the node and then exclude it from the cluster. Once the node is out of the cluster, it is torn down and replaced by a freshly provisioned node.

That way, the hacker has lost their base, and the problem has been eliminated. This concept is not yet broadly available, though, but to me, it seems to be a huge step toward increased security, and, as far as I have discussed it with engineers who are working in this area, it is not difficult to implement.

Facilitating introspection

So far, we have discussed a lot of tasks for which the orchestrator is responsible and that it can execute in a completely autonomous way. But there is also the need for human operators to be able to see and analyze what is currently running on the cluster and in what state or health the individual applications are. For all this, we need the possibility of introspection. The orchestrator needs to surface crucial information in a way that is easily consumable and understandable.

The orchestrator should collect system metrics from all the cluster nodes and make them accessible to the operators. Metrics include CPU, memory, disk usage, network bandwidth consumption, and more. The information should be easily available on a node-per-node basis, as well as in an aggregated form.

We also want the orchestrator to give us access to logs that are produced by service instances or containers. Even more, the orchestrator should provide us with exec access to each and every container if we have the correct authorization to do so. With exec access to containers, we can then debug misbehaving containers.

In highly distributed applications, where each request to the application goes through numerous services until it is completely handled, tracing requests is a really important task. Ideally, the orchestrator supports us in implementing a tracing strategy or gives us some good guidelines to follow.

Finally, human operators can best monitor a system when working with a graphical representation of all the collected metrics and logging and tracing information. Here, we are speaking about dashboards. Every decent orchestrator should offer at least

some basic dashboard with a graphical representation of the most critical system parameters.

However, human operators are not the only ones concerned about introspection. We also need to be able to connect external systems with the orchestrator in order to consume this information. There needs to be an API available over which external systems can access data such as cluster state, metrics, and logs and use this information to make automated decisions, such as creating pager or phone alerts, sending out e-mails, or triggering an alarm siren if some thresholds are exceeded by the system.

Overview of popular orchestrators

Now that you have a good understanding of what an orchestrator is and what type of tasks it is commonly assigned let us have a quick look at what the most popular orchestrators are. Let us start with the elephant in the room, Kubernetes.

Kubernetes

Kubernetes was originally designed by Google and later donated to the **Cloud Native Computing Foundation (CNCF)**. Kubernetes was modeled after Google's proprietary Borg system, which has been running containers on a supermassive scale for years. Kubernetes was Google's attempt to go back to the drawing board and completely start over and design a system that incorporates all the lessons that were learned with Borg.

Contrary to Borg, which is proprietary technology, Kubernetes was open-sourced early on. This was a very wise choice by Google since it attracted a huge number of contributors from outside of the company and, over only a couple of years, an even more massive ecosystem evolved around Kubernetes. You can rightfully say that Kubernetes is the darling of the community in the container orchestration space. No other orchestrator has been able to produce so much hype and attract so many talented people who are willing to contribute in a meaningful way to the success of the project as a contributor or an early adopters.

In that regard, Kubernetes in the container orchestration space looks to me very much like what Linux has become in the server operating system space. Linux has become the de facto standard of server operating systems. All relevant companies, such as Microsoft, IBM, Amazon, Red Hat, and even Docker, have embraced Kubernetes.

And there is one thing that cannot be denied: Kubernetes was designed from the very beginning for massive scalability. After all, it was designed with Google Borg in mind.

some basic dashboard with a graphical representation of the most critical system parameters.

However, human operators are not the only ones concerned about introspection. We also need to be able to connect external systems with the orchestrator in order to consume this information. There needs to be an API available over which external systems can access data such as cluster state, metrics, and logs and use this information to make automated decisions, such as creating pager or phone alerts, sending out e-mails, or triggering an alarm siren if some thresholds are exceeded by the system.

Overview of popular orchestrators

Now that you have a good understanding of what an orchestrator is and what type of tasks it is commonly assigned let us have a quick look at what the most popular orchestrators are. Let us start with the elephant in the room, Kubernetes.

Kubernetes

Kubernetes was originally designed by Google and later donated to the **Cloud Native Computing Foundation (CNCF)**. Kubernetes was modeled after Google's proprietary Borg system, which has been running containers on a supermassive scale for years. Kubernetes was Google's attempt to go back to the drawing board and completely start over and design a system that incorporates all the lessons that were learned with Borg.

Contrary to Borg, which is proprietary technology, Kubernetes was open-sourced early on. This was a very wise choice by Google since it attracted a huge number of contributors from outside of the company and, over only a couple of years, an even more massive ecosystem evolved around Kubernetes. You can rightfully say that Kubernetes is the darling of the community in the container orchestration space. No other orchestrator has been able to produce so much hype and attract so many talented people who are willing to contribute in a meaningful way to the success of the project as a contributor or an early adopters.

In that regard, Kubernetes in the container orchestration space looks to me very much like what Linux has become in the server operating system space. Linux has become the de facto standard of server operating systems. All relevant companies, such as Microsoft, IBM, Amazon, Red Hat, and even Docker, have embraced Kubernetes.

And there is one thing that cannot be denied: Kubernetes was designed from the very beginning for massive scalability. After all, it was designed with Google Borg in mind.

One negative aspect that you could be voiced against Kubernetes is that it is still complex to set up and manage, at least at the time of writing. There is a significant hurdle to overcome for newcomers. The first step is steep, but once you have worked with this orchestrator for a while, it all makes sense. The overall design is carefully thought through and executed very well.

In release 1.10 of Kubernetes, whose **general availability (GA)** was in March 2018, most of the initial shortcomings compared to other orchestrators, such as Docker Swarm, has been eliminated. For example, security and confidentiality are now not only an afterthought but an integral part of the system.

New features are implemented at a tremendous speed. New releases are happening every three months or so, more precisely, about every 100 days. Most of the new features are demand-driven; that is, companies using Kubernetes to orchestrate their mission-critical applications can voice their needs. This makes Kubernetes enterprise-ready. It would be wrong to assume that this orchestrator is only for start-ups and not for risk-averse enterprises. The contrary is the case. On what do I base this claim? Well, my claim is justified by the fact that companies such as Microsoft, Docker, and Red Hat, whose clients are mostly big enterprises, have fully embraced Kubernetes and provide enterprise-grade support for it if it is used and integrated into their enterprise offerings.

Note that Kubernetes supports both Linux and Windows containers.

Docker Swarm

It is well known that Docker popularized and commoditized software containers. Docker did not invent containers but standardized them and made them broadly available, not least by offering the free image registry—Docker Hub. Initially, Docker focused mainly on the developer and the development life cycle. However, companies that started to use and love containers soon also wanted to use them not just during the development or testing of new applications but also to run those applications in production.

Initially, Docker had nothing to offer in that space, so other companies jumped into that vacuum and offered help to the users. But it did not take long, and Docker recognized that there was a huge demand for a simple yet powerful orchestrator. Docker's first attempt was a product called **Classic Swarm.** It was a standalone product that enabled users to create a cluster of Docker host machines that could be used to run and scale their containerized applications in a highly available and self-healing way.

The setup of a classic Docker swarm, though, was hard. A lot of complicated manual steps were involved. Customers loved the product but struggled with its complexity. So, Docker decided it could do better. It went back to the drawing board and came

One negative aspect that you could be voiced against Kubernetes is that it is still complex to set up and manage, at least at the time of writing. There is a significant hurdle to overcome for newcomers. The first step is steep, but once you have worked with this orchestrator for a while, it all makes sense. The overall design is carefully thought through and executed very well.

In release 1.10 of Kubernetes, whose **general availability (GA)** was in March 2018, most of the initial shortcomings compared to other orchestrators, such as Docker Swarm, has been eliminated. For example, security and confidentiality are now not only an afterthought but an integral part of the system.

New features are implemented at a tremendous speed. New releases are happening every three months or so, more precisely, about every 100 days. Most of the new features are demand-driven; that is, companies using Kubernetes to orchestrate their mission-critical applications can voice their needs. This makes Kubernetes enterprise-ready. It would be wrong to assume that this orchestrator is only for start-ups and not for risk-averse enterprises. The contrary is the case. On what do I base this claim? Well, my claim is justified by the fact that companies such as Microsoft, Docker, and Red Hat, whose clients are mostly big enterprises, have fully embraced Kubernetes and provide enterprise-grade support for it if it is used and integrated into their enterprise offerings.

Note that Kubernetes supports both Linux and Windows containers.

Docker Swarm

It is well known that Docker popularized and commoditized software containers. Docker did not invent containers but standardized them and made them broadly available, not least by offering the free image registry—Docker Hub. Initially, Docker focused mainly on the developer and the development life cycle. However, companies that started to use and love containers soon also wanted to use them not just during the development or testing of new applications but also to run those applications in production.

Initially, Docker had nothing to offer in that space, so other companies jumped into that vacuum and offered help to the users. But it did not take long, and Docker recognized that there was a huge demand for a simple yet powerful orchestrator. Docker's first attempt was a product called **Classic Swarm.** It was a standalone product that enabled users to create a cluster of Docker host machines that could be used to run and scale their containerized applications in a highly available and self-healing way.

The setup of a classic Docker swarm, though, was hard. A lot of complicated manual steps were involved. Customers loved the product but struggled with its complexity. So, Docker decided it could do better. It went back to the drawing board and came

up with SwarmKit. SwarmKit was introduced at DockerCon 2016 in Seattle and was an integral part of the newest version of the Docker engine. Yes, you got that right; SwarmKit was, and still is to this day, an integral part of the Docker engine. Thus, if you install a Docker host, you automatically have SwarmKit available with it.

SwarmKit was designed with simplicity and security in mind. The mantra was and still is that it has to be almost trivial to set up a swarm and that the swarm has to be highly secure out of the box. Docker Swarm operates on the assumption of least privilege.

Installing a complete, highly available Docker swarm is literally as simple as starting with `docker swarm init` on the first node in the cluster, which becomes the so-called leader, and then `docker swarm join <join-token>` on all other nodes. `<join-token>` is generated by the leader during initialization. The whole process takes fewer than 5 minutes on a swarm with up to 10 nodes. If it is automated, it takes even less time.

As I already mentioned, security was top on the list of must-haves when Docker designed and developed SwarmKit. Containers provide security by relying on

Linux kernel namespaces and cgroups, as well as Linux syscall whitelisting (seccomp), and the support of Linux capabilities and the **Linux security module (LSM)**. Now, on top of that, SwarmKit adds MTLS and secrets that are encrypted at rest and in transit. Furthermore, Swarm defines the so-called **container network model (CNM)**, which allows for SDNs that provide sandboxing for application services that are running on the swarm.

Similar to Kubernetes, Docker SwarmKit supports both Linux and Windows containers.

Apache Mesos and Marathon

Apache Mesos is an open-source project and was originally designed to make a cluster of servers or nodes look like one single big server from the outside. Mesos is software that makes the management of computer clusters simple. Users of Mesos should not have to care about individual servers but just assume they have a gigantic pool of resources at their disposal, which corresponds to the aggregate of all the resources of all the nodes in the cluster.

Mesos, in IT terms, is already pretty old, at least compared to the other orchestrators. It was first publicly presented in 2009, but at that time, of course, it was not designed to run containers since Docker did not even exist yet. Similar to what Docker does with containers, Mesos uses Linux cgroups to isolate resources such as CPU, memory, or disk I/O for individual applications or services.

up with SwarmKit. SwarmKit was introduced at DockerCon 2016 in Seattle and was an integral part of the newest version of the Docker engine. Yes, you got that right; SwarmKit was, and still is to this day, an integral part of the Docker engine. Thus, if you install a Docker host, you automatically have SwarmKit available with it.

SwarmKit was designed with simplicity and security in mind. The mantra was and still is that it has to be almost trivial to set up a swarm and that the swarm has to be highly secure out of the box. Docker Swarm operates on the assumption of least privilege.

Installing a complete, highly available Docker swarm is literally as simple as starting with `docker swarm init` on the first node in the cluster, which becomes the so-called leader, and then `docker swarm join <join-token>` on all other nodes. `<join-token>` is generated by the leader during initialization. The whole process takes fewer than 5 minutes on a swarm with up to 10 nodes. If it is automated, it takes even less time.

As I already mentioned, security was top on the list of must-haves when Docker designed and developed SwarmKit. Containers provide security by relying on

Linux kernel namespaces and cgroups, as well as Linux syscall whitelisting (seccomp), and the support of Linux capabilities and the **Linux security module (LSM)**. Now, on top of that, SwarmKit adds MTLS and secrets that are encrypted at rest and in transit. Furthermore, Swarm defines the so-called **container network model (CNM)**, which allows for SDNs that provide sandboxing for application services that are running on the swarm.

Similar to Kubernetes, Docker SwarmKit supports both Linux and Windows containers.

Apache Mesos and Marathon

Apache Mesos is an open-source project and was originally designed to make a cluster of servers or nodes look like one single big server from the outside. Mesos is software that makes the management of computer clusters simple. Users of Mesos should not have to care about individual servers but just assume they have a gigantic pool of resources at their disposal, which corresponds to the aggregate of all the resources of all the nodes in the cluster.

Mesos, in IT terms, is already pretty old, at least compared to the other orchestrators. It was first publicly presented in 2009, but at that time, of course, it was not designed to run containers since Docker did not even exist yet. Similar to what Docker does with containers, Mesos uses Linux cgroups to isolate resources such as CPU, memory, or disk I/O for individual applications or services.

Mesos is really the underlying infrastructure for other interesting services built on top of it. From the perspective of containers specifically, Marathon is important. Marathon is a container orchestrator that runs on top of Mesos, which is able to scale to thousands of nodes.

Marathon supports multiple container runtimes, such as Docker or its own Mesos containers. It supports not only stateless but also stateful application services, for example, databases such as PostgreSQL or MongoDB. Similar to Kubernetes and Docker SwarmKit, it supports many of the features that were described earlier in this chapter, such as high availability, health checks, service discovery, load balancing, and location awareness, to name but a few of the most important ones.

Although Mesos and, to a certain extent, Marathon are rather mature projects, their reach is relatively limited. It seems to be most popular in the area of big data, that is, to run data-crunching services such as Spark or Hadoop.

Amazon AWS ECS

If you are looking for a simple orchestrator and have already heavily bought into the AWS ecosystem, then Amazon's **elastic container service** (**ECS**) might be the right choice for you. It is important to point out one very important limitation of ECS: if you buy into this container orchestrator, then you lock yourself into AWS. You will not be able to easily port an application that is running on ECS to another platform or cloud.

Amazon promotes its ECS service as a highly scalable, fast container management service that makes it easy to run, stop, and manage Docker containers on a cluster. Next to running containers, ECS gives direct access to many other AWS services from the application services that run inside the containers. This tight and seamless integration with many popular AWS services is what makes ECS compelling for users who are looking for an easy way to get their containerized applications up and running in a robust and highly scalable environment. Amazon also provides its own private image registry.

With AWS ECS, you can use Fargate to have it fully manage the underlying infrastructure, allowing you to concentrate exclusively on deploying containerized applications, and you do not have to care about how to create and manage a cluster of nodes. ECS supports both Linux and Windows containers.

In summary, ECS is simple to use, highly scalable, and well-integrated with other popular AWS services; but it is not as powerful as, say, Kubernetes or Docker SwarmKit, and it is only available on Amazon AWS.

Mesos is really the underlying infrastructure for other interesting services built on top of it. From the perspective of containers specifically, Marathon is important. Marathon is a container orchestrator that runs on top of Mesos, which is able to scale to thousands of nodes.

Marathon supports multiple container runtimes, such as Docker or its own Mesos containers. It supports not only stateless but also stateful application services, for example, databases such as PostgreSQL or MongoDB. Similar to Kubernetes and Docker SwarmKit, it supports many of the features that were described earlier in this chapter, such as high availability, health checks, service discovery, load balancing, and location awareness, to name but a few of the most important ones.

Although Mesos and, to a certain extent, Marathon are rather mature projects, their reach is relatively limited. It seems to be most popular in the area of big data, that is, to run data-crunching services such as Spark or Hadoop.

Amazon AWS ECS

If you are looking for a simple orchestrator and have already heavily bought into the AWS ecosystem, then Amazon's **elastic container service** (**ECS**) might be the right choice for you. It is important to point out one very important limitation of ECS: if you buy into this container orchestrator, then you lock yourself into AWS. You will not be able to easily port an application that is running on ECS to another platform or cloud.

Amazon promotes its ECS service as a highly scalable, fast container management service that makes it easy to run, stop, and manage Docker containers on a cluster. Next to running containers, ECS gives direct access to many other AWS services from the application services that run inside the containers. This tight and seamless integration with many popular AWS services is what makes ECS compelling for users who are looking for an easy way to get their containerized applications up and running in a robust and highly scalable environment. Amazon also provides its own private image registry.

With AWS ECS, you can use Fargate to have it fully manage the underlying infrastructure, allowing you to concentrate exclusively on deploying containerized applications, and you do not have to care about how to create and manage a cluster of nodes. ECS supports both Linux and Windows containers.

In summary, ECS is simple to use, highly scalable, and well-integrated with other popular AWS services; but it is not as powerful as, say, Kubernetes or Docker SwarmKit, and it is only available on Amazon AWS.

Microsoft Azure ACS

Similar to what we said about ECS, we can claim the same for Microsoft's **Azure container service (ACS)**. It is a simple container orchestration service that makes sense if you are already heavily invested in the Azure ecosystem. I should say the same as I have pointed out for Amazon ECS: if you buy into ACS, then you lock yourself into the offerings of Microsoft. It will not be easy to move your containerized applications from ACS to any other platform or cloud.

ACS is Microsoft's container service, which supports multiple orchestrators such as Kubernetes, Docker Swarm, and Mesos DC/OS. With Kubernetes becoming more and more popular, the focus of Microsoft has clearly shifted to that orchestrator. Microsoft has even rebranded its service and called it **Azure Kubernetes Service (AKS)** in order to put the focus on Kubernetes.

Understanding the architecture of Kubernetes

A Kubernetes cluster consists of a set of servers. These servers can be VMs or physical servers. The latter is also called bare metal. Each member of the cluster can have one of two roles. It is either a Kubernetes master or a (worker) node. The former is used to manage the cluster, whereas the latter will run an application workload. I have put the worker in parentheses since, in Kubernetes parlance, you only talk about a node when you are talking about a server that runs application workloads.

In a cluster, you have a small and odd number of masters and as many worker nodes as needed. Small clusters might only have a few worker nodes, while more realistic clusters might have dozens or even hundreds of worker nodes. Technically, there is no limit to how many worker nodes a cluster can have; in reality, though, you might experience a significant slowdown in some management operations when dealing with thousands of nodes. All members of the cluster need to be connected by a physical network, the so-called underlay network.

Kubernetes defines one flat network for the whole cluster. Kubernetes does not provide any networking implementation out of the box; instead, it relies on plugins from third parties. Kubernetes just defines the **Container Network Interface (CNI)** and leaves the implementation to others. The CNI is pretty simple. It basically states that each pod running in the cluster must be able to reach any other pod also running in the cluster without any **Network Address Translation (NAT)** happening in-between. The same must be true between cluster nodes and pods; that is, applications or daemons running directly on a cluster node must be able to reach each pod in the cluster and vice versa.

Microsoft Azure ACS

Similar to what we said about ECS, we can claim the same for Microsoft's **Azure container service (ACS)**. It is a simple container orchestration service that makes sense if you are already heavily invested in the Azure ecosystem. I should say the same as I have pointed out for Amazon ECS: if you buy into ACS, then you lock yourself into the offerings of Microsoft. It will not be easy to move your containerized applications from ACS to any other platform or cloud.

ACS is Microsoft's container service, which supports multiple orchestrators such as Kubernetes, Docker Swarm, and Mesos DC/OS. With Kubernetes becoming more and more popular, the focus of Microsoft has clearly shifted to that orchestrator. Microsoft has even rebranded its service and called it **Azure Kubernetes Service (AKS)** in order to put the focus on Kubernetes.

Understanding the architecture of Kubernetes

A Kubernetes cluster consists of a set of servers. These servers can be VMs or physical servers. The latter is also called bare metal. Each member of the cluster can have one of two roles. It is either a Kubernetes master or a (worker) node. The former is used to manage the cluster, whereas the latter will run an application workload. I have put the worker in parentheses since, in Kubernetes parlance, you only talk about a node when you are talking about a server that runs application workloads.

In a cluster, you have a small and odd number of masters and as many worker nodes as needed. Small clusters might only have a few worker nodes, while more realistic clusters might have dozens or even hundreds of worker nodes. Technically, there is no limit to how many worker nodes a cluster can have; in reality, though, you might experience a significant slowdown in some management operations when dealing with thousands of nodes. All members of the cluster need to be connected by a physical network, the so-called underlay network.

Kubernetes defines one flat network for the whole cluster. Kubernetes does not provide any networking implementation out of the box; instead, it relies on plugins from third parties. Kubernetes just defines the **Container Network Interface (CNI)** and leaves the implementation to others. The CNI is pretty simple. It basically states that each pod running in the cluster must be able to reach any other pod also running in the cluster without any **Network Address Translation (NAT)** happening in-between. The same must be true between cluster nodes and pods; that is, applications or daemons running directly on a cluster node must be able to reach each pod in the cluster and vice versa.

The following diagram illustrates the high-level architecture of a Kubernetes cluster:

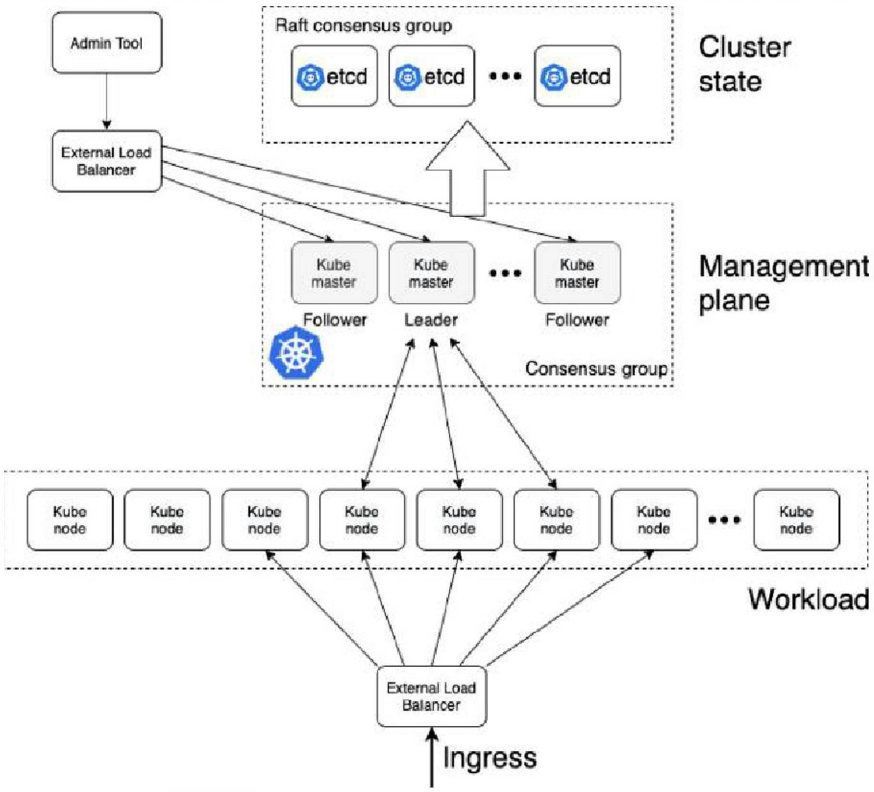


Figure 10.5: High-level architecture diagram of Kubernetes

In the previous diagram, we can see the following:

1. On the top, in the middle, we have a cluster of **etcd** nodes. etcd is a distributed key-value store that, in a Kubernetes cluster, is used to store all the state of the cluster. The number of etcd nodes has to be odd, as mandated by the Raft consensus protocol, which states which nodes are used to coordinate among themselves. When we talk about the Cluster State, we do not include data that is produced or consumed by applications running in the cluster; instead, we are talking about all the information on the topology of the cluster, what services are running, network settings, secrets used, and more. That said, this etcd cluster is really mission-critical to the overall cluster, and thus, we should never run only a single etcd server in a production environment or any environment that needs to be highly available.

The following diagram illustrates the high-level architecture of a Kubernetes cluster:

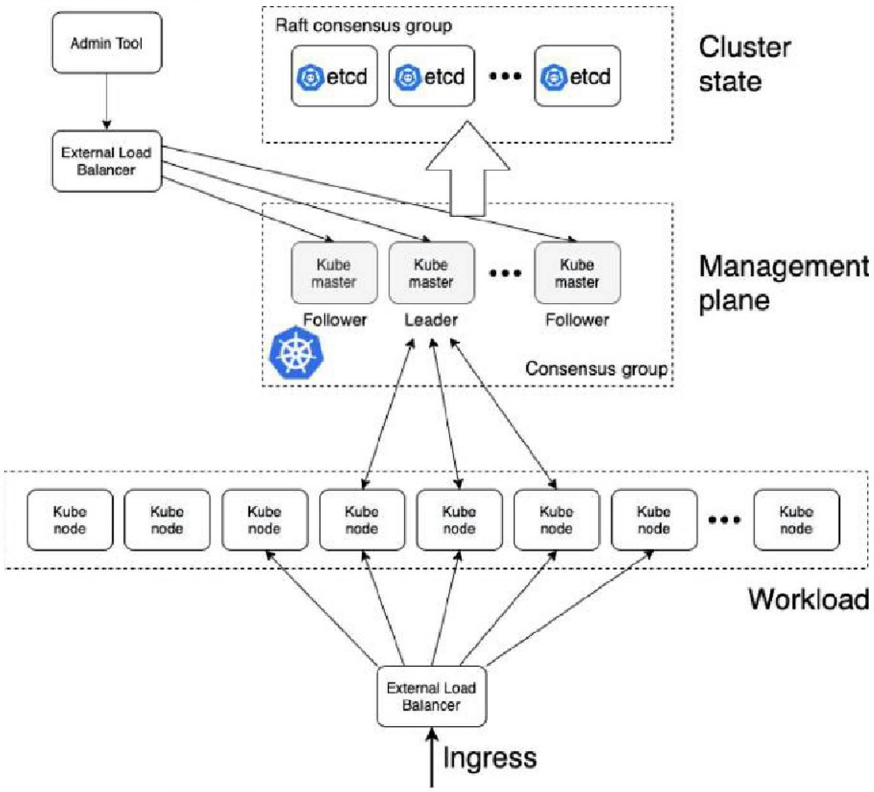


Figure 10.5: High-level architecture diagram of Kubernetes

In the previous diagram, we can see the following:

1. On the top, in the middle, we have a cluster of **etcd** nodes. etcd is a distributed key-value store that, in a Kubernetes cluster, is used to store all the state of the cluster. The number of etcd nodes has to be odd, as mandated by the Raft consensus protocol, which states which nodes are used to coordinate among themselves. When we talk about the Cluster State, we do not include data that is produced or consumed by applications running in the cluster; instead, we are talking about all the information on the topology of the cluster, what services are running, network settings, secrets used, and more. That said, this etcd cluster is really mission-critical to the overall cluster, and thus, we should never run only a single etcd server in a production environment or any environment that needs to be highly available.

The Raft consensus protocol is a distributed algorithm that ensures the agreement of a set of state machines, replicating data across them. It is designed to be a highly available, fault-tolerant, and easy-to-understand alternative to Paxos, another popular consensus algorithm.

2. Then, we have a cluster of Kubernetes master nodes, which also form a Consensus Group among themselves, similar to the etcd nodes. The number of master nodes also has to be odd. We can run a cluster with a single master, but we should never do that in a production or mission-critical system. There, we should always have at least three master nodes. Since the master nodes are used to manage the whole cluster, we are also talking about the management plane. Master nodes use the etcd cluster as their backing store. It is good practice to put a **load balancer (LB)** in front of master nodes with a well-known, **Fully Qualified Domain Name (FQDN)**, such as **https://admin.example.com**. All tools that are used to manage the Kubernetes cluster should access it through this LB rather than using the public IP address of one of the master nodes. This is shown on the left upper side of the preceding diagram.

3. Toward the bottom of the diagram, we have a cluster of worker nodes. The number of nodes can be as low as one and does not have an upper limit. Kubernetes master and worker nodes communicate with each other. It is a bidirectional form of communication. All ingress traffic accessing applications running in the cluster should go through another load balancer. This is the application load balancer or reverse proxy. We never want external traffic to directly access any of the worker nodes.

Now that we have an idea about the high-level architecture of a Kubernetes cluster let us dive a bit more deeply and look at the Kubernetes master and worker nodes.

The Raft consensus protocol is a distributed algorithm that ensures the agreement of a set of state machines, replicating data across them. It is designed to be a highly available, fault-tolerant, and easy-to-understand alternative to Paxos, another popular consensus algorithm.

2. Then, we have a cluster of Kubernetes master nodes, which also form a Consensus Group among themselves, similar to the etcd nodes. The number of master nodes also has to be odd. We can run a cluster with a single master, but we should never do that in a production or mission-critical system. There, we should always have at least three master nodes. Since the master nodes are used to manage the whole cluster, we are also talking about the management plane. Master nodes use the etcd cluster as their backing store. It is good practice to put a **load balancer (LB)** in front of master nodes with a well-known, **Fully Qualified Domain Name (FQDN)**, such as **https://admin.example.com**. All tools that are used to manage the Kubernetes cluster should access it through this LB rather than using the public IP address of one of the master nodes. This is shown on the left upper side of the preceding diagram.

3. Toward the bottom of the diagram, we have a cluster of worker nodes. The number of nodes can be as low as one and does not have an upper limit. Kubernetes master and worker nodes communicate with each other. It is a bidirectional form of communication. All ingress traffic accessing applications running in the cluster should go through another load balancer. This is the application load balancer or reverse proxy. We never want external traffic to directly access any of the worker nodes.

Now that we have an idea about the high-level architecture of a Kubernetes cluster let us dive a bit more deeply and look at the Kubernetes master and worker nodes.

Master and worker nodes

Kubernetes master nodes are used to manage a Kubernetes cluster. The following is a high-level diagram of such a master:

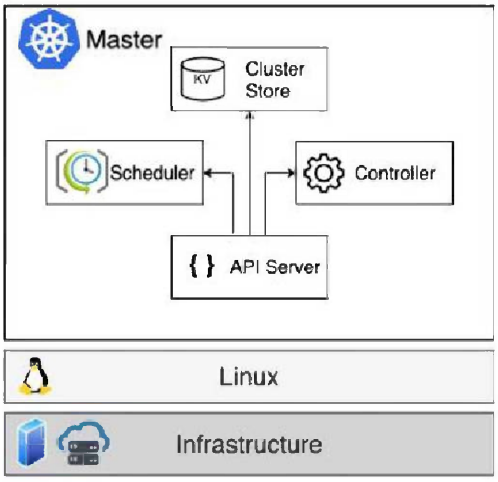


Figure 10.6: Kubernetes master

At the bottom of the preceding diagram, we have the infrastructure, which can be a VM on-premise or in the cloud or a server (often called bare metal) on-premise or in the cloud. Currently, Kubernetes masters only run on Linux. The most popular Linux distributions, such as RHEL, CentOS, and Ubuntu, are supported. On this Linux machine, we have at least the following four Kubernetes services running:

- **API server**: This is the gateway to Kubernetes. All requests to list, create, modify, or delete any resources in the cluster must go through this service. It exposes a REST interface that tools such as `kubectl` use to manage the cluster and applications in the cluster.
- **Controller**: The controller, or more precisely the controller manager, is a control loop that observes the state of the cluster through the API server and makes changes, attempting to move the current or effective state toward the desired state if they differ.
- **Scheduler**: The scheduler is a service that tries its best to schedule pods on worker nodes while considering various boundary conditions, such as resource requirements, policies, quality of service requirements, and more.
- **Cluster store**: This is an instance of etcd that is used to store all information about the state of the cluster.

Master and worker nodes

Kubernetes master nodes are used to manage a Kubernetes cluster. The following is a high-level diagram of such a master:

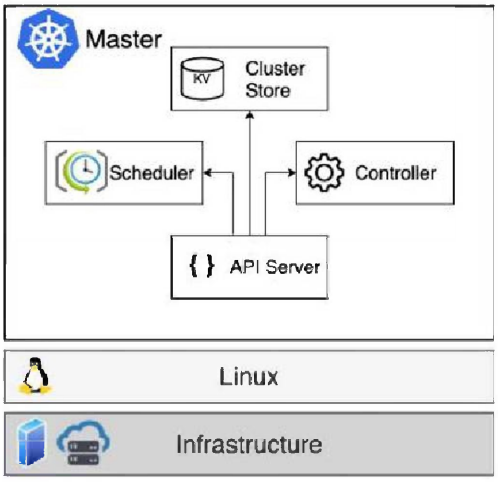


Figure 10.6: Kubernetes master

At the bottom of the preceding diagram, we have the infrastructure, which can be a VM on-premise or in the cloud or a server (often called bare metal) on-premise or in the cloud. Currently, Kubernetes masters only run on Linux. The most popular Linux distributions, such as RHEL, CentOS, and Ubuntu, are supported. On this Linux machine, we have at least the following four Kubernetes services running:

- **API server**: This is the gateway to Kubernetes. All requests to list, create, modify, or delete any resources in the cluster must go through this service. It exposes a REST interface that tools such as `kubectl` use to manage the cluster and applications in the cluster.
- **Controller**: The controller, or more precisely the controller manager, is a control loop that observes the state of the cluster through the API server and makes changes, attempting to move the current or effective state toward the desired state if they differ.
- **Scheduler**: The scheduler is a service that tries its best to schedule pods on worker nodes while considering various boundary conditions, such as resource requirements, policies, quality of service requirements, and more.
- **Cluster store**: This is an instance of etcd that is used to store all information about the state of the cluster.

We need at least one master, but to achieve high availability, we need three or more master nodes. This is very similar to what we find as manager nodes of a Docker Swarm. In this regard, a Kubernetes master is equivalent to a Swarm manager node.

Kubernetes masters never run application workloads. Their sole purpose is to manage the cluster. Kubernetes masters build a Raft consensus group.

The state of the Kubernetes cluster is stored in an etcd. If the Kubernetes cluster is supposed to be highly available, then the etcd must also be configured in HA mode, which normally means that we have at least three etcd instances running on different nodes.

Now, let us look at the nodes that will be running the actual workload of the cluster.

Cluster nodes are the nodes with which Kubernetes schedules application workloads. They are the workhorses of the cluster. A Kubernetes cluster can have a few, dozens, hundreds, or even thousands of cluster nodes. Kubernetes has been built from the ground up for high scalability. Do not forget that Kubernetes was modeled with Google Borg in mind, which has been running tens of thousands of containers for years.

A worker node can run on a VM, bare metal, on-premise, or in the cloud. Originally, worker nodes could only be configured on Linux. But since version 1.10 of Kubernetes, worker nodes can also run on Windows Server 2010 or newer. It is perfectly fine to have a mixed cluster with Linux and Windows worker nodes.

Now that we have a good understanding of the high-level architecture of Kubernetes, it is time to learn about all the important Kubernetes objects that are used to define and manage an application that will run on Kubernetes. We will start with pods.

Pods

Contrary to what you are used to when working with Docker Desktop, you cannot run containers directly in a Kubernetes cluster. In a Kubernetes cluster, you can only run pods. Pods in so far are the atomic units of deployment in Kubernetes. A pod is an abstraction of one or many co-located containers that share the same kernel namespaces, such as the network or user namespace.

We need at least one master, but to achieve high availability, we need three or more master nodes. This is very similar to what we find as manager nodes of a Docker Swarm. In this regard, a Kubernetes master is equivalent to a Swarm manager node.

Kubernetes masters never run application workloads. Their sole purpose is to manage the cluster. Kubernetes masters build a Raft consensus group.

The state of the Kubernetes cluster is stored in an etcd. If the Kubernetes cluster is supposed to be highly available, then the etcd must also be configured in HA mode, which normally means that we have at least three etcd instances running on different nodes.

Now, let us look at the nodes that will be running the actual workload of the cluster.

Cluster nodes are the nodes with which Kubernetes schedules application workloads. They are the workhorses of the cluster. A Kubernetes cluster can have a few, dozens, hundreds, or even thousands of cluster nodes. Kubernetes has been built from the ground up for high scalability. Do not forget that Kubernetes was modeled with Google Borg in mind, which has been running tens of thousands of containers for years.

A worker node can run on a VM, bare metal, on-premise, or in the cloud. Originally, worker nodes could only be configured on Linux. But since version 1.10 of Kubernetes, worker nodes can also run on Windows Server 2010 or newer. It is perfectly fine to have a mixed cluster with Linux and Windows worker nodes.

Now that we have a good understanding of the high-level architecture of Kubernetes, it is time to learn about all the important Kubernetes objects that are used to define and manage an application that will run on Kubernetes. We will start with pods.

Pods

Contrary to what you are used to when working with Docker Desktop, you cannot run containers directly in a Kubernetes cluster. In a Kubernetes cluster, you can only run pods. Pods in so far are the atomic units of deployment in Kubernetes. A pod is an abstraction of one or many co-located containers that share the same kernel namespaces, such as the network or user namespace.

The fact that several containers can be co-located and thus share the same network namespace is a very powerful concept. In the following picture, you see two examples of a pod:

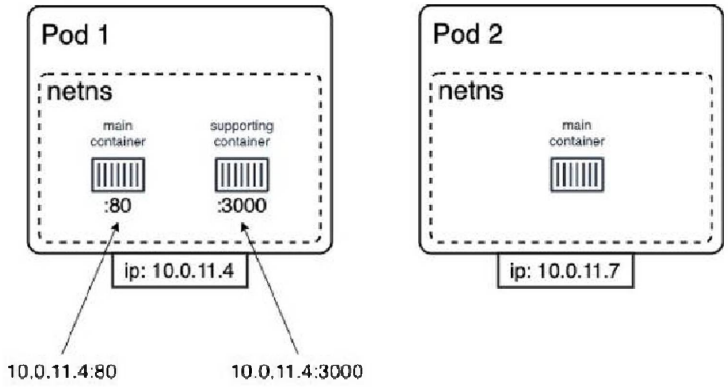


Figure 10.7: Kubernetes pods

In the preceding diagram, you can see two pods called Pod 1 and Pod 2. The first pod contains two containers, whilst the second pod only contains a single container. Kubernetes assigns a unique IP address to each pod. In this case, Pod 1 is assigned IP address 10.0.11.4, and Pod 2 is assigned IP address 10.0.11.7. Both IP addresses are part of a private subnet managed by the Kubernetes network driver.

As stated previously, a pod can host one to many containers. All the containers in a single pod share the same Linux kernel namespaces, and specifically, they share the same network namespace. In the preceding diagram, we have indicated this by the dashed box surrounding the containers. Next to the many advantages that this offers, we have to consider one point though. Each container needs to make sure it uses its own set of ports since duplicate ports are not allowed in a given network namespace. Thus, in Pod 1, the main container uses port 80, whereas the supporting container uses port 3000. If both containers would want to use the same port, say port 80, then an exception would happen, and the pod would fail to start.

Requests from other pods or cluster nodes can use the pods' IP address combined with the corresponding port number to access the individual containers. To access the main container in Pod 1 as an example, you would have to use `10.0.11.4:80`.

Another important consequence of multiple containers sharing the same network namespace is the way that they communicate with each other. In the following diagram, we illustrate once again a pod that contains two containers called **main container** and **supporting container**, respectively, the former listens on port 80 while the latter listens on port 3000:

The fact that several containers can be co-located and thus share the same network namespace is a very powerful concept. In the following picture, you see two examples of a pod:

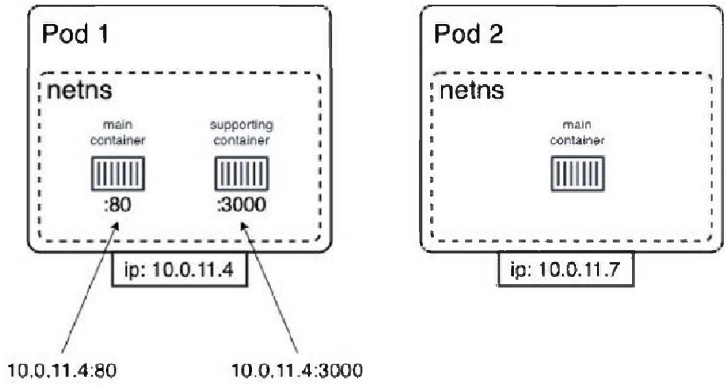


Figure 10.7: Kubernetes pods

In the preceding diagram, you can see two pods called Pod 1 and Pod 2. The first pod contains two containers, whilst the second pod only contains a single container. Kubernetes assigns a unique IP address to each pod. In this case, Pod 1 is assigned IP address 10.0.11.4, and Pod 2 is assigned IP address 10.0.11.7. Both IP addresses are part of a private subnet managed by the Kubernetes network driver.

As stated previously, a pod can host one to many containers. All the containers in a single pod share the same Linux kernel namespaces, and specifically, they share the same network namespace. In the preceding diagram, we have indicated this by the dashed box surrounding the containers. Next to the many advantages that this offers, we have to consider one point though. Each container needs to make sure it uses its own set of ports since duplicate ports are not allowed in a given network namespace. Thus, in Pod 1, the main container uses port 80, whereas the supporting container uses port 3000. If both containers would want to use the same port, say port 80, then an exception would happen, and the pod would fail to start.

Requests from other pods or cluster nodes can use the pods' IP address combined with the corresponding port number to access the individual containers. To access the main container in Pod 1 as an example, you would have to use `10.0.11.4:80`.

Another important consequence of multiple containers sharing the same network namespace is the way that they communicate with each other. In the following diagram, we illustrate once again a pod that contains two containers called **main container** and **supporting container**, respectively, the former listens on port 80 while the latter listens on port 3000:

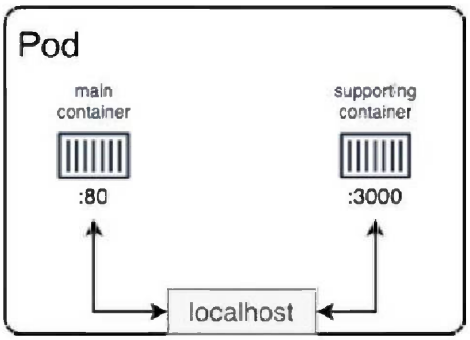


Figure 10.8: Containers in a pod communicating via localhost

When the two containers want to communicate with each other, they can do so through `localhost`, similar to when two processes are running on the same host (VM or bare metal server); they can communicate with each other via `localhost`. As an example, the code running inside the **main container** can reach the process running inside the **supporting container** through `http://localhost:3000`. This has been depicted in the preceding graphic.

The pod life cycle

You have learned that a container has a life cycle. It can go through the phases of being initialized, run and finally exit. When a container exits, it can do this gracefully, with an exit code of zero, or it can terminate with an error, which is equivalent to a non-zero exit code.

Equivalently, a Kubernetes pod also has a life cycle. But since a pod can contain more than one container, its life cycle is a bit more complex than the one of a single container. We have tried to illustrate this in the following graphic:

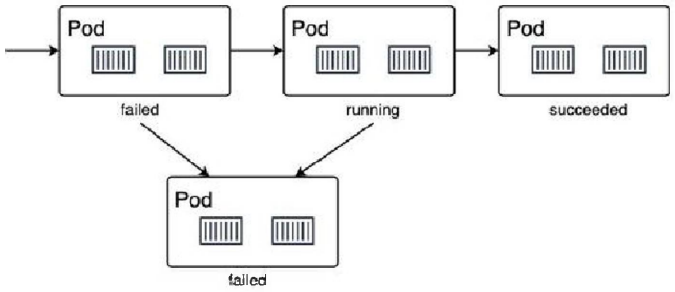


Figure 10.9: Life cycle of a Kubernetes pod

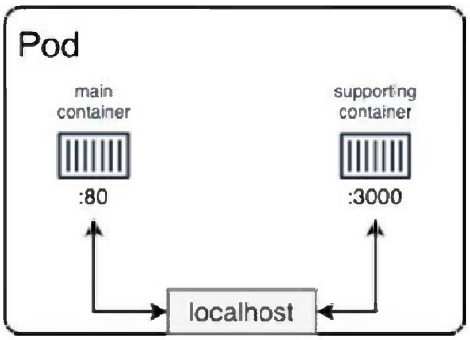


Figure 10.8: Containers in a pod communicating via localhost

When the two containers want to communicate with each other, they can do so through `localhost`, similar to when two processes are running on the same host (VM or bare metal server); they can communicate with each other via `localhost`. As an example, the code running inside the **main container** can reach the process running inside the **supporting container** through `http://localhost:3000`. This has been depicted in the preceding graphic.

The pod life cycle

You have learned that a container has a life cycle. It can go through the phases of being initialized, run and finally exit. When a container exits, it can do this gracefully, with an exit code of zero, or it can terminate with an error, which is equivalent to a non-zero exit code.

Equivalently, a Kubernetes pod also has a life cycle. But since a pod can contain more than one container, its life cycle is a bit more complex than the one of a single container. We have tried to illustrate this in the following graphic:

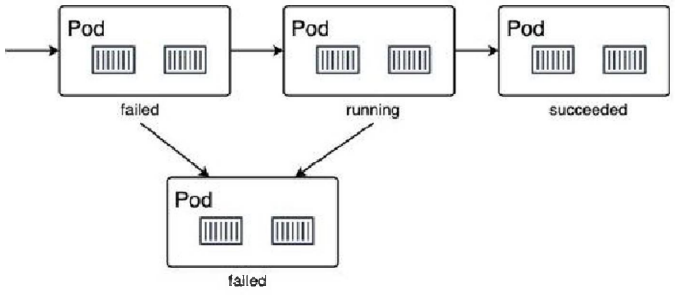


Figure 10.9: Life cycle of a Kubernetes pod

When a pod is created on a cluster node, it first enters into the status `pending`. During this phase, all the containers inside the pod are starting up. Once all containers of the pod are running, the pod switches to status `running`. But it is important to notice that the pod only ever enters into this status if all containers run successfully.

If Kubernetes is requesting a pod to terminate, then it requests all its enclosed containers to terminate. Once all containers have been terminated with an exit code of zero, then the pod enters in its final status of `succeeded`.

What we have described so far corresponds to the happy path.

But the reality is harsh, and often the happy path is not what we will observe. Let us, thus, dive into some scenarios that lead the pod to end in status `failed`.

1. During the startup of the pod, at least one container is not able to run and fails with a non-zero exist code. In this case, the pod transitions from the `pending` status into the `failed` status.
2. While the pod in is in status running suddenly, one of the contained containers crashes or exits with a non-zero exit code, then the pod transitions from status `running` to status `failed`.
3. Kubernetes requests the pod to terminate. During the shutdown phase, at least one of the contained containers hangs (that is, does not respond in a timely manner) or exits with a non-zero exit code. Consequently, the pod will transition from status `running` to status `failed`.

You may have noticed that the preceding illustration does not contain all possible states a pod can be in. We have purposefully only shown the most important ones to not overcomplicate things. If you are interested in the full list of all possible states, then have a look at the following table:

State	Description
Pending:	The pod has been accepted by the Kubernetes system, but one or more of its containers have not yet started.
Running	All of the containers in the pod are running and ready.
Succeeded	All containers in the pod have terminated and been completed successfully.
Failed	One or more containers in the pod have terminated and failed.
Unknown	The state of the pod is unknown, which can occur when there is a problem with the communication between the control plane and the node where the pod is running.

When a pod is created on a cluster node, it first enters into the status `pending`. During this phase, all the containers inside the pod are starting up. Once all containers of the pod are running, the pod switches to status `running`. But it is important to notice that the pod only ever enters into this status if all containers run successfully.

If Kubernetes is requesting a pod to terminate, then it requests all its enclosed containers to terminate. Once all containers have been terminated with an exit code of zero, then the pod enters in its final status of `succeeded`.

What we have described so far corresponds to the happy path.

But the reality is harsh, and often the happy path is not what we will observe. Let us, thus, dive into some scenarios that lead the pod to end in status `failed`.

1. During the startup of the pod, at least one container is not able to run and fails with a non-zero exist code. In this case, the pod transitions from the `pending` status into the `failed` status.
2. While the pod in is in status running suddenly, one of the contained containers crashes or exits with a non-zero exit code, then the pod transitions from status `running` to status `failed`.
3. Kubernetes requests the pod to terminate. During the shutdown phase, at least one of the contained containers hangs (that is, does not respond in a timely manner) or exits with a non-zero exit code. Consequently, the pod will transition from status `running` to status `failed`.

You may have noticed that the preceding illustration does not contain all possible states a pod can be in. We have purposefully only shown the most important ones to not overcomplicate things. If you are interested in the full list of all possible states, then have a look at the following table:

State	Description
Pending:	The pod has been accepted by the Kubernetes system, but one or more of its containers have not yet started.
Running	All of the containers in the pod are running and ready.
Succeeded	All containers in the pod have terminated and been completed successfully.
Failed	One or more containers in the pod have terminated and failed.
Unknown	The state of the pod is unknown, which can occur when there is a problem with the communication between the control plane and the node where the pod is running.

CrashLoopBackoff	The pod is restarting continuously, likely due to a failure in one of its containers.
Completed	The pod has been terminated, and all containers have been completed, but at least one container has a non-zero exit status.
Unschedulable	The pod cannot be scheduled to a node due to resource constraints or other reasons.
Terminating	The pod is in the process of being terminated.

Table 10.1: *List of possible states*

These statuses provide a way to monitor the health and behavior of pods in a Kubernetes cluster and to identify and resolve any issues that may arise.

Now that we have some high-level understanding of what a pod is and how it can behave, let us get our hands dirty and create and manipulate some pods.

Working with pods

When creating a pod in a Kubernetes cluster, we can use either an imperative or a declarative approach. Earlier in this book, we used the Docker CLI, and more specifically, the `docker container run` command to imperatively run containers, and we used the Docker Compose YAML file or manifest to declaratively run one-to-many containers.

When dealing with Kubernetes, we can use the `kubectl` tool to imperatively run pods, and we can define so-called pod specifications in YAML to declaratively run a pod. In this chapter, we are going to concentrate on the latter approach. Let us, thus, start by defining a simple pod specification:

1. Open a new terminal window, and navigate to the folder where you have your source code accompanying this book, e.g.:

   ```
   cd ~/src
   ```

2. Create a new subfolder `ch10`, for this chapter

   ```
   mkdir ch10
   ```

3. Copy the sample application `demo-app` from *Chapter 9, Establishing an Automated Build Pipeline*, into this folder:

   ```
   cp -rf ch09/demo-app ch010/
   ```

CrashLoopBackoff	The pod is restarting continuously, likely due to a failure in one of its containers.
Completed	The pod has been terminated, and all containers have been completed, but at least one container has a non-zero exit status.
Unschedulable	The pod cannot be scheduled to a node due to resource constraints or other reasons.
Terminating	The pod is in the process of being terminated.

***Table 10.1:** List of possible states*

These statuses provide a way to monitor the health and behavior of pods in a Kubernetes cluster and to identify and resolve any issues that may arise.

Now that we have some high-level understanding of what a pod is and how it can behave, let us get our hands dirty and create and manipulate some pods.

Working with pods

When creating a pod in a Kubernetes cluster, we can use either an imperative or a declarative approach. Earlier in this book, we used the Docker CLI, and more specifically, the **`docker container run`** command to imperatively run containers, and we used the Docker Compose YAML file or manifest to declaratively run one-to-many containers.

When dealing with Kubernetes, we can use the **`kubectl`** tool to imperatively run pods, and we can define so-called pod specifications in YAML to declaratively run a pod. In this chapter, we are going to concentrate on the latter approach. Let us, thus, start by defining a simple pod specification:

1. Open a new terminal window, and navigate to the folder where you have your source code accompanying this book, e.g.:

   ```
   cd ~/src
   ```

2. Create a new subfolder **`ch10`**, for this chapter

   ```
   mkdir ch10
   ```

3. Copy the sample application **`demo-app`** from *Chapter 9, Establishing an Automated Build Pipeline*, into this folder:

   ```
   cp -rf ch09/demo-app ch010/
   ```

4. In the root of the demo-app, create a new folder called **.kube**. It will contain all our Kubernetes-related specifications:

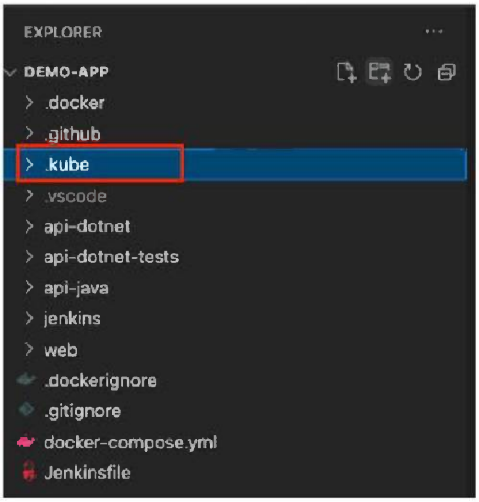


Figure 10.10: .kube folder in the demo application

5. Navigate to the demo-app folder and build the Docker image for the Web project:

```
docker image build -t <user-id>/web:ch10
```

Please make sure that you replace <user-id> with your own Docker user ID. In our case, we will use **dockerforjobseekers**, resulting in the following:

```
docker image build -t dockerforjobseekers/web:ch10 web
```

6. Optionally, and if not done so earlier, login to Docker hub:

```
docker login
```

and enter your credentials when asked

7. Push the image to the Docker hub:

```
docker image push <user-id>/web:ch10
```

Again, replace the <user-id> with your own Docker user ID.

8. Open the sample application in VS Code:

```
code ch10/demo-app
```

4. In the root of the demo-app, create a new folder called **.kube**. It will contain all our Kubernetes-related specifications:

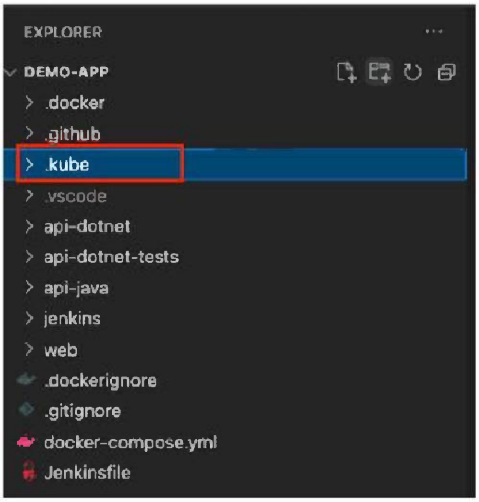


Figure 10.10: .kube folder in the demo application

5. Navigate to the demo-app folder and build the Docker image for the Web project:

```
docker image build -t <user-id>/web:ch10
```

Please make sure that you replace <user-id> with your own Docker user ID. In our case, we will use **dockerforjobseekers**, resulting in the following:

```
docker image build -t dockerforjobseekers/web:ch10 web
```

6. Optionally, and if not done so earlier, login to Docker hub:

```
docker login
```

and enter your credentials when asked

7. Push the image to the Docker hub:

```
docker image push <user-id>/web:ch10
```

Again, replace the <user-id> with your own Docker user ID.

8. Open the sample application in VS Code:

```
code ch10/demo-app
```

9. To the `.kube` folder, add a file `web-pod.yaml` with the following content:

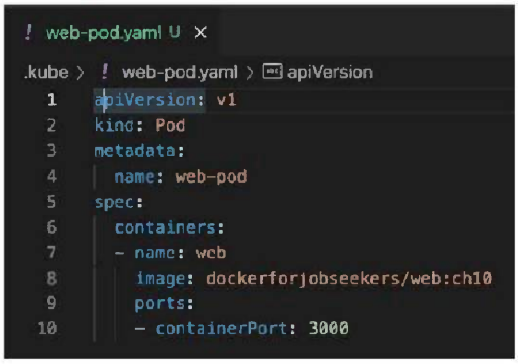


Figure 10.11: Pod specification for the Web container

- **Line 1:** Each specification in Kubernetes starts with the version information. Pods have been around for quite some time, and thus, the API version is `v1`.
- **Line 2:** The second line specifies the `kind` of Kubernetes object or resource we want to define. Obviously, in this case, we want to specify a Pod.
- **Line 3:** Next follows a block containing `metadata`. At a bare minimum, we need to give the pod a `name`. Here, we call it `web-pod`.
- **Line 5:** The next block that follows is the `spec` block, which contains the specification of the pod. The most important part (and the only one in this simple sample) is a list of all containers that are part of this pod.
- **Lines 7:** We only have one container here, but multiple containers are possible. The name we choose for our container is `web`.
- **Line 8:** The container image is `dockerforjobseekers/web:ch10`. In your case, please replace `dockerforjobseekers` with your own Docker user ID.
- **Lines 9–10:** Finally, we define a list of `ports` the container is exposing.

10. Make sure that your Docker Desktop is running Kubernetes:
    a. Open the Docker Desktop dashboard
    b. Select the cogwheel in the header of the dashboard leading you to the Preferences screen

9. To the `.kube` folder, add a file `web-pod.yaml` with the following content:

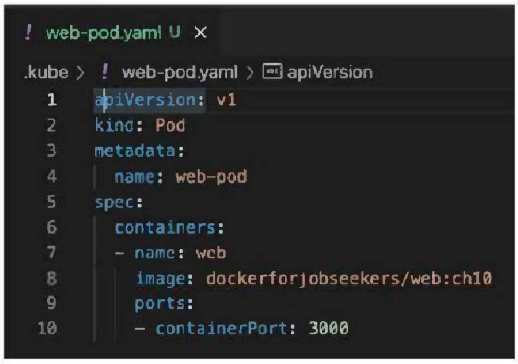


Figure 10.11: Pod specification for the Web container

- **Line 1:** Each specification in Kubernetes starts with the version information. Pods have been around for quite some time, and thus, the API version is `v1`.
- **Line 2:** The second line specifies the `kind` of Kubernetes object or resource we want to define. Obviously, in this case, we want to specify a Pod.
- **Line 3:** Next follows a block containing `metadata`. At a bare minimum, we need to give the pod a `name`. Here, we call it `web-pod`.
- **Line 5:** The next block that follows is the `spec` block, which contains the specification of the pod. The most important part (and the only one in this simple sample) is a list of all containers that are part of this pod.
- **Lines 7:** We only have one container here, but multiple containers are possible. The name we choose for our container is `web`.
- **Line 8:** The container image is `dockerforjobseekers/web:ch10`. In your case, please replace `dockerforjobseekers` with your own Docker user ID.
- **Lines 9–10:** Finally, we define a list of `ports` the container is exposing.

10. Make sure that your Docker Desktop is running Kubernetes:
    a. Open the Docker Desktop dashboard
    b. Select the cogwheel in the header of the dashboard leading you to the Preferences screen

c. On the **Preferences** screen, select the **Kubernetes** tab and make sure the **Enable Kubernetes** checkbox is checked:

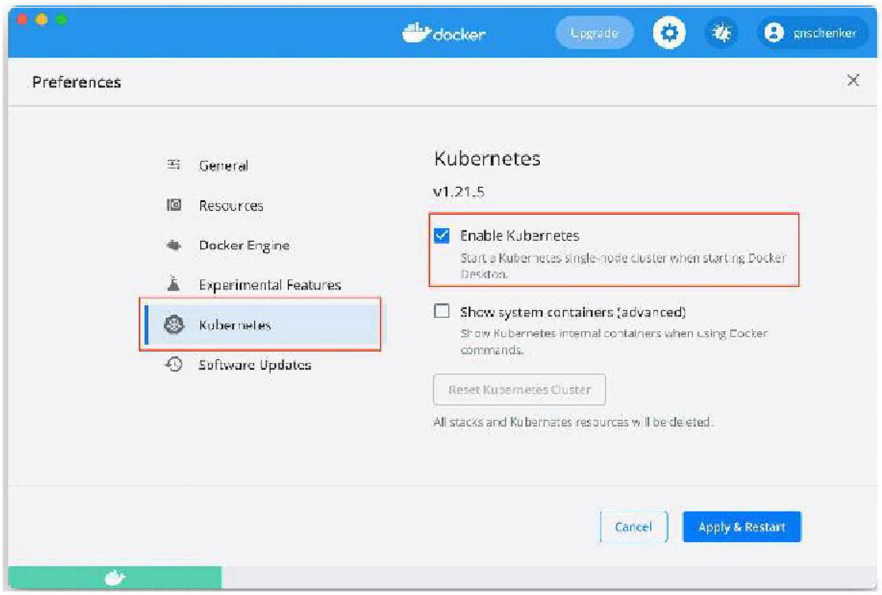


Figure 10.12: Enabling Kubernetes in Docker Desktop

d. Click the **Apply & Restart** button.

e. Wait until Kubernetes is installed and running. This may take a few minutes.

11. Back in your terminal window, use the `kubectl` tool to create and run the pod defined in the previous specification

```
kubectl create -f .kube/web-pod.yaml
```

the response should be the following:

```
pod/web-pod created
```

12. Immediately check if the pod is running:

```
kubectl get pods
```

which should show a list of all running pods on your cluster. In this case, the list should look like the following:

```
NAME      READY   STATUS    RESTARTS   AGE
web-pod   1/1     Running   0          2m28s
```

c. On the **Preferences** screen, select the **Kubernetes** tab and make sure the **Enable Kubernetes** checkbox is checked:

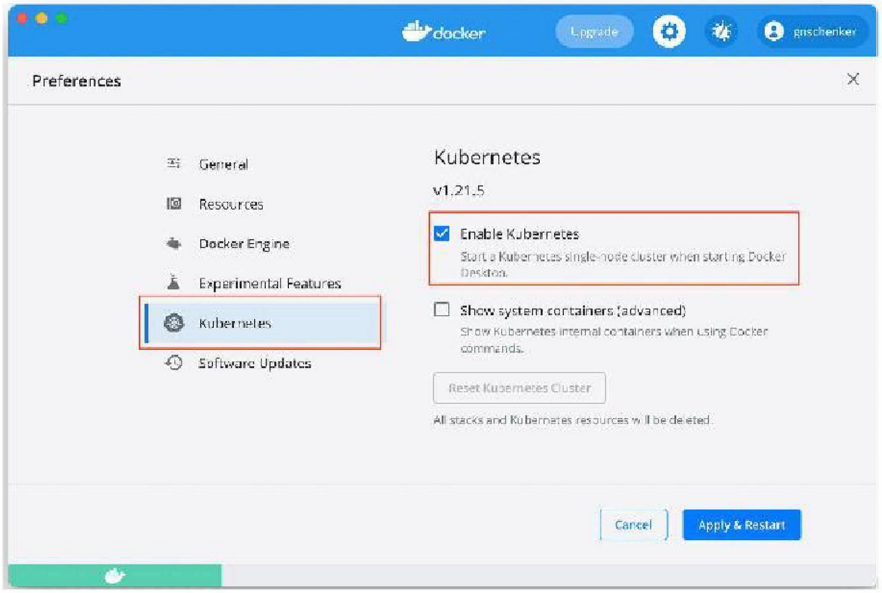


Figure 10.12: Enabling Kubernetes in Docker Desktop

d. Click the **Apply & Restart** button.

e. Wait until Kubernetes is installed and running. This may take a few minutes.

11. Back in your terminal window, use the `kubectl` tool to create and run the pod defined in the previous specification

```
kubectl create -f .kube/web-pod.yaml
```

the response should be the following:

```
pod/web-pod created
```

12. Immediately check if the pod is running:

```
kubectl get pods
```

which should show a list of all running pods on your cluster. In this case, the list should look like the following:

```
NAME      READY   STATUS    RESTARTS   AGE
web-pod   1/1     Running   0          2m28s
```

The preceding output is explained in detail:

- **NAME:** shows the name of our pod as we have defined on Line 4 of the specification
- **READY:** shows the total number of containers in the pod and the number of containers that are ready. In our case, we only have a single container, and apparently, it is up and running
- **STATUS:** the status of the pod. Since all containers are running, the status of the pod is running too
- **AGE:** shows the amount of time the pod has been up and running

13. Next, we want to describe the running pod to get some more valuable information:

```
kubectl describe pod/web-pod
```

The output is quite rich in information and should look similar to the following:

```
Name:         web-pod
Namespace:    default
Priority:     0
Node:         docker-desktop/192.168.65.4
Start Time:   Sun, 07 Nov 2021 13:17:43 +0100
Labels:       <none>
Annotations:  <none>
Status:       Running
IP:           10.1.0.6
IPs:
  IP:  10.1.0.6
Containers:
  web:
    Container ID:   docker://d02114f667df479cf7a4c0c64cb01a6ba8f7509531f94a1aca9179315d50290a
    Image:          dockerforjobseekers/web:ch10
    Image ID:       docker-pullable://dockerforjobseekers/web@sha256:3455fa425fe93426cacce9e61b684eaf8411cb88889a6d6356bf80196246a778
    Port:           3000/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Sun, 07 Nov 2021 13:17:44 +0100
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-9pcrl (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  kube-api-access-9pcrl:
    Type:                    Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   BestEffort
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason     Age    From               Message
  ----    ------     ----   ----               -------
  Normal  Scheduled  8m18s  default-scheduler  Successfully assigned default/web-pod to docker-desktop
  Normal  Pulled     8m17s  kubelet            Container image "dockerforjobseekers/web:ch10" already present on machine
  Normal  Created    8m17s  kubelet            Created container web
  Normal  Started    8m17s  kubelet            Started container web
```

Figure 10.13: Describing a pod

The preceding output is explained in detail:

- **NAME:** shows the name of our pod as we have defined on Line 4 of the specification
- **READY:** shows the total number of containers in the pod and the number of containers that are ready. In our case, we only have a single container, and apparently, it is up and running
- **STATUS:** the status of the pod. Since all containers are running, the status of the pod is running too
- **AGE:** shows the amount of time the pod has been up and running

13. Next, we want to describe the running pod to get some more valuable information:

    ```
    kubectl describe pod/web-pod
    ```

    The output is quite rich in information and should look similar to the following:

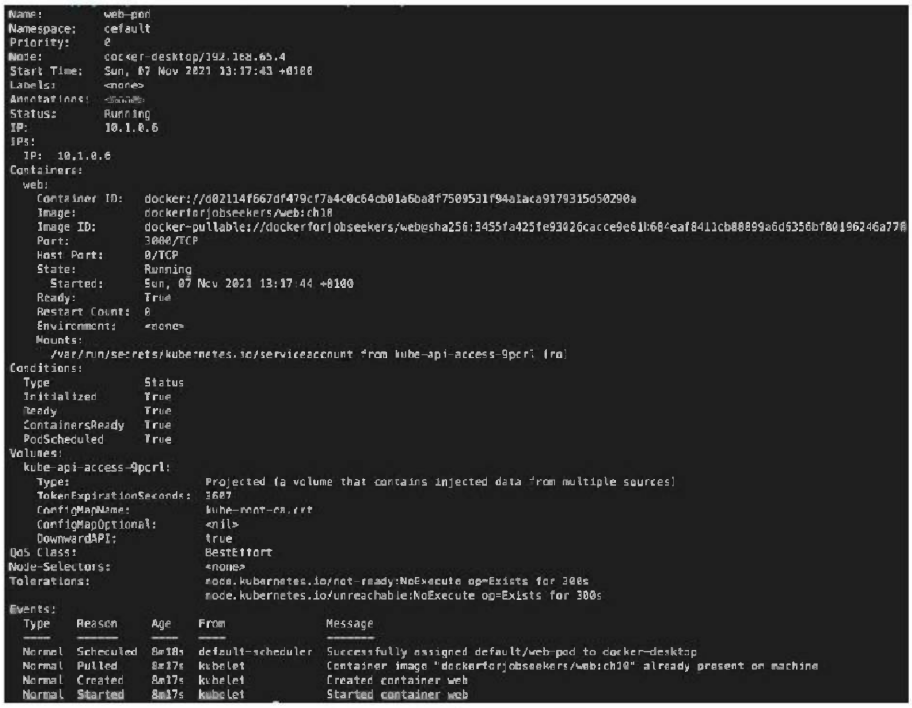


***Figure 10.13:** Describing a pod*

The described command gives us a wealth of information about the pod. Particularly interesting is the series of events that happened to this pod during its life cycle. This list is shown at the bottom of the output. Take a moment to browse through this output and familiarize yourself with what is available.

14. Next, we want to use the **kubectl exec** command to run a process within the container running inside the pod. In our case, it will be a simple shell:

    ```
    kubectl exec -it web-pod -- /bin/bash
    ```

    you will be seeing the following output:

    ```
    root@web-pod:/app#
    ```

    indicating that you are indeed in a Bash shell running inside the Web container.

15. Try to run some commands inside this container, and when done, type **exit** to end the shell session. For example:

    ```
    # list the contents of the /app folder
    ls -al
    # list all environment variables defined inside the container
    export
    # leave the container
    exit
    ```

16. Back on the host in the terminal window, run the following command to output any logs generated by the container in the pod:

    ```
    kubectl logs pod/web-pod
    ```

    you should get the following:

    ```
    Demo Web application listening at http://localhost:3000
    ```

17. We now want to access the **web** application service running inside the pod now. For this, we need to use the **kubectl port-forward** command:

    a. Run the following command to forward the host port 8080 to the container port 3000 of the Web container running in the **web-pod** pod:

    ```
    kubectl port-forward pod/web-pod 8080:3000
    ```

    Note: this command is blocking, and the output should be similar to the following:

    ```
    web-pod 8080:3000
    Forwarding from 127.0.0.1:8080 -> 3000
    Forwarding from [::1]:8080 -> 3000
    ```

The described command gives us a wealth of information about the pod. Particularly interesting is the series of events that happened to this pod during its life cycle. This list is shown at the bottom of the output. Take a moment to browse through this output and familiarize yourself with what is available.

14. Next, we want to use the **kubectl exec** command to run a process within the container running inside the pod. In our case, it will be a simple shell:

    ```
    kubectl exec -it web-pod -- /bin/bash
    ```

    you will be seeing the following output:

    ```
    root@web-pod:/app#
    ```

    indicating that you are indeed in a Bash shell running inside the Web container.

15. Try to run some commands inside this container, and when done, type **exit** to end the shell session. For example:

    ```
    # list the contents of the /app folder
    ls -al
    # list all environment variables defined inside the container
    export
    # leave the container
    exit
    ```

16. Back on the host in the terminal window, run the following command to output any logs generated by the container in the pod:

    ```
    kubectl logs pod/web-pod
    ```

    you should get the following:

    ```
    Demo Web application listening at http://localhost:3000
    ```

17. We now want to access the **web** application service running inside the pod now. For this, we need to use the **kubectl port-forward** command:

    a. Run the following command to forward the host port 8080 to the container port 3000 of the Web container running in the **web-pod** pod:

    ```
    kubectl port-forward pod/web-pod 8080:3000
    ```

    Note: this command is blocking, and the output should be similar to the following:

    ```
    web-pod 8080:3000
    Forwarding from 127.0.0.1:8080 -> 3000
    Forwarding from [::1]:8080 -> 3000
    ```

b. Open a second terminal window or pane and use the `curl` command to access the `web` application service:

```
curl localhost:8080
```

The answer should be the following:

```
<html><head><title>Demo Application</title></
head><body><h1>Demo Web application!</h1></body></html>
```

Please note that in the first terminal window, each time you run the curl command, the following output is generated:

```
Handling connection for 8080
```

c. In the terminal window where the port forwarder command runs, press CTRL-c to end the port forwarding.

18. Once you are done experimenting with the pod, you will want to terminate it. Use the `delete` command to do so:

```
kubectl delete pod web-pod
```

After a short moment, the following output is generated:

```
pod "web-pod" deleted
```

Double-check that the pod is indeed gone with the following:

```
kubectl get pods
```

The pod should have disappeared from the list, and the answer is:

```
No resources found in default namespace.
```

With that, we have familiarized ourselves with the most important aspects of a pod. Next, we will discuss ReplicaSets and Deployments.

ReplicaSets and Deployments

A single pod in an environment with high availability requirements is insufficient. What if the pod crashes? What if we need to update the application running inside the pod but cannot afford any service interruption? These questions and more indicate that pods alone are not enough, and we need a higher-level concept that can manage multiple instances of the same pod. In Kubernetes, the ReplicaSet is used to define and manage such a collection of identical pods that are running on different cluster nodes. Among other things, a ReplicaSet defines which container images are used by the containers running inside a pod and how many instances of the pod will run in the cluster. These properties and many others are called the desired state.

b. Open a second terminal window or pane and use the `curl` command to access the `web` application service:

```
curl localhost:8080
```

The answer should be the following:

```
<html><head><title>Demo Application</title></
head><body><h1>Demo Web application!</h1></body></html>
```

Please note that in the first terminal window, each time you run the curl command, the following output is generated:

```
Handling connection for 8080
```

c. In the terminal window where the port forwarder command runs, press CTRL-c to end the port forwarding.

18. Once you are done experimenting with the pod, you will want to terminate it. Use the `delete` command to do so:

```
kubectl delete pod web-pod
```

After a short moment, the following output is generated:

```
pod "web-pod" deleted
```

Double-check that the pod is indeed gone with the following:

```
kubectl get pods
```

The pod should have disappeared from the list, and the answer is:

```
No resources found in default namespace.
```

With that, we have familiarized ourselves with the most important aspects of a pod. Next, we will discuss ReplicaSets and Deployments.

ReplicaSets and Deployments

A single pod in an environment with high availability requirements is insufficient. What if the pod crashes? What if we need to update the application running inside the pod but cannot afford any service interruption? These questions and more indicate that pods alone are not enough, and we need a higher-level concept that can manage multiple instances of the same pod. In Kubernetes, the ReplicaSet is used to define and manage such a collection of identical pods that are running on different cluster nodes. Among other things, a ReplicaSet defines which container images are used by the containers running inside a pod and how many instances of the pod will run in the cluster. These properties and many others are called the desired state.

The ReplicaSet is responsible for reconciling the desired state at all times if the actual state ever deviates from it. In the following image, we show a Kubernetes ReplicaSet:

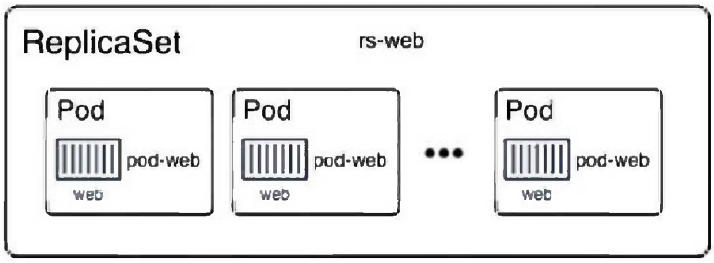


Figure 10.14: Kubernetes ReplicaSet

In the preceding diagram, we can see a ReplicaSet called `rs-web`, which governs a number of pods. The pods are called `pod-web`. The ReplicaSet is responsible for making sure that, at any given time, there is always the desired number of pods running. If one of the pods crashes for whatever reason, the ReplicaSet schedules a new pod on a node with free resources instead. If there are more pods than the desired number, then the ReplicaSet kills superfluous pods. With this, we can say that the ReplicaSet guarantees a self-healing and scalable set of pods. There is no limit to how many pods a ReplicaSet can hold.

Working with a ReplicaSet

It is always helpful to do some hands-on exercises to discover what the theory stated previously about ReplicaSets actually means. Thus, let us dive into it and define such an object for the `web` container we have been using for the preceding pod.

1. Let us start with the specification. Add a new file called `rs-web.yaml` to the `.kube` folder and give it this content:

The ReplicaSet is responsible for reconciling the desired state at all times if the actual state ever deviates from it. In the following image, we show a Kubernetes ReplicaSet:

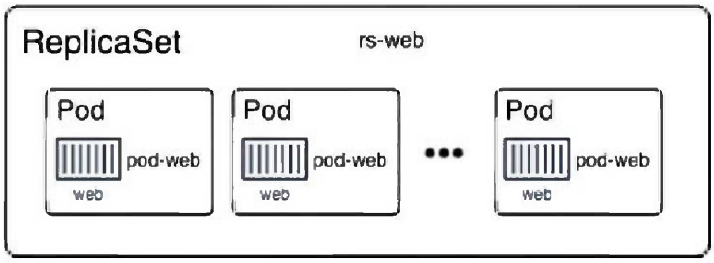


Figure 10.14: Kubernetes ReplicaSet

In the preceding diagram, we can see a ReplicaSet called **rs-web**, which governs a number of pods. The pods are called **pod-web**. The ReplicaSet is responsible for making sure that, at any given time, there is always the desired number of pods running. If one of the pods crashes for whatever reason, the ReplicaSet schedules a new pod on a node with free resources instead. If there are more pods than the desired number, then the ReplicaSet kills superfluous pods. With this, we can say that the ReplicaSet guarantees a self-healing and scalable set of pods. There is no limit to how many pods a ReplicaSet can hold.

Working with a ReplicaSet

It is always helpful to do some hands-on exercises to discover what the theory stated previously about ReplicaSets actually means. Thus, let us dive into it and define such an object for the **web** container we have been using for the preceding pod.

1. Let us start with the specification. Add a new file called **rs-web.yaml** to the **.kube** folder and give it this content:

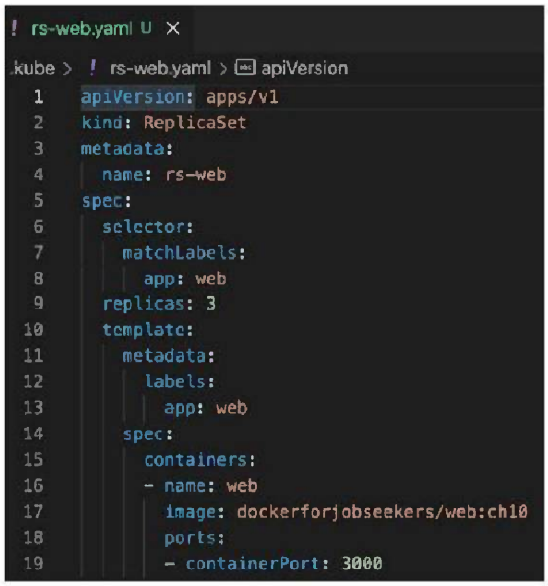


Figure 10.15: ReplicaSet specification for the Web container

a. **Line 1:** Once again, we have to define the version of the API we use

b. **Line 2:** This time, the kind of object we are defining is a **`ReplicaSet`**

c. **Lines 3–4:** We define the **`metadata`** for the **`ReplicaSet`**. At a minimum, we need to define a **`name`**

d. **Line 5:** This **`spec`** block specifies what the **`ReplicaSet`** is

e. **Lines 6–8:** We define a selector which determines the pods that will be part of the **`ReplicaSet`**. In this case, the pods need to have a label called app with a value of **`web`** to be part of the ReplicaSet.

f. **Line 9:** We define how many (identical) pods we want to have run at any given time

g. **Lines 10–19:** In the **`template`** section, we pretty much find the same information we had in the container specification. We have the **`metadata`** section and the container specification. Please make sure you compare this part of the specification with the pod specification in the file **`web-pod.yaml`**.

2. Run the **`ReplicaSet`** with the following:

```
kubectl create -f .kube/rs-web
```

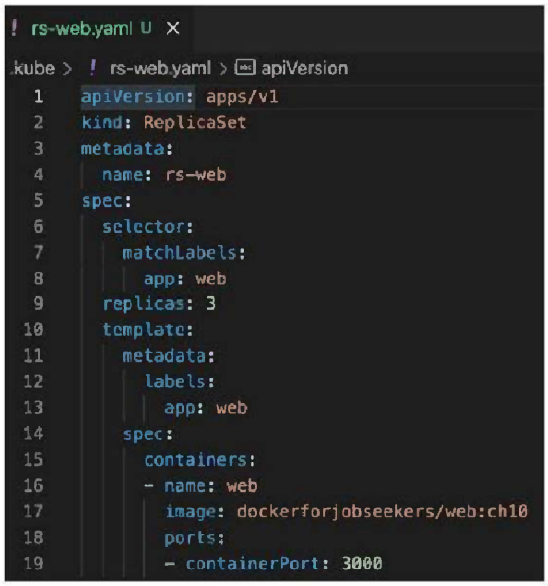


Figure 10.15: ReplicaSet specification for the Web container

a. **Line 1:** Once again, we have to define the version of the API we use

b. **Line 2:** This time, the kind of object we are defining is a **`ReplicaSet`**

c. **Lines 3–4:** We define the **`metadata`** for the **`ReplicaSet`**. At a minimum, we need to define a **`name`**

d. **Line 5:** This **`spec`** block specifies what the **`ReplicaSet`** is

e. **Lines 6–8:** We define a selector which determines the pods that will be part of the **`ReplicaSet`**. In this case, the pods need to have a label called app with a value of **`web`** to be part of the ReplicaSet.

f. **Line 9:** We define how many (identical) pods we want to have run at any given time

g. **Lines 10–19:** In the **`template`** section, we pretty much find the same information we had in the container specification. We have the **`metadata`** section and the container specification. Please make sure you compare this part of the specification with the pod specification in the file **`web-pod.yaml`**.

2. Run the **`ReplicaSet`** with the following:

```
kubectl create -f .kube/rs-web
```

The output should be:

```
replicaset.apps/rs-web created
```

3. Use `kubectl` to get the list of all objects running:

   ```
   kubectl get all
   ```

   The output should look similar to the following:

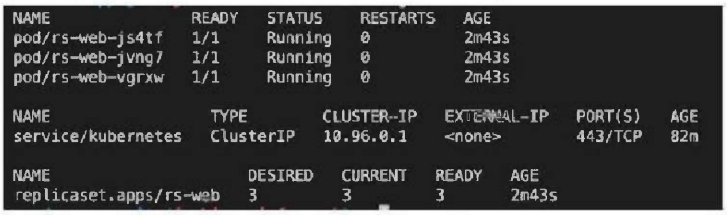


Figure 10.16: Output of kubectl get all for a ReplicaSet

Ignore for a moment the object `service/kubernetes` as it is not related to the ReplicaSet we just created. We have one entry at the bottom of the preceding picture called `replicaset.apps/rs-web`. This is our ReplicaSet. In the column DESIRED, we see the number of desired instances of the pod that we want. In the column CURRENT, we see the number of instances effectively running at this very instant. The ReplicaSet will try its best to keep the number in CURRENT equal to the one in DESIRED. The column READY shows us how many of the pods report status running. Finally, the column AGE tells us how long this ReplicaSet has been existing.

At the top of the preceding image, we see the list of pods that are running as part of this ReplicaSet. Notice the names of the pods. Kubernetes automatically appends a unique alpha-numeric string to the name given to the replicaset. An individual pod can then be identified by this name, e.g., `pod/rs-web-js4tf` in our example.

4. Now, let us try to see if the ReplicaSet indeed will re-establish the number of instances if we kill a pod. For this, run the following command in the terminal window:

   ```
   kubectl get pods -w
   ```

   This will list the pods running on the cluster and refresh the output every few seconds

5. Open another terminal window and run the following command to kill one of the pods:

   ```
   kubectl delete pod/rs-web-js4tf
   ```

The output should be:

```
replicaset.apps/rs-web created
```

3. Use `kubectl` to get the list of all objects running:

   ```
   kubectl get all
   ```

   The output should look similar to the following:

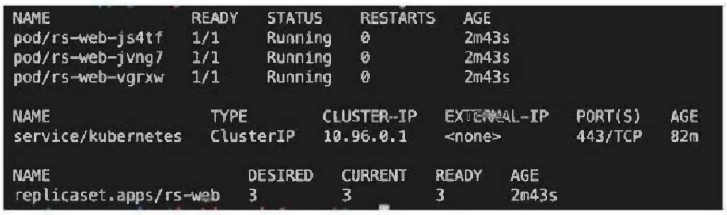


Figure 10.16: Output of kubectl get all for a ReplicaSet

Ignore for a moment the object `service/kubernetes` as it is not related to the ReplicaSet we just created. We have one entry at the bottom of the preceding picture called `replicaset.apps/rs-web`. This is our ReplicaSet. In the column DESIRED, we see the number of desired instances of the pod that we want. In the column CURRENT, we see the number of instances effectively running at this very instant. The ReplicaSet will try its best to keep the number in CURRENT equal to the one in DESIRED. The column READY shows us how many of the pods report status running. Finally, the column AGE tells us how long this ReplicaSet has been existing.

At the top of the preceding image, we see the list of pods that are running as part of this ReplicaSet. Notice the names of the pods. Kubernetes automatically appends a unique alpha-numeric string to the name given to the replicaset. An individual pod can then be identified by this name, e.g., `pod/rs-web-js4tf` in our example.

4. Now, let us try to see if the ReplicaSet indeed will re-establish the number of instances if we kill a pod. For this, run the following command in the terminal window:

   ```
   kubectl get pods -w
   ```

   This will list the pods running on the cluster and refresh the output every few seconds

5. Open another terminal window and run the following command to kill one of the pods:

   ```
   kubectl delete pod/rs-web-js4tf
   ```

make sure you replace the **`rs-web-js4tf`** with the name of one of your own pods that are listed in your first terminal window.

Observe what is happening. In the authors' case, the output looked like the following:

```
NAME          READY   STATUS              RESTARTS   AGE
rs-web-js4tf  1/1     Running             0          16h
rs-web-jvng7  1/1     Running             0          16h
rs-web-vgrxw  1/1     Running             0          16h
rs-web-js4tf  1/1     Terminating         0          16h
rs-web-kswk4  0/1     Pending             0          0s
rs-web-kswk4  0/1     Pending             0          0s
rs-web-kswk4  0/1     ContainerCreating   0          0s
rs-web-kswk4  1/1     Running             0          2s
rs-web-js4tf  0/1     Terminating         0          16h
rs-web-js4tf  0/1     Terminating         0          16h
rs-web-js4tf  0/1     Terminating         0          16h
```

Figure 10.17: Killing a pod of a replicaset

Pod **`rs-web-js4tf`** is being killed whilst immediately a new instance **`rs-web-kswk4`** is created in its place. This proves that the ReplicaSet indeed makes sure that, at all times, we have the desired number of pods running.

6. We will also discover that event of creating a new pod when describing our ReplicaSet with the following:

   `kubectl describe replicaset/rs-web`

   The output should look similar to the following:

```
Name:         rs-web
Namespace:    default
Selector:     app=web
Labels:       <none>
Annotations:  <none>
Replicas:     3 current / 3 desired
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=web
  Containers:
   web:
    Image:        dockerforjobseekers/web:ch10
    Port:         3000/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age    From                   Message
  ----    ------            ----   ----                   -------
  Normal  SuccessfulCreate  6m15s  replicaset-controller  Created pod: rs-web-kswk4
```

Figure 10.18: Describing a replicaset

make sure you replace the **`rs-web-js4tf`** with the name of one of your own pods that are listed in your first terminal window.

Observe what is happening. In the authors' case, the output looked like the following:

```
NAME          READY   STATUS              RESTARTS   AGE
rs-web-js4tf  1/1     Running             0          16h
rs-web-jvng7  1/1     Running             0          16h
rs-web-vgrxw  1/1     Running             0          16h
rs-web-js4tf  1/1     Terminating         0          16h
rs-web-kswk4  0/1     Pending             0          0s
rs-web-kswk4  0/1     Pending             0          0s
rs-web-kswk4  0/1     ContainerCreating   0          0s
rs-web-kswk4  1/1     Running             0          2s
rs-web-js4tf  0/1     Terminating         0          16h
rs-web-js4tf  0/1     Terminating         0          16h
rs-web-js4tf  0/1     Terminating         0          16h
```

Figure 10.17: Killing a pod of a replicaset

Pod **`rs-web-js4tf`** is being killed whilst immediately a new instance **`rs-web-kswk4`** is created in its place. This proves that the ReplicaSet indeed makes sure that, at all times, we have the desired number of pods running.

6. We will also discover that event of creating a new pod when describing our ReplicaSet with the following:

   `kubectl describe replicaset/rs-web`

   The output should look similar to the following:

```
Name:         rs-web
Namespace:    default
Selector:     app=web
Labels:       <none>
Annotations:  <none>
Replicas:     3 current / 3 desired
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=web
  Containers:
   web:
    Image:        dockerforjobseekers/web:ch10
    Port:         3000/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age    From                   Message
  ----    ------            ----   ----                   -------
  Normal  SuccessfulCreate  6m15s  replicaset-controller  Created pod: rs-web-kswk4
```

Figure 10.18: Describing a replicaset

Notice the list of events at the end of the preceding output. We have exactly one event at this time. It is the event describing that the replicaset created the new pod `rs-web-kswk4`.

7. When done playing with the ReplicaSet please make sure to remove it from the cluster with the following command:

   ```
   kubectl delete replicaset/rs-web
   ```

   and double-check that all pods are removed with

   ```
   kubectl get pods -w
   ```

   It may take a moment until all pods are terminated and removed from the cluster. Stop the preceding command by pressing CTRL-c.

   Check that everything is clean with the following:

   ```
   kubectl get all
   ```

Now that we have a good grasp of what a Kubernetes `ReplicaSet` is, let us move on to the Kubernetes `Deployment`, which it turns out is kind of an extension of the `ReplicaSet`.

Working with a Deployment

Kubernetes takes the single-responsibility principle very seriously. All Kubernetes objects are designed to do one thing and one thing only, and they are designed to do this one thing very well. In this regard, we have to understand Kubernetes `ReplicaSets` and `Deployments`. A `ReplicaSet`, as we have learned, is responsible for achieving and reconciling the desired state of an application service. This means that the `ReplicaSet` manages a set of pods.

`Deployment` augments a `ReplicaSet` by providing rolling updates and rollback functionality on top of it. The following diagram shows the relationship between a `Deployment` to a `ReplicaSet`:

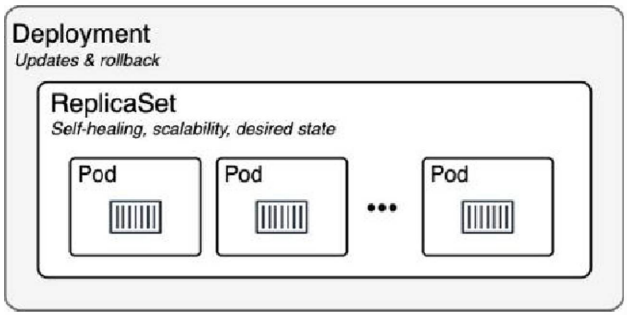


Figure 10.19: Kubernetes Deployment

Notice the list of events at the end of the preceding output. We have exactly one event at this time. It is the event describing that the replicaset created the new pod `rs-web-kswk4`.

7. When done playing with the ReplicaSet please make sure to remove it from the cluster with the following command:

   ```
   kubectl delete replicaset/rs-web
   ```

   and double-check that all pods are removed with

   ```
   kubectl get pods -w
   ```

   It may take a moment until all pods are terminated and removed from the cluster. Stop the preceding command by pressing CTRL-c.

   Check that everything is clean with the following:

   ```
   kubectl get all
   ```

Now that we have a good grasp of what a Kubernetes `ReplicaSet` is, let us move on to the Kubernetes `Deployment`, which it turns out is kind of an extension of the `ReplicaSet`.

Working with a Deployment

Kubernetes takes the single-responsibility principle very seriously. All Kubernetes objects are designed to do one thing and one thing only, and they are designed to do this one thing very well. In this regard, we have to understand Kubernetes `ReplicaSets` and `Deployments`. A `ReplicaSet`, as we have learned, is responsible for achieving and reconciling the desired state of an application service. This means that the `ReplicaSet` manages a set of pods.

`Deployment` augments a `ReplicaSet` by providing rolling updates and rollback functionality on top of it. The following diagram shows the relationship between a `Deployment` to a `ReplicaSet`:

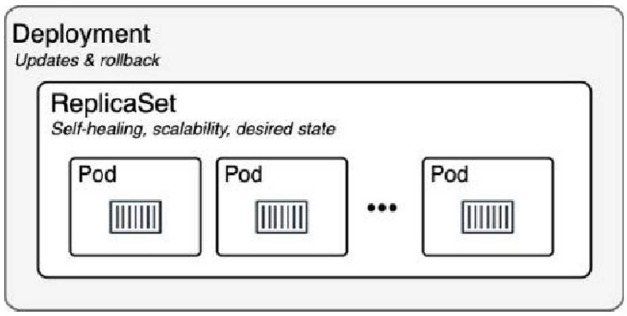


Figure 10.19: Kubernetes Deployment

In the preceding diagram, the `ReplicaSet` is defining and governing a set of identical pods. The main characteristics of the `ReplicaSet` are that it is self-healing, scalable, and always does its best to reconcile the desired state. Kubernetes `Deployment`, in turn, adds rolling updates and rollback functionality to this. In this regard, a deployment is really a wrapper object to a `ReplicaSet`.

As we have stated before, the theory is good, but let us practice a bit to get a feeling for what a deployment really is.

1. To the `.kube` folder, add a new file called `deployment-web.yaml` and add this content to it:



Figure 10.20: The specification for a Kubernetes Deployment for service Web

Looking at the preceding specification, you may notice that it is very similar or even identical to the specification of the `ReplicaSet` we discussed in the previous section. This is not an accident, as the `Deployment` is basically an extension of the `ReplicaSet`.

a. **Line 4:** We define the name of the deployment as `web-deployment`

b. **Line 6:** We associate a label named app with the value `web` with this deployment

c. **Line 8:** We specify that we want to run three instances of the pod containing the service `web`

In the preceding diagram, the **ReplicaSet** is defining and governing a set of identical pods. The main characteristics of the **ReplicaSet** are that it is self-healing, scalable, and always does its best to reconcile the desired state. Kubernetes **Deployment**, in turn, adds rolling updates and rollback functionality to this. In this regard, a deployment is really a wrapper object to a **ReplicaSet**.

As we have stated before, the theory is good, but let us practice a bit to get a feeling for what a deployment really is.

1. To the **.kube** folder, add a new file called **deployment-web.yaml** and add this content to it:



Figure 10.20: The specification for a Kubernetes Deployment for service Web

Looking at the preceding specification, you may notice that it is very similar or even identical to the specification of the **ReplicaSet** we discussed in the previous section. This is not an accident, as the **Deployment** is basically an extension of the **ReplicaSet**.

a. **Line 4:** We define the name of the deployment as **web-deployment**

b. **Line 6:** We associate a label named app with the value **web** with this deployment

c. **Line 8:** We specify that we want to run three instances of the pod containing the service **web**

d. **Lines 9 to 11:** We specify the selector that defines which pods are included in this deployment. It is all pods that have a label with name `app` and value `web`

e. **Lines 12 to 21:** We define the pods that will be part of this deployment.

2. Make sure your Kubernetes cluster is clean by using the `kubectl get all` command. Remove any leftover objects except of the default `service/kubernetes` service.

3. Create the Deployment using the preceding specification with the following:

```
kubectl create -f .kube/deployment-web.yaml
```

4. Analyze what objects have been created with:

```
kubectl get all
```

You should see something similar to the following:

```
→ demo-app git:(main) ✗ kubectl get all
NAME                                 READY   STATUS    RESTARTS   AGE
pod/web-deployment-656685fd4f-fxtd9   1/1     Running   0          10s
pod/web-deployment-656685fd4f-psfnt   1/1     Running   0          10s
pod/web-deployment-656685fd4f-pz9hx   1/1     Running   0          10s

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   5d23h

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/web-deployment      3/3     3            3           10s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/web-deployment-656685fd4f      3         3         3       10s
```

Figure 10.21: All objects created by the Kubernetes Deployment for service Web

In the preceding output, we can see that

a. Three pods have been created. Their names have the following general pattern:

```
<deployment-name>-<replicaset-hash>-<pod-hash>
```

where `<replicaset-hash>` is the hash given to the `ReplicaSet` object by Kubernetes; in our case, it has a value of `656685fd4f`. `<pod-hash>`, in turn, is the hash given to each pod by Kubernetes. The first pod in the list has the hash `fxtd9`.

b. At the bottom of the preceding output, we see that a `ReplicaSet` object has been created. Its name is constructed according to this pattern:

```
<deployment-name>-<replicaset-hash>
```

d. **Lines 9 to 11**: We specify the selector that defines which pods are included in this deployment. It is all pods that have a label with name `app` and value `web`

e. **Lines 12 to 21:** We define the pods that will be part of this deployment.

2. Make sure your Kubernetes cluster is clean by using the `kubectl get all` command. Remove any leftover objects except of the default `service/kubernetes` service.

3. Create the Deployment using the preceding specification with the following:

```
kubectl create -f .kube/deployment-web.yaml
```

4. Analyze what objects have been created with:

```
kubectl get all
```

You should see something similar to the following:

```
→ demo-app git:(main) ✗ kubectl get all
NAME                                 READY   STATUS    RESTARTS   AGE
pod/web-deployment-656685fd4f-fxtd9   1/1     Running   0          10s
pod/web-deployment-656685fd4f-psfnt   1/1     Running   0          10s
pod/web-deployment-656685fd4f-pz9hx   1/1     Running   0          10s

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   5d23h

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/web-deployment      3/3     3            3           10s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/web-deployment-656685fd4f      3         3         3       10s
```

Figure 10.21: All objects created by the Kubernetes Deployment for service Web

In the preceding output, we can see that

a. Three pods have been created. Their names have the following general pattern:

```
<deployment-name>-<replicaset-hash>-<pod-hash>
```

where `<replicaset-hash>` is the hash given to the `ReplicaSet` object by Kubernetes; in our case, it has a value of `656685fd4f`. `<pod-hash>`, in turn, is the hash given to each pod by Kubernetes. The first pod in the list has the hash `fxtd9`.

b. At the bottom of the preceding output, we see that a `ReplicaSet` object has been created. Its name is constructed according to this pattern:

```
<deployment-name>-<replicaset-hash>
```

c. Finally, a **Deployment** object with the name **web-deployment** has been created

5. Next, we want to show how to enable a rolling upgrade of the service **web**.

   a. Open the **web/views/index.pug** file and change the message on Line 5 to something like

   ```
   h1 Demo Web application (version 2.0)!
   ```

   b. Save the changes and create a new version of the image and push it to Docker Hub:

   ```
   docker image build -t dockerforjobseekers/web:ch10-v2.0 web

   docker push dockerforjobseekers/web:ch10-v2.0
   ```

   Make sure to replace the **dockerforjobseekers** user in the preceding commands with your own Docker user.

   c. Now use **kubectl** to replace the version of the image used by the deployment:

   ```
   kubectl set image deployment.v1.apps/web-deployment \
       web=dockerforjobseekers/web:ch10-v2.0
   ```

   Again, make sure to replace the **dockerforjobseekers** user above with your own Docker user.

   d. Use **kubectl** to observe how the pods get replaced with new versions:

   ```
   kubectl get pod -w
   ```

   And describe one of the new pods to verify that the new image has been used:

   ```
   kubectl describe pod/web-deployment-797d96b856-6x8lc
   ```

So far, so good; we have our service **web** deployed and managed by a Kubernetes **Deployment** object. But we cannot yet access the **web** application services running inside the pods from outside (without using the hack of a proxy). We will show next what needs to be done to make this possible.

c. Finally, a **Deployment** object with the name **web-deployment** has been created

5. Next, we want to show how to enable a rolling upgrade of the service **web**.

   a. Open the **web/views/index.pug** file and change the message on Line 5 to something like

   ```
   h1 Demo Web application (version 2.0)!
   ```

   b. Save the changes and create a new version of the image and push it to Docker Hub:

   ```
   docker image build -t dockerforjobseekers/web:ch10-v2.0 web

   docker push dockerforjobseekers/web:ch10-v2.0
   ```

   Make sure to replace the **dockerforjobseekers** user in the preceding commands with your own Docker user.

   c. Now use **kubectl** to replace the version of the image used by the deployment:

   ```
   kubectl set image deployment.v1.apps/web-deployment \
       web=dockerforjobseekers/web:ch10-v2.0
   ```

   Again, make sure to replace the **dockerforjobseekers** user above with your own Docker user.

   d. Use **kubectl** to observe how the pods get replaced with new versions:

   ```
   kubectl get pod -w
   ```

   And describe one of the new pods to verify that the new image has been used:

   ```
   kubectl describe pod/web-deployment-797d96b856-6x8lc
   ```

So far, so good; we have our service **web** deployed and managed by a Kubernetes **Deployment** object. But we cannot yet access the **web** application services running inside the pods from outside (without using the hack of a proxy). We will show next what needs to be done to make this possible.

Services

The basic idea of a `Service` is to group a set of Pod endpoints into a single resource. What does that mean? The moment we start to work with applications consisting of more than one application service, we need service discovery. The following diagram illustrates this problem:

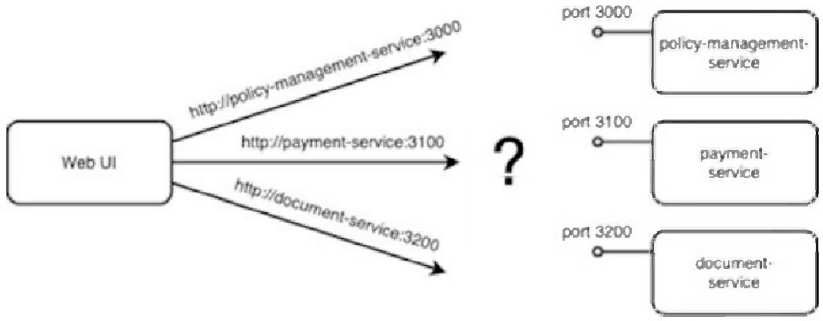


Figure 10.22: Service discovery

In the preceding diagram, we have a Web UI service that needs access to three other services: policy management, payments, and document service. The Web UI should never have to care about how and where to find those three services. In the code of the Web UI, we just want to use the name of the service we want to reach and its port number. A sample would be the following

URL **http://policy-management:3000**, which is used to access an instance of the policy management service.

In Kubernetes, the policy management service is represented by a `ReplicaSet` of pods. Due to the nature of highly distributed systems, we cannot assume that pods have stable endpoints. A pod can come and go in a blink of an eye. But that is a problem if we need to access the corresponding application service from an internal or external client. If we cannot rely on pod endpoints being stable, what else can we do? This is where Kubernetes services come into play. They are meant to provide stable endpoints to `ReplicaSets` or `Deployments`, as follows:

Services

The basic idea of a `Service` is to group a set of Pod endpoints into a single resource. What does that mean? The moment we start to work with applications consisting of more than one application service, we need service discovery. The following diagram illustrates this problem:

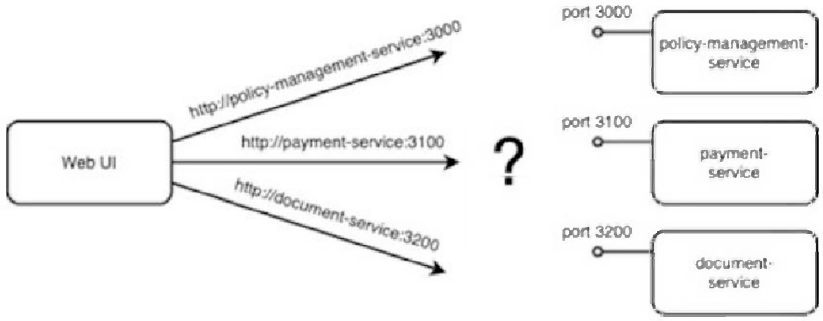


Figure 10.22: Service discovery

In the preceding diagram, we have a Web UI service that needs access to three other services: policy management, payments, and document service. The Web UI should never have to care about how and where to find those three services. In the code of the Web UI, we just want to use the name of the service we want to reach and its port number. A sample would be the following

URL **http://policy-management:3000**, which is used to access an instance of the policy management service.

In Kubernetes, the policy management service is represented by a `ReplicaSet` of pods. Due to the nature of highly distributed systems, we cannot assume that pods have stable endpoints. A pod can come and go in a blink of an eye. But that is a problem if we need to access the corresponding application service from an internal or external client. If we cannot rely on pod endpoints being stable, what else can we do? This is where Kubernetes services come into play. They are meant to provide stable endpoints to `ReplicaSets` or `Deployments`, as follows:

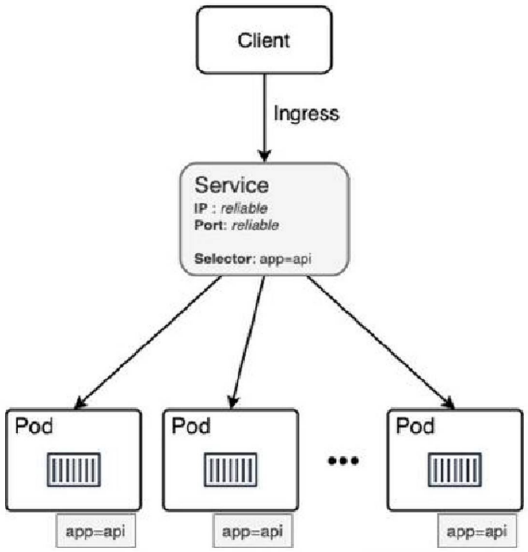


Figure 10.23: Kubernetes service providing stable endpoint

In the preceding diagram, in the center, we can see such a Kubernetes Service. It provides a reliable cluster-wide IP address, also called a **virtual IP (VIP)**, as well as a reliable Port that is unique in the whole cluster. The pods that the Kubernetes service is proxying are determined by the `Selector` defined in the service specification. Selectors are always based on labels. Every Kubernetes object can have zero to many labels assigned to it. In our case, the `Selector` is `app=api`; that is, all pods that have a label called app with a value of `api` are proxied.

There are different types of services in Kubernetes:

- **ClusterIP**: This type of service is the default, and it exposes the application service on a cluster-internal, that is, a private IP address. With this, the service is only reachable from within the cluster. Use this type if you do not want to publicly expose your application service.
- **NodePort**: This exposes the application service on each cluster node at a static port (called the `<node-port>`). A ClusterIP service, to which the NodePort service routes, is automatically created for you. With this, one can reach the NodePort service from outside the cluster via `<node-ip>:<node-port>`.
- **LoadBalancer**: This exposes your application service to the public by using a load balancer service of the cloud provider where you run the service in Kubernetes. This is what you normally want to use when you run in the

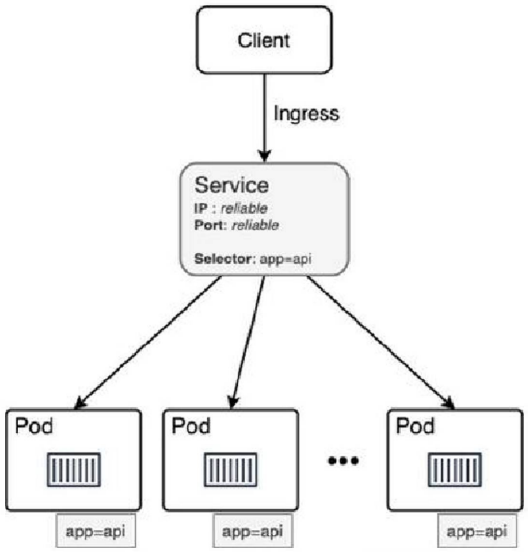


Figure 10.23: Kubernetes service providing stable endpoint

In the preceding diagram, in the center, we can see such a Kubernetes Service. It provides a reliable cluster-wide IP address, also called a **virtual IP (VIP)**, as well as a reliable Port that is unique in the whole cluster. The pods that the Kubernetes service is proxying are determined by the `Selector` defined in the service specification. Selectors are always based on labels. Every Kubernetes object can have zero to many labels assigned to it. In our case, the `Selector` is `app=api`; that is, all pods that have a label called app with a value of `api` are proxied.

There are different types of services in Kubernetes:

- **ClusterIP**: This type of service is the default, and it exposes the application service on a cluster-internal, that is, a private IP address. With this, the service is only reachable from within the cluster. Use this type if you do not want to publicly expose your application service.
- **NodePort**: This exposes the application service on each cluster node at a static port (called the `<node-port>`). A ClusterIP service, to which the NodePort service routes, is automatically created for you. With this, one can reach the NodePort service from outside the cluster via `<node-ip>:<node-port>`.
- **LoadBalancer**: This exposes your application service to the public by using a load balancer service of the cloud provider where you run the service in Kubernetes. This is what you normally want to use when you run in the

cloud using one of the many offerings of hosted Kubernetes. We will discuss some of those later in this chapter.

- **ExternalName:** Maps the Service to the contents of the `externalName` field (e.g. `web.example.com`) by returning a CNAME record with its value. No proxying of any kind is set up.
- **Headless:** There is one more service type called as Headless. Use a headless service when you want a pod grouping but do not need a stable IP address.

Now, let us get our hands dirty and play with such a Kubernetes service. As always, we are going to use our `web` component as an example.

1. To the `.kube` folder, add a new file called `service-web.yaml` and add this content to it:

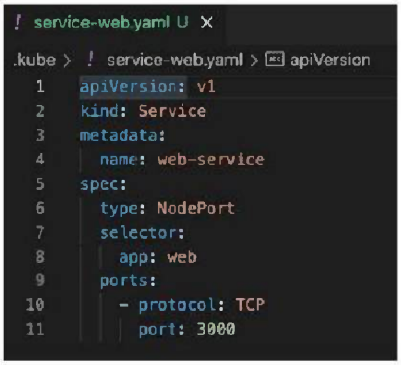


Figure 10.24: A Kubernetes service for the Web component

   a. **Line 4:** We set the name of this Service object to `web-service`.

   b. **Line 6:** We define the type of Service object we are using. Since the `web` component has to be accessible from outside of the cluster; this cannot be a Service object of the type `ClusterIP` and must be either of the type `NodePort` or `LoadBalancer`. We discussed the various types of Kubernetes services in the previous sections, so we will not go into further detail about this. In our sample, we are using a service of type `NodePort`.

   c. **Lines 7 and 8:** We define the filter criteria for the pods that this service will be a stable endpoint for. In this case, it is all the pods that have a label `app` with a value `web`.

   d. **Lines 9 to 11:** We specify that we want to expose port 3000 for access through the TCP protocol. Kubernetes will map container port 3000

cloud using one of the many offerings of hosted Kubernetes. We will discuss some of those later in this chapter.

- **ExternalName:** Maps the Service to the contents of the `externalName` field (e.g. `web.example.com`) by returning a CNAME record with its value. No proxying of any kind is set up.
- **Headless:** There is one more service type called as Headless. Use a headless service when you want a pod grouping but do not need a stable IP address.

Now, let us get our hands dirty and play with such a Kubernetes service. As always, we are going to use our `web` component as an example.

1. To the `.kube` folder, add a new file called `service-web.yaml` and add this content to it:

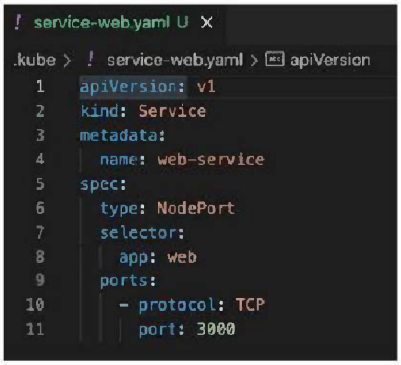


Figure 10.24: A Kubernetes service for the Web component

   a. **Line 4:** We set the name of this Service object to `web-service`.

   b. **Line 6:** We define the type of Service object we are using. Since the `web` component has to be accessible from outside of the cluster; this cannot be a Service object of the type `ClusterIP` and must be either of the type `NodePort` or `LoadBalancer`. We discussed the various types of Kubernetes services in the previous sections, so we will not go into further detail about this. In our sample, we are using a service of type `NodePort`.

   c. **Lines 7 and 8:** We define the filter criteria for the pods that this service will be a stable endpoint for. In this case, it is all the pods that have a label `app` with a value `web`.

   d. **Lines 9 to 11:** We specify that we want to expose port 3000 for access through the TCP protocol. Kubernetes will map container port 3000

automatically to a free host port in the range of 30,000 to 32,768. Which port Kubernetes effectively chooses can be determined using the **`kubectl get service`** or **`kubectl describe`** command for the service after it has been created.

2. Before you proceed, make sure that your cluster is clean using the **`kubectl get all`** command. Delete any leftover object except the default **`service/kubernetes`** service.

3. Now, let us deploy the **`web`** service using the **`Deployment`** specification from the previous section:

   ```
   kubectl create -f .kube/deployment-web.yaml
   ```

   Double-check that all is up and running with **`kubectl get all`**.

4. Use this command to create the Kubernetes service for **`web`** in the cluster:

   ```
   kubectl create -f .kube/service-web.yaml
   ```

5. Make sure the service has been created successfully by using the following command:

   ```
   kubectl get service
   ```

   The output should look similar to the following:

```
→  demo-app git:(main) ✗ kubectl get service
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
kubernetes    ClusterIP   10.96.0.1       <none>        443/TCP          5d23h
web-service   NodePort    10.108.23.73    <none>        3000:31675/TCP   3s
```

Figure 10.25: Output of `kubectl get service` showing the Kubernetes services

The first entry in the list is just the default service that must always be present, and the second item is the service we just created. Specifically, note the value in the PORT(S) column. In our case, it is **`3000:31675/TCP`**. This means that the container port 3000 has been mapped to host port **`31675`** and is open for TCP-type traffic. Note, in your case, the host port may be a different number! The number is automatically selected by Kubernetes and is dependent on your specific environment.

6. Use **`curl`** to access the Web application:

   ```
   curl localhost:31675
   ```

   and you should see something like the following:

   ```
   <html><head><title>Demo  Application</title></head><body><h1>Demo
   Web application!</h1></body></html>%
   ```

automatically to a free host port in the range of 30,000 to 32,768. Which port Kubernetes effectively chooses can be determined using the **`kubectl get service`** or **`kubectl describe`** command for the service after it has been created.

2. Before you proceed, make sure that your cluster is clean using the **`kubectl get all`** command. Delete any leftover object except the default **`service/kubernetes`** service.

3. Now, let us deploy the **`web`** service using the **`Deployment`** specification from the previous section:

   ```
   kubectl create -f .kube/deployment-web.yaml
   ```

   Double-check that all is up and running with **`kubectl get all`**.

4. Use this command to create the Kubernetes service for **`web`** in the cluster:

   ```
   kubectl create -f .kube/service-web.yaml
   ```

5. Make sure the service has been created successfully by using the following command:

   ```
   kubectl get service
   ```

   The output should look similar to the following:

```
→  demo-app git:(main) ✗ kubectl get service
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
kubernetes    ClusterIP   10.96.0.1       <none>        443/TCP          5d23h
web-service   NodePort    10.108.23.73    <none>        3000:31675/TCP   3s
```

Figure 10.25: Output of `kubectl get service` showing the Kubernetes services

The first entry in the list is just the default service that must always be present, and the second item is the service we just created. Specifically, note the value in the PORT(S) column. In our case, it is **`3000:31675/TCP`**. This means that the container port 3000 has been mapped to host port **`31675`** and is open for TCP-type traffic. Note, in your case, the host port may be a different number! The number is automatically selected by Kubernetes and is dependent on your specific environment.

6. Use **`curl`** to access the Web application:

   ```
   curl localhost:31675
   ```

   and you should see something like the following:

   ```
   <html><head><title>Demo  Application</title></head><body><h1>Demo
   Web application!</h1></body></html>%
   ```

This is proof that we can now indeed reach the **web** application service as the output is what our simple web application produces for the base URL.

Once again, make sure that you use the port number that the **`kubectl get service`** command shows you on your system.

7. When done playing with the deployment and the Kubernetes service, make sure to remove all objects from the cluster before you continue.

In the next section, we will learn more about context-based routing and how Kubernetes alleviates this task.

Context-based routing

Often, we want to configure context-based routing for our Kubernetes cluster. Kubernetes offers us various ways to do this. The preferred and most scalable way at this time is to use an **IngressController**. The following diagram tries to illustrate how this ingress controller works:

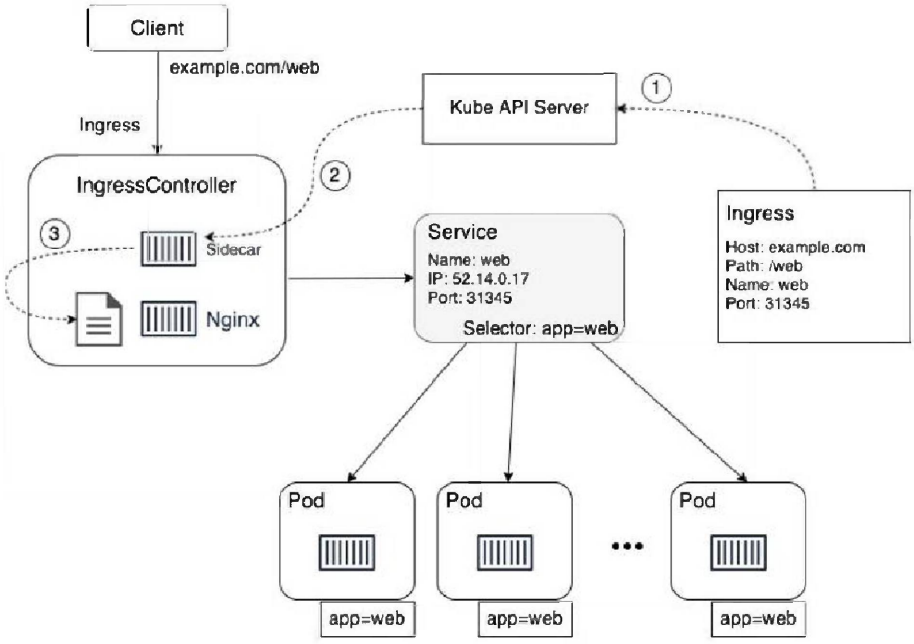


Figure 10.26: Using an ingress controller for context-based routing

This is proof that we can now indeed reach the **web** application service as the output is what our simple web application produces for the base URL.

Once again, make sure that you use the port number that the **`kubectl get service`** command shows you on your system.

7. When done playing with the deployment and the Kubernetes service, make sure to remove all objects from the cluster before you continue.

In the next section, we will learn more about context-based routing and how Kubernetes alleviates this task.

Context-based routing

Often, we want to configure context-based routing for our Kubernetes cluster. Kubernetes offers us various ways to do this. The preferred and most scalable way at this time is to use an **IngressController**. The following diagram tries to illustrate how this ingress controller works:

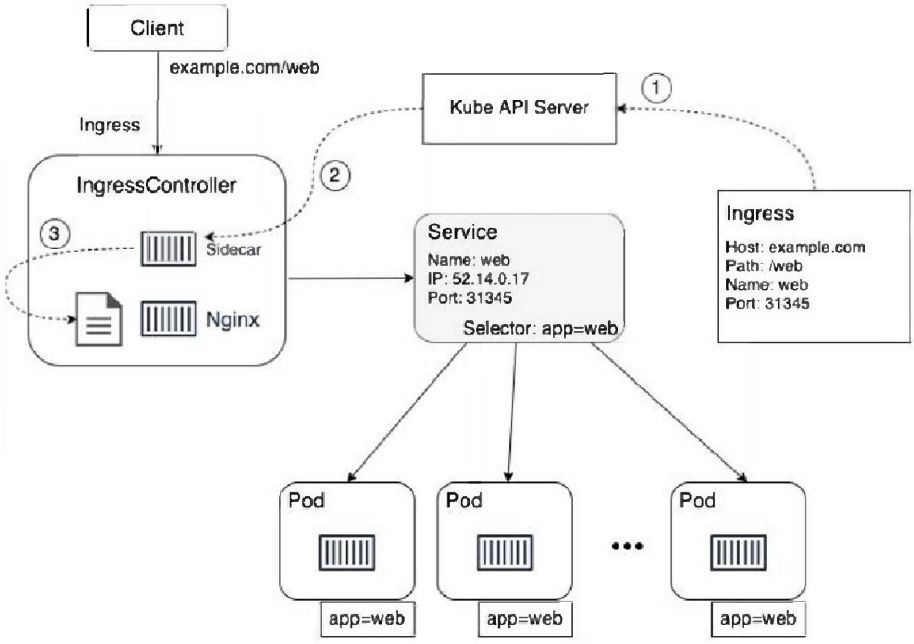


Figure 10.26: Using an ingress controller for context-based routing

In the preceding diagram, we can see how context-based (or Layer 7) routing works when using an `IngressController`, such as Nginx. Here, we have the deployment of an application service called `web`. All the pods of this application service have the following label: `app=web`. Then, we have a Kubernetes service called `web` that provides a stable endpoint to those pods. The service has a (virtual) IP of `52.14.0.17` and exposes port `31345`. That is, if a request comes to any node of the Kubernetes cluster for the name `web` and port `31345`, then it is forwarded to this service. The service then load-balances the request to one of the pods.

So far, so good, but how is an ingress request from a client to the `http[s]://example.com/web` URL routed to our `web` service? First, we have to define routing from a context-based request to a corresponding `<service name>/<port>` request. This is done through an Ingress object:

1. In the **Ingress** object, we define the **Host** and **Path** as the source and the (service) name and the port as the target.
2. When this `Ingress` object is created by the Kubernetes API server, then a process that runs as a sidecar inside the `IngressController` picks this change-up.
3. The sidecar modifies the configuration file of the Nginx reverse proxy.
4. By adding the new route, Nginx is then asked to reload its configuration, and thus, will be able to correctly route any incoming requests to `http[s]://example.com/web`.

In the next section, we are going to compare a few hosted Kubernetes offerings in the cloud.

Leveraging managed Kubernetes offerings in the cloud

Managing your own Kubernetes cluster on-premise or in the cloud is hard. Although the situation has quite improved over time, and many helpful tools are available for a cluster admin, it is still a fact that Kubernetes is a complicated beast. Furthermore, the service that a Kubernetes cluster provides is not unique to our or your business. It is a commodity. Companies of all sizes and of all sectors of the industry leverage Kubernetes to orchestrate their mission-critical applications. Thus, hardly any project or product is special enough that it can claim self-managing a Kubernetes cluster provides a competitive advantage.

That said, we recommend to all but the most demanding or exotic projects or applications out there to leverage one of the existing hosted Kubernetes offerings that exist in the cloud. With this, you will outsource the hard tasks of installing,

In the preceding diagram, we can see how context-based (or Layer 7) routing works when using an `IngressController`, such as Nginx. Here, we have the deployment of an application service called `web`. All the pods of this application service have the following label: `app=web`. Then, we have a Kubernetes service called `web` that provides a stable endpoint to those pods. The service has a (virtual) IP of `52.14.0.17` and exposes port `31345`. That is, if a request comes to any node of the Kubernetes cluster for the name `web` and port `31345`, then it is forwarded to this service. The service then load-balances the request to one of the pods.

So far, so good, but how is an ingress request from a client to the `http[s]://example.com/web` URL routed to our `web` service? First, we have to define routing from a context-based request to a corresponding `<service name>/<port>` request. This is done through an Ingress object:

1. In the **Ingress** object, we define the **Host** and **Path** as the source and the (service) name and the port as the target.
2. When this `Ingress` object is created by the Kubernetes API server, then a process that runs as a sidecar inside the `IngressController` picks this change-up.
3. The sidecar modifies the configuration file of the Nginx reverse proxy.
4. By adding the new route, Nginx is then asked to reload its configuration, and thus, will be able to correctly route any incoming requests to `http[s]://example.com/web`.

In the next section, we are going to compare a few hosted Kubernetes offerings in the cloud.

Leveraging managed Kubernetes offerings in the cloud

Managing your own Kubernetes cluster on-premise or in the cloud is hard. Although the situation has quite improved over time, and many helpful tools are available for a cluster admin, it is still a fact that Kubernetes is a complicated beast. Furthermore, the service that a Kubernetes cluster provides is not unique to our or your business. It is a commodity. Companies of all sizes and of all sectors of the industry leverage Kubernetes to orchestrate their mission-critical applications. Thus, hardly any project or product is special enough that it can claim self-managing a Kubernetes cluster provides a competitive advantage.

That said, we recommend to all but the most demanding or exotic projects or applications out there to leverage one of the existing hosted Kubernetes offerings that exist in the cloud. With this, you will outsource the hard tasks of installing,

running, upgrading, and monitoring the cluster to specialists that do nothing else than concentrate on these kinds of tasks. As a consequence, you will be able to concentrate on what really provides business value or competitive advantage to your application. Kubernetes, for you, will just be a bunch of worker nodes to which you can deploy your containers with your application services. You can easily scale up and down the number of worker nodes on a need basis. Upgrading the cluster to the newest version of Kubernetes is as simple as clicking a button. Monitoring your application will be easy as these hosted Kubernetes offerings are usually deeply integrated with the other services, logging, and monitoring the cloud provider offers.

Let us start by peeking into the hosted Kubernetes offering of AWS, the biggest cloud provider still out there.

Amazon AWS Elastic Kubernetes Service

In this section, we want to create a fully managed Kubernetes cluster on AWS EKS using Fargate. The process of creating a new cluster is well described in the AWS documentation, and we will refer to the respective pages to not duplicate too much information. That said, let us start with the following:

Let us first have a few prerequisites out of our way:

1. Make sure you have access to an AWS account. If not, you can get a free one-year trial account here: **https://aws.amazon.com/free**
2. Login to your AWS account.
3. Create a new access key and access key secret pair for your account that you will use to configure your **aws** CLI, such as that you can access your account from the command line.
    a. Locate your profile on the top left of the screen, and in the dropdown, select **Security credentials.**
    b. Select Access keys (access key ID and secret access key) and then click **Create New Access Key:**

running, upgrading, and monitoring the cluster to specialists that do nothing else than concentrate on these kinds of tasks. As a consequence, you will be able to concentrate on what really provides business value or competitive advantage to your application. Kubernetes, for you, will just be a bunch of worker nodes to which you can deploy your containers with your application services. You can easily scale up and down the number of worker nodes on a need basis. Upgrading the cluster to the newest version of Kubernetes is as simple as clicking a button. Monitoring your application will be easy as these hosted Kubernetes offerings are usually deeply integrated with the other services, logging, and monitoring the cloud provider offers.

Let us start by peeking into the hosted Kubernetes offering of AWS, the biggest cloud provider still out there.

Amazon AWS Elastic Kubernetes Service

In this section, we want to create a fully managed Kubernetes cluster on AWS EKS using Fargate. The process of creating a new cluster is well described in the AWS documentation, and we will refer to the respective pages to not duplicate too much information. That said, let us start with the following:

Let us first have a few prerequisites out of our way:

1. Make sure you have access to an AWS account. If not, you can get a free one-year trial account here: **https://aws.amazon.com/free**
2. Login to your AWS account.
3. Create a new access key and access key secret pair for your account that you will use to configure your **aws** CLI, such as that you can access your account from the command line.
    a. Locate your profile on the top left of the screen, and in the dropdown, select **Security credentials.**
    b. Select Access keys (access key ID and secret access key) and then click **Create New Access Key:**



Figure 10.27: Create a new access key and secret pair

Note down the access key ID and secret pair in a safe place.

4. Open a new terminal.

5. Make sure you have the `aws` CLI installed:

   ```
   aws --version
   ```

6. Configure your `aws` CLI. For this, you need your AWS Access Key ID, AWS Secret Access Key that you created in preceding Step 3, as well as your default region.

   Then use the following command:

   ```
   aws configure
   ```

   Enter the appropriate values when asked. For the default output format, select `json`:

```
→  demo-app git:(main) ✗ aws configure
AWS Access Key ID [None]: AKIA3S
AWS Secret Access Key [None]: DSjeH7WGv
Default region name [None]: eu-central-1
Default output format [None]: JSON
```

Figure 10.28: Configuring the aws CLI



Figure 10.27: Create a new access key and secret pair

Note down the access key ID and secret pair in a safe place.

4. Open a new terminal.
5. Make sure you have the `aws` CLI installed:

   ```
   aws --version
   ```

6. Configure your `aws` CLI. For this, you need your AWS Access Key ID, AWS Secret Access Key that you created in preceding Step 3, as well as your default region.

   Then use the following command:

   ```
   aws configure
   ```

   Enter the appropriate values when asked. For the default output format, select `json`:

```
→  demo-app git:(main) ✗ aws configure
AWS Access Key ID [None]: AKIA3S
AWS Secret Access Key [None]: DSjeH7WGv
Default region name [None]: eu-central-1
Default output format [None]: JSON
```

Figure 10.28: Configuring the aws CLI

7. Try accessing your account with a command like the following:

   ```
   aws s3 ls
   ```

   which should list all the S3 buckets defined for your account. Your list may be empty. The important thing here is that the command succeeds.

8. Finally, make sure you have `kubectl` installed:

   ```
   kubectl --version
   ```

Now, we are ready to create the AWS EKS cluster:

1. Follow the step-by-step instructions at the following link to create your cluster:

   https://docs.aws.amazon.com/eks/latest/userguide/getting-started-console.html (Getting started with Amazon EKS – AWS Management Console and AWS CLI)

   a. When creating the VPC with public and private subnets using the aws CLI and the proposed template `amazon-eks-vpc-private-subnets.yaml` make sure you select the region nearest to you, such as `eu-central-1`.

   b. When selecting a name for your cluster, use `docker-for-job-seekers-cluster`.

   c. When asked which type of nodes to create select **Fargate - Linux** and continue the setup accordingly.

2. Once the cluster is ready, configure `kubectl` to access your new cluster on AWS:

   ```
   aws eks update-kubeconfig --name docker-for-job-seekers-cluster
   ```

   The response should be similar to the following:

   ```
   Added new context arn:aws:eks:eu-central-...:cluster/docker-for-job-seekers-cluster to /Users/<user-name>/.kube/config
   ```

3. Double-check that `kubectl` is using the correct context—the one that was just created for the cluster on AWS and added to your `.kube/config` file:

   ```
   kubectl config current-context
   ```

   The answer should look similar to the following:

   ```
   arn:aws:eks:eu-central-...:cluster/docker-for-job-seekers-cluster
   ```

4. Use `kubectl` to list all resources on your cluster:

   ```
   kubectl get all
   ```

7. Try accessing your account with a command like the following:

   ```
   aws s3 ls
   ```

   which should list all the S3 buckets defined for your account. Your list may be empty. The important thing here is that the command succeeds.

8. Finally, make sure you have `kubectl` installed:

   ```
   kubectl --version
   ```

Now, we are ready to create the AWS EKS cluster:

1. Follow the step-by-step instructions at the following link to create your cluster:

   https://docs.aws.amazon.com/eks/latest/userguide/getting-started-console.html (Getting started with Amazon EKS – AWS Management Console and AWS CLI)

   a. When creating the VPC with public and private subnets using the aws CLI and the proposed template `amazon-eks-vpc-private-subnets.yaml` make sure you select the region nearest to you, such as `eu-central-1`.

   b. When selecting a name for your cluster, use `docker-for-job-seekers-cluster`.

   c. When asked which type of nodes to create select **Fargate - Linux** and continue the setup accordingly.

2. Once the cluster is ready, configure `kubectl` to access your new cluster on AWS:

   ```
   aws eks update-kubeconfig --name docker-for-job-seekers-cluster
   ```

   The response should be similar to the following:

   ```
   Added new context arn:aws:eks:eu-central-...:cluster/docker-for-job-seekers-cluster to /Users/<user-name>/.kube/config
   ```

3. Double-check that `kubectl` is using the correct context—the one that was just created for the cluster on AWS and added to your `.kube/config` file:

   ```
   kubectl config current-context
   ```

   The answer should look similar to the following:

   ```
   arn:aws:eks:eu-central-...:cluster/docker-for-job-seekers-cluster
   ```

4. Use `kubectl` to list all resources on your cluster:

   ```
   kubectl get all
   ```

The answer at this time should be like the following:

```
NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)
AGE
service/kuber...     ClusterIP   10.100.0.1   <none>        443/TCP
21m
```

5. Navigate to the **ch10** folder of this chapter, and create a subfolder **aws-eks**:

```
cd ~/src/ch10 && mkdir aws-eks && cd aws-eks
```

6. In this subfolder, create a file **deploy-nginx.yaml** with the following content:

```
! deploy-nginx.yaml ×
ch10 > aws-eks > ! deploy-nginx.yaml > apiVersion
1   apiVersion: apps/v1
2   kind: Deployment
3   metadata:
4     name: web-deployment
5     labels:
6       app: web
7   spec:
8     replicas: 3
9     selector:
10      matchLabels:
11        app: web
12    template:
13      metadata:
14        labels:
15          app: web
16      spec:
17        containers:
18        - name: web
19          image: nginx:latest
20          ports:
21          - containerPort: 80
```

Figure 10.29: Deployment specification for Nginx on AWS EKS

7. Use **kubectl** to deploy our deployment to the cluster:

```
kubectl apply -f deploy-nginx.yaml
```

8. Observe the creation of the pods with the following:

```
kubectl get pods -w
```

The answer at this time should be like the following:

```
NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)
AGE
service/kuber...     ClusterIP   10.100.0.1   <none>        443/TCP
21m
```

5. Navigate to the **ch10** folder of this chapter, and create a subfolder **aws-eks**:

```
cd ~/src/ch10 && mkdir aws-eks && cd aws-eks
```

6. In this subfolder, create a file **deploy-nginx.yaml** with the following content:



Figure 10.29: Deployment specification for Nginx on AWS EKS

7. Use **kubectl** to deploy our deployment to the cluster:

```
kubectl apply -f deploy-nginx.yaml
```

8. Observe the creation of the pods with the following:

```
kubectl get pods -w
```

And wait until they are ready:

```
→  .kube git:(main) ✗ kubectl get pods -w
NAME                                READY   STATUS    RESTARTS   AGE
web-deployment-656685fd4f-2jlpk     0/1     Pending   0          32s
web-deployment-656685fd4f-kb4zw     0/1     Pending   0          32s
web-deployment-656685fd4f-n8zjj     0/1     Pending   0          32s
```

Figure 10.30: Listing the pods of the deployment to AWS

Wait until their value in the READY column is 1/1.

9. In the AWS console, navigate to your cluster, and in the **Overview** tab, observe that three Fargate nodes were created.
    a. Drill down to a node to see the pod that has been deployed to it.
    b. Drill further down to the pod and observe the list of events shown in its details view.

Congratulations, you have created a fully hosted Kubernetes cluster on AWS and created a first Deployment on it using **kubectl**. As you will, this is quite an achievement. It turns out that of all the discussed cloud providers, AWS requires by far the most steps to get a Kubernetes cluster up and running.

Before you leave, and to avoid unexpected cost, make sure you clean up all the resources that you have created during this exercise. For this, follow the steps in the section Delete your cluster and nodes in the AWS documentation:

https://docs.aws.amazon.com/eks/latest/userguide/getting-started-console.html

Now that we have a rough understanding of what AWS EKS offers let us have a look at what the second-biggest cloud providers have in its portfolio.

Microsoft Azure Kubernetes Service

Microsoft is the second biggest cloud provider right after AWS with its Microsoft Azure. Let us see what Azure is offering regarding hosted Kubernetes. One would expect a first-class service given the fact that Microsoft has been known for always having a strong relationship with major businesses out there. Let us have a look.

1. Make sure you have access to an Azure account. If not, you can get a free trial account here: **https://azure.microsoft.com/en-us/free/**
2. Login to your Azure account.
3. Locate and navigate to the **Azure Kubernetes Service (AKS)**.
4. Click **Create** and select **Create a Kubernetes cluster**.

And wait until they are ready:

```
→  .kube git:(main) ✗ kubectl get pods -w
NAME                                READY   STATUS    RESTARTS   AGE
web-deployment-656685fd4f-2jlpk     0/1     Pending   0          32s
web-deployment-656685fd4f-kb4zw     0/1     Pending   0          32s
web-deployment-656685fd4f-n8zjj     0/1     Pending   0          32s
```

Figure 10.30: Listing the pods of the deployment to AWS

Wait until their value in the READY column is 1/1.

9. In the AWS console, navigate to your cluster, and in the **Overview** tab, observe that three Fargate nodes were created.
    a. Drill down to a node to see the pod that has been deployed to it.
    b. Drill further down to the pod and observe the list of events shown in its details view.

Congratulations, you have created a fully hosted Kubernetes cluster on AWS and created a first Deployment on it using **kubectl**. As you will, this is quite an achievement. It turns out that of all the discussed cloud providers, AWS requires by far the most steps to get a Kubernetes cluster up and running.

Before you leave, and to avoid unexpected cost, make sure you clean up all the resources that you have created during this exercise. For this, follow the steps in the section Delete your cluster and nodes in the AWS documentation:

https://docs.aws.amazon.com/eks/latest/userguide/getting-started-console.html

Now that we have a rough understanding of what AWS EKS offers let us have a look at what the second-biggest cloud providers have in its portfolio.

Microsoft Azure Kubernetes Service

Microsoft is the second biggest cloud provider right after AWS with its Microsoft Azure. Let us see what Azure is offering regarding hosted Kubernetes. One would expect a first-class service given the fact that Microsoft has been known for always having a strong relationship with major businesses out there. Let us have a look.

1. Make sure you have access to an Azure account. If not, you can get a free trial account here: **https://azure.microsoft.com/en-us/free/**
2. Login to your Azure account.
3. Locate and navigate to the **Azure Kubernetes Service (AKS)**.
4. Click **Create** and select **Create a Kubernetes cluster**.

5. On the Create Kubernetes cluster view.
    a. Enter / select your subscription
    b. Leave the default suggested by Azure for the **Resource** group
    c. Select **Dev/Test** as the **Cluster preset configuration** to save cost
    d. Provide a **Kubernetes cluster name**. In our case, we select **`docker-for-job-seekers-cluster`**
    e. Select the **Region** that is closest to you
    f. Leave the **Availability zones** and **Kubernetes version** with their default values.
    g. Set the **Scale method** to **`Autoscale`** and leave the default in the **Node count range.**
    h. Review your settings and click **Review + create.**
    i. If all looks good, click **Create** to create the cluster. This may take a moment. It is the right time to grab a cup of tea or coffee. When done, you should see a screen similar to the following:

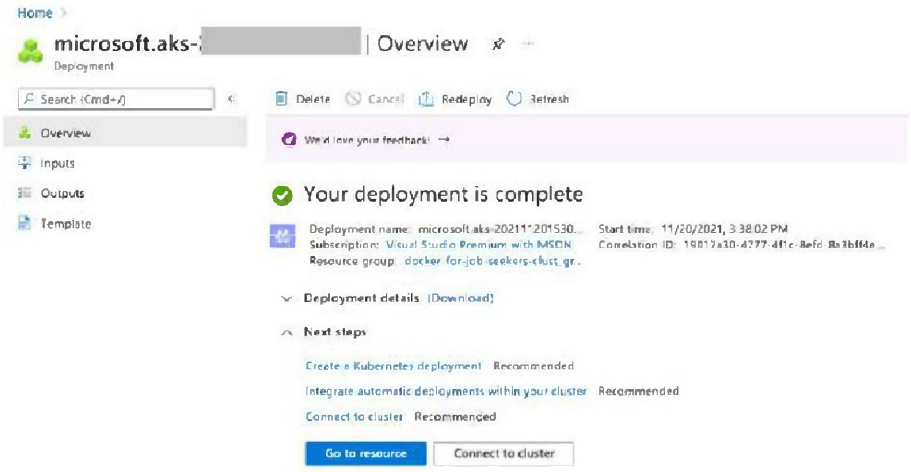


Figure 10.31: Azure Kubernetes cluster is ready

6. Click **Go to resource** to navigate to the detail screen of the cluster you just created. Analyze what you see and make sure it corresponds to your expectations.

5. On the Create Kubernetes cluster view.
    a. Enter / select your subscription
    b. Leave the default suggested by Azure for the **Resource** group
    c. Select **Dev/Test** as the **Cluster preset configuration** to save cost
    d. Provide a **Kubernetes cluster name**. In our case, we select **`docker-for-job-seekers-cluster`**
    e. Select the **Region** that is closest to you
    f. Leave the **Availability zones** and **Kubernetes version** with their default values.
    g. Set the **Scale method** to **`Autoscale`** and leave the default in the **Node count range.**
    h. Review your settings and click **Review + create.**
    i. If all looks good, click **Create** to create the cluster. This may take a moment. It is the right time to grab a cup of tea or coffee. When done, you should see a screen similar to the following:

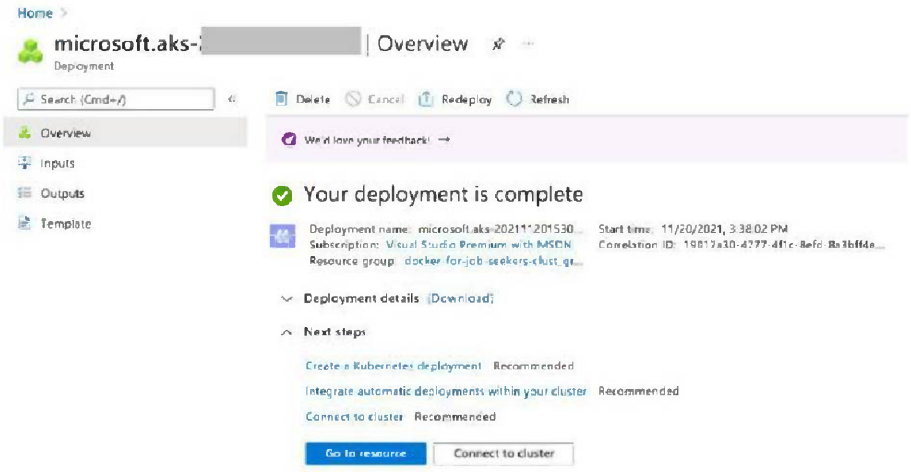


Figure 10.31: Azure Kubernetes cluster is ready

6. Click **Go to resource** to navigate to the detail screen of the cluster you just created. Analyze what you see and make sure it corresponds to your expectations.

7. Select the **Get started** tab on the details view and click **Create a deployment**.
8. On the next screen, called **Add with YAML**, click the link **Deploy a quickstart application to get up and running**.
9. Have a look at the YAML that is used for the quick start application and notice that it contains the specification for a **Kubernetes Deployment** and a **Kubernetes Service**, each for a backend and a frontend component labeled `azure-vote-back` and `azure-vote-front` respectively.
10. When done reviewing the YAML click **Deploy** to have this quick-start application deployed to your cluster. This will take a moment. Once the deployment is complete, you should see something similar to the following:

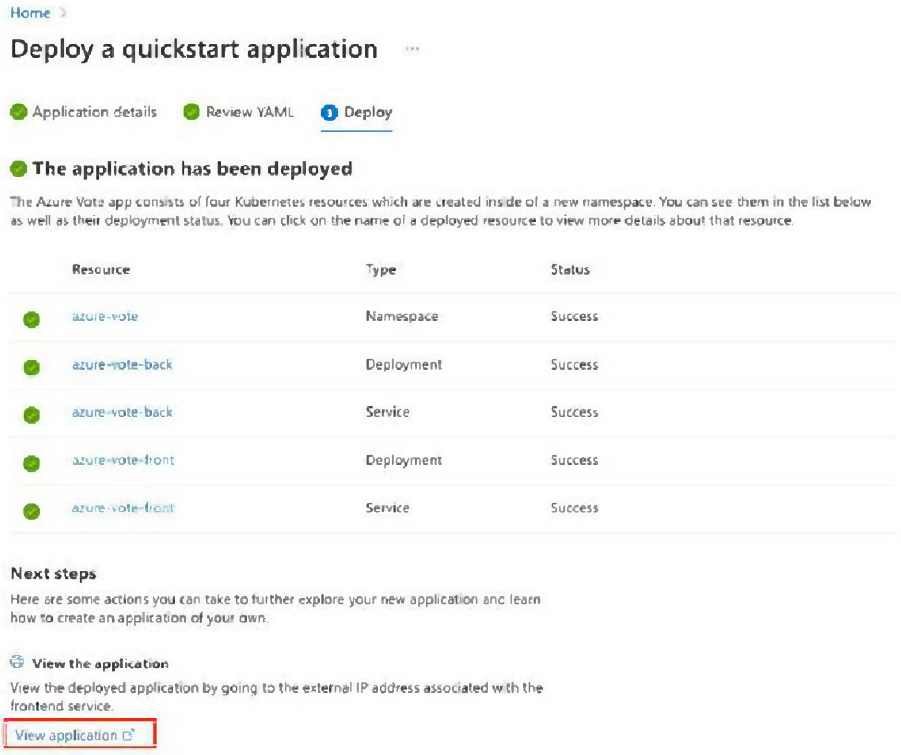


Figure 10.32: Azure deployment succeeded

7. Select the **Get started** tab on the details view and click **Create a deployment**.
8. On the next screen, called **Add with YAML**, click the link **Deploy a quickstart application to get up and running**.
9. Have a look at the YAML that is used for the quick start application and notice that it contains the specification for a **Kubernetes Deployment** and a **Kubernetes Service**, each for a backend and a frontend component labeled `azure-vote-back` and `azure-vote-front` respectively.
10. When done reviewing the YAML click **Deploy** to have this quick-start application deployed to your cluster. This will take a moment. Once the deployment is complete, you should see something similar to the following:

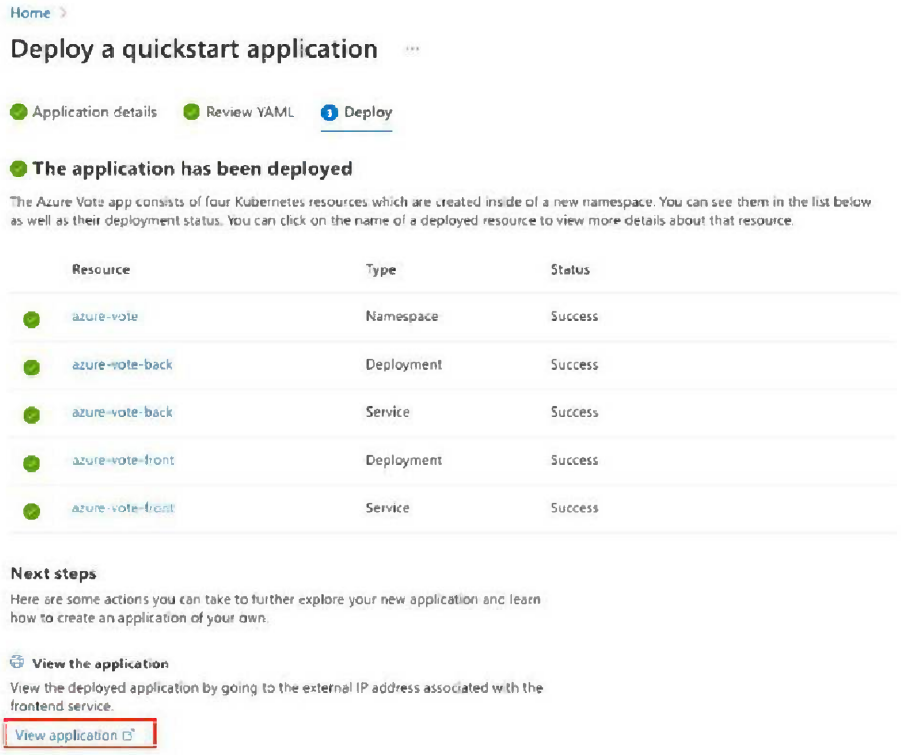


Figure 10.32: Azure deployment succeeded

11. Click on the **View application** link shown previously. A new browser window will open, and the UI of the quick start voting application is shown:

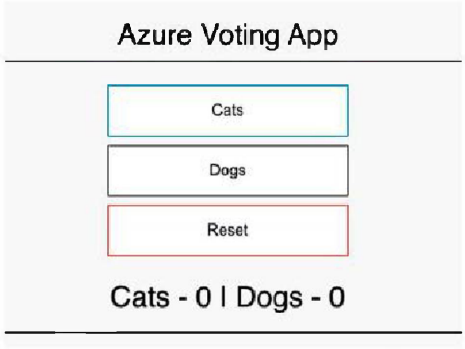


Figure 10.33: Azure quick start voting application

Congratulations, you have successfully deployed your first application on a hosted Kubernetes cluster on Microsoft Azure.

12. Go back to your Azure portal and navigate to your Kubernetes cluster. On the left side of the cluster details view, select **Workloads** to investigate what is running on your cluster. You should see the following:

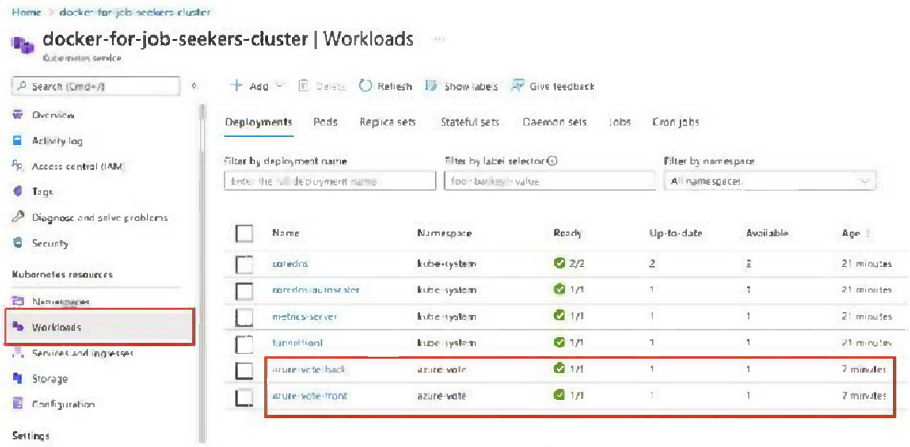


Figure 10.34: Workload on the Azure Kubernetes cluster

Please notice the two services, `azure-vote-back` and `azure-vote-front`.

13. Select either of the two services and explore their details, such as the underlying YAML, the logs, the events, and more.

11. Click on the **View application** link shown previously. A new browser window will open, and the UI of the quick start voting application is shown:

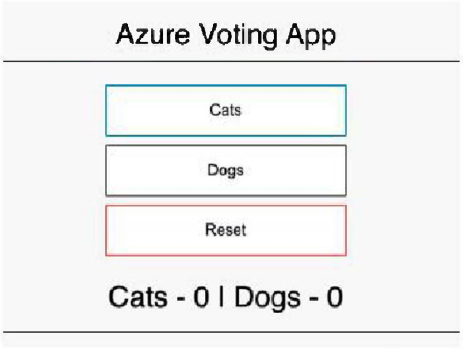


Figure 10.33: Azure quick start voting application

Congratulations, you have successfully deployed your first application on a hosted Kubernetes cluster on Microsoft Azure.

12. Go back to your Azure portal and navigate to your Kubernetes cluster. On the left side of the cluster details view, select **Workloads** to investigate what is running on your cluster. You should see the following:

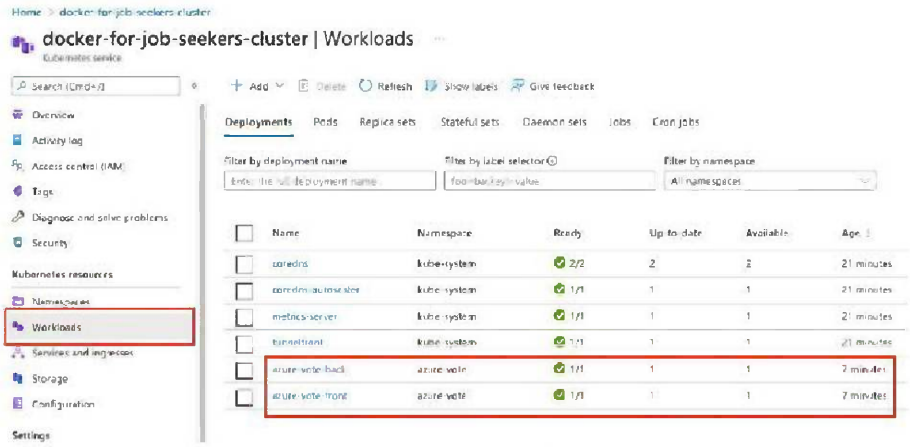


Figure 10.34: Workload on the Azure Kubernetes cluster

Please notice the two services, `azure-vote-back` and `azure-vote-front`.

13. Select either of the two services and explore their details, such as the underlying YAML, the logs, the events, and more.

14. Before you leave, and to avoid unexpected cost, make sure you clean up all the resources that you have created during this exercise. To do this, you may need to:
    a. Navigate to the details view of your cluster and then select **Delete.**
    b. Once the cluster has been deleted, navigate to Home. On the dashboard, you will find the Resource group created for the cluster listed under **Recent resources.** Select this group and then delete it.

OK, we have seen what the Azure AKS service can offer us in regard to hosted Kubernetes. It is now time to discuss the offering of the third major player in the area, Google.

Google GCP

As we have discussed earlier in this book, Google was the company that invented Kubernetes. Kubernetes is based on Google's internal Borg engine and has been open-sourced a few years ago. Kubernetes is now under the control of the cloud native computing foundation CNCF.

Let us see what the mother of Kubernetes has to offer us on their **Google Cloud Platform** (**GCP**) in regard to hosted Kubernetes.

1. Make sure you have access to a Google cloud account. If not, you can get a free one-year trial account here: **https://cloud.google.com/free**
2. Login to your Google cloud account.
3. Locate the, and navigate to the dashboard, and from there, select **Kubernetes Engine.**
4. Select the **Clusters** tab and click on the **CREATE PROJECT** link.
5. Give the project a meaningful name. In our case, we call it **docker for Job Seekers** and click **CREATE.** You may have to wait a few moments for the project to initialize.
6. Next, click on **ENABLE** when asked to enable the Kubernetes Engine API. Once again, it may take a few moments until this step completes. Once completed, you should see the following:

14. Before you leave, and to avoid unexpected cost, make sure you clean up all the resources that you have created during this exercise. To do this, you may need to:
    a. Navigate to the details view of your cluster and then select **Delete.**
    b. Once the cluster has been deleted, navigate to Home. On the dashboard, you will find the Resource group created for the cluster listed under **Recent resources.** Select this group and then delete it.

OK, we have seen what the Azure AKS service can offer us in regard to hosted Kubernetes. It is now time to discuss the offering of the third major player in the area, Google.

Google GCP

As we have discussed earlier in this book, Google was the company that invented Kubernetes. Kubernetes is based on Google's internal Borg engine and has been open-sourced a few years ago. Kubernetes is now under the control of the cloud native computing foundation CNCF.

Let us see what the mother of Kubernetes has to offer us on their **Google Cloud Platform** (**GCP**) in regard to hosted Kubernetes.

1. Make sure you have access to a Google cloud account. If not, you can get a free one-year trial account here: **https://cloud.google.com/free**
2. Login to your Google cloud account.
3. Locate the, and navigate to the dashboard, and from there, select **Kubernetes Engine.**
4. Select the **Clusters** tab and click on the **CREATE PROJECT** link.
5. Give the project a meaningful name. In our case, we call it **docker for Job Seekers** and click **CREATE.** You may have to wait a few moments for the project to initialize.
6. Next, click on **ENABLE** when asked to enable the Kubernetes Engine API. Once again, it may take a few moments until this step completes. Once completed, you should see the following:

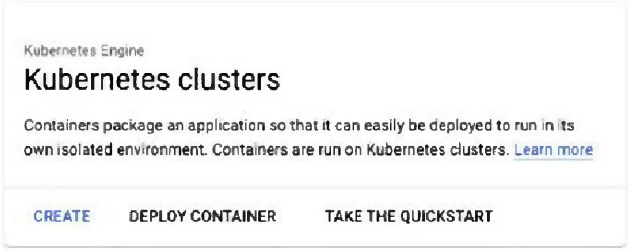


Figure 10.35: Google is ready to create a Kubernetes cluster

7. Click **CREATE** to create a new Kubernetes cluster. In the next dialog box, select GKE Autopilot and click **CONFIGURE**.

8. Enter a name for the cluster. In our case, we chose **`docker-for-job-seekers-cluster`**. Select the region that is closest to you. Leave the cluster as **public**. And then click **CREATE**. This operation will again take a few moments. It is a good time to grab a cup of tea or coffee. If all goes well, then the result should look similar to the following:

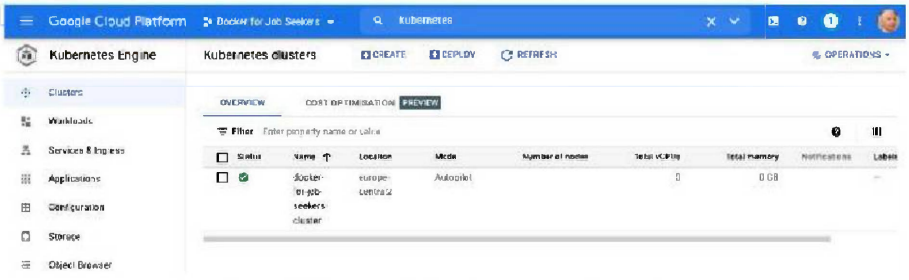


Figure 10.36: Hosted Kubernetes cluster on Google cloud

9. Click **DEPLOY** to deploy the first Docker image to your new cluster. In the dialog box, leave all the default settings. This will deploy an Nginx container in three instances. Note that initially, the deployment will seem to fail. Do

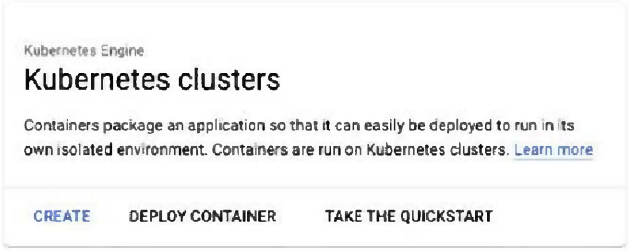


Figure 10.35: Google is ready to create a Kubernetes cluster

7. Click **CREATE** to create a new Kubernetes cluster. In the next dialog box, select GKE Autopilot and click **CONFIGURE**.

8. Enter a name for the cluster. In our case, we chose **`docker-for-job-seekers-cluster`**. Select the region that is closest to you. Leave the cluster as **public**. And then click **CREATE**. This operation will again take a few moments. It is a good time to grab a cup of tea or coffee. If all goes well, then the result should look similar to the following:

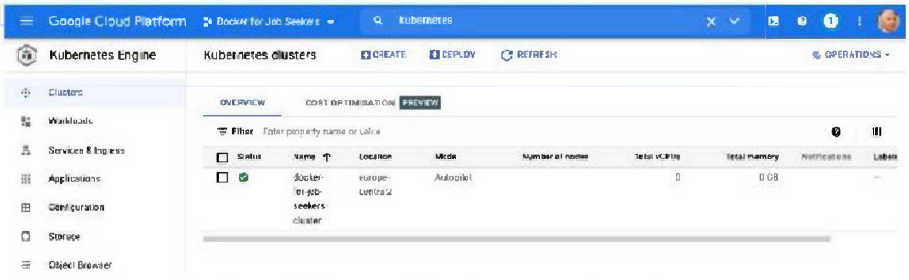


Figure 10.36: Hosted Kubernetes cluster on Google cloud

9. Click **DEPLOY** to deploy the first Docker image to your new cluster. In the dialog box, leave all the default settings. This will deploy an Nginx container in three instances. Note that initially, the deployment will seem to fail. Do

not worry, this is because your cluster does not have any nodes allocated yet. But the cluster will auto-scale, and the deployment will eventually succeed.



Figure 10.37: First workload deployed to cluster

10. Select this workload and on the details, click select **EXPOSE** to make Nginx reachable from the internet. In the port mapping on the next screen, select **8080** as **Target port**. Leave everything else as default. Optionally analyze the YAML that GCP will use to expose Nginx by clicking on **VIEW YAML**. You will notice that it is a specification for a Kubernetes service of type **LoadBalancer**. When done, click **EXPOSE**.

11. Once the service has been deployed, you should find yourself forwarded to the details view of the service. There you will find an item **External endpoints**:

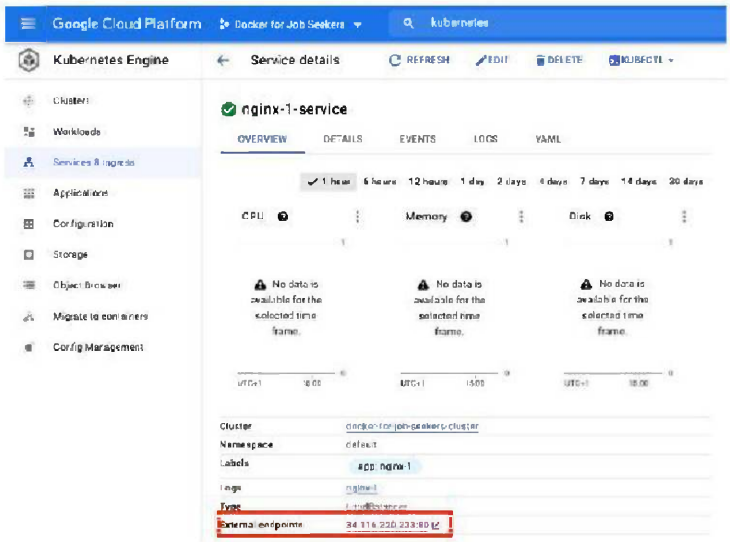


Figure 10.38: Service details on the GCP Kubernetes cluster

not worry, this is because your cluster does not have any nodes allocated yet. But the cluster will auto-scale, and the deployment will eventually succeed.

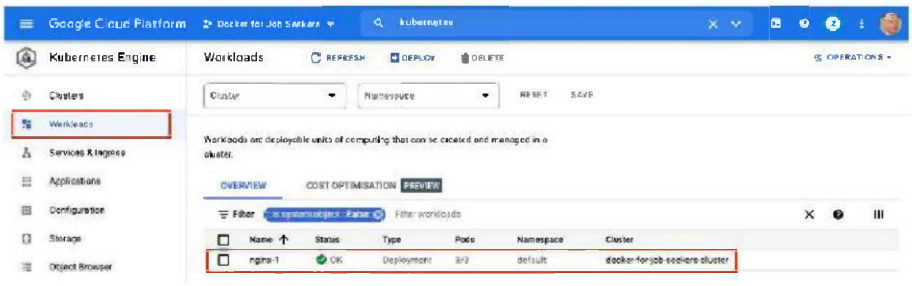


Figure 10.37: First workload deployed to cluster

10. Select this workload and on the details, click select **EXPOSE** to make Nginx reachable from the internet. In the port mapping on the next screen, select **8080** as **Target port**. Leave everything else as default. Optionally analyze the YAML that GCP will use to expose Nginx by clicking on **VIEW YAML**. You will notice that it is a specification for a Kubernetes service of type **LoadBalancer**. When done, click **EXPOSE**.

11. Once the service has been deployed, you should find yourself forwarded to the details view of the service. There you will find an item **External endpoints**:

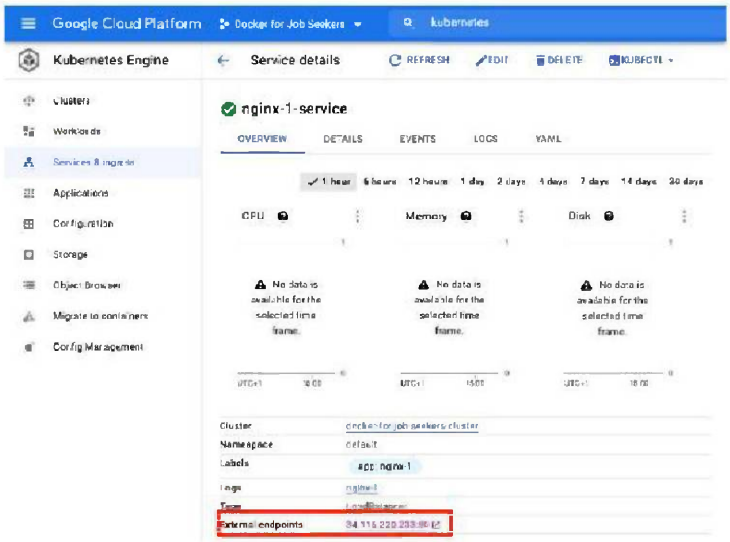


Figure 10.38: Service details on the GCP Kubernetes cluster

Click on the link, and you should see the welcome screen of Nginx. Congratulations, you have just successfully created your first workload on a hosted Kubernetes cluster on GCP.

12. Take your time to explore the workload you and the service you just defined. Maybe try to scale up or down your workload. Also, explore the (container) logs of the workload.

13. Before you leave, and to avoid unexpected cost, make sure you clean up all the resources that you have created during this exercise. To do this:

    a) Go back to the **Clusters** tab of the Kubernetes Engine and select your cluster. Then click **DELETE.**

    b) Navigate to **Home | Dashboard** and navigate to the project settings of your project. Select **SHUT DOWN**.

By now, you should have a good understanding of what the three major cloud providers, Amazon AWS, Microsoft Azure, and Google, can provide you with their hosted Kubernetes service and how easy it is to get started with each of those services.

There are many more offerings out there that we have not mentioned in this book due to space limitations. Please inform yourself before you decide on which offering to accept.

But even if you start with provider A and then it turns out that provider B would have an offering that better suits your needs, nothing (or very little) is lost. Kubernetes is Kubernetes, whether or not Amazon, Microsoft, Google, or your other favorite company runs and manages it. You should be able to migrate an application fairly easily from one cloud to the other.

Discussing containers in the context of serverless

You may be hearing the statement that serverless offerings such as AWS lambdas or Azure functions, serverless cloud databases, serverless and fully hosted Apache Kafka offerings, and so on will make containers obsolete. The reasoning is that one "only" needs to write a few scripts or functions in one of the supported languages, such as Node JS, and everything else is taken care of.

From the perspective of the developer, it may well be that they do not see containers in action directly anymore. But trust me, containers are still there. The difference is that somebody else builds and runs the containers for us. The infrastructure necessary to run serverless functions in the cloud often runs in containers and is

Click on the link, and you should see the welcome screen of Nginx. Congratulations, you have just successfully created your first workload on a hosted Kubernetes cluster on GCP.

12. Take your time to explore the workload you and the service you just defined. Maybe try to scale up or down your workload. Also, explore the (container) logs of the workload.

13. Before you leave, and to avoid unexpected cost, make sure you clean up all the resources that you have created during this exercise. To do this:

    a) Go back to the **Clusters** tab of the Kubernetes Engine and select your cluster. Then click **DELETE.**

    b) Navigate to **Home | Dashboard** and navigate to the project settings of your project. Select **SHUT DOWN**.

By now, you should have a good understanding of what the three major cloud providers, Amazon AWS, Microsoft Azure, and Google, can provide you with their hosted Kubernetes service and how easy it is to get started with each of those services.

There are many more offerings out there that we have not mentioned in this book due to space limitations. Please inform yourself before you decide on which offering to accept.

But even if you start with provider A and then it turns out that provider B would have an offering that better suits your needs, nothing (or very little) is lost. Kubernetes is Kubernetes, whether or not Amazon, Microsoft, Google, or your other favorite company runs and manages it. You should be able to migrate an application fairly easily from one cloud to the other.

Discussing containers in the context of serverless

You may be hearing the statement that serverless offerings such as AWS lambdas or Azure functions, serverless cloud databases, serverless and fully hosted Apache Kafka offerings, and so on will make containers obsolete. The reasoning is that one "only" needs to write a few scripts or functions in one of the supported languages, such as Node JS, and everything else is taken care of.

From the perspective of the developer, it may well be that they do not see containers in action directly anymore. But trust me, containers are still there. The difference is that somebody else builds and runs the containers for us. The infrastructure necessary to run serverless functions in the cloud often runs in containers and is

deployed on Kubernetes. Whenever one of your serverless functions is triggered, say by an incoming HTTP request, then the (fully managed) infrastructure may just spin up a new container instance inside a pod on Kubernetes and run your function inside this container. For you, all this does not really matter, I understand, since you are only interested in the fact that the function runs, that it runs fast, and has a low latency to start. But in the bigger picture, containers continue to play a major role in enabling this whole serverless approach.

It is, in a way, quite similar to what we see in regard to **virtual machines (VMs)**. Hardly anyone that runs software in the cloud, still works directly with VMs. We all use fully managed services such as cloud databases, hosted Kubernetes, and more. But it is VMs that are to this day, the building blocks that make the cloud possible in the first place.

Summarizing, containers are still and will be, in the foreseeable future, important building blocks that enable self-hosted or cloud-based applications.

Conclusion

In this chapter, we have learned what an orchestrator is and what its responsibilities are. We then dove deep into the innards of the currently most popular container orchestrator Kubernetes. We started with an overview of the architecture of Kubernetes and then discussed the most important parts of Kubernetes, starting with the pods, which are the atomic pieces of deployment and working ourselves through until the Kubernetes services, which route private and public access to the containerized applications we run in Kubernetes.

We then got a brief overview of the most popular hosted Kubernetes offerings in the cloud and ended the chapter by discussing the place of containers and container orchestrators such as Kubernetes in the context of the current serverless hype.

Questions

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

1. Explain in a few short sentences what the role of a Kubernetes master is.
2. List the elements that need to be present on each Kubernetes (worker) node.
3. We cannot run individual containers in a Kubernetes cluster.
   a. Yes
   b. No
4. Explain the reason why the containers in a pod can use localhost to communicate with each other.

deployed on Kubernetes. Whenever one of your serverless functions is triggered, say by an incoming HTTP request, then the (fully managed) infrastructure may just spin up a new container instance inside a pod on Kubernetes and run your function inside this container. For you, all this does not really matter, I understand, since you are only interested in the fact that the function runs, that it runs fast, and has a low latency to start. But in the bigger picture, containers continue to play a major role in enabling this whole serverless approach.

It is, in a way, quite similar to what we see in regard to **virtual machines (VMs)**. Hardly anyone that runs software in the cloud, still works directly with VMs. We all use fully managed services such as cloud databases, hosted Kubernetes, and more. But it is VMs that are to this day, the building blocks that make the cloud possible in the first place.

Summarizing, containers are still and will be, in the foreseeable future, important building blocks that enable self-hosted or cloud-based applications.

Conclusion

In this chapter, we have learned what an orchestrator is and what its responsibilities are. We then dove deep into the innards of the currently most popular container orchestrator Kubernetes. We started with an overview of the architecture of Kubernetes and then discussed the most important parts of Kubernetes, starting with the pods, which are the atomic pieces of deployment and working ourselves through until the Kubernetes services, which route private and public access to the containerized applications we run in Kubernetes.

We then got a brief overview of the most popular hosted Kubernetes offerings in the cloud and ended the chapter by discussing the place of containers and container orchestrators such as Kubernetes in the context of the current serverless hype.

Questions

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

1. Explain in a few short sentences what the role of a Kubernetes master is.
2. List the elements that need to be present on each Kubernetes (worker) node.
3. We cannot run individual containers in a Kubernetes cluster.
   a. Yes
   b. No
4. Explain the reason why the containers in a pod can use localhost to communicate with each other.

5. What is the purpose of the so-called pause container in a pod?
6. Bob tells you, "Our application consists of three Docker images: `web`, `inventory`, and `db`. Since we can run multiple containers in a Kubernetes pod, we are going to deploy all the services of our application in a single pod." List three to four reasons why this is a bad idea.
7. Explain in your own words why we need Kubernetes ReplicaSets.
8. Under which circumstances do we need Kubernetes Deployments?
9. List at least three types of Kubernetes services and explain their purposes and their differences.

Answers

1. The Kubernetes master is responsible for managing the cluster. All requests to create objects, reschedule pods, manage ReplicaSets, and more happen on the master. The master does not run the application workload in a production or production-like cluster.
2. On each worker node, we have the kubelet, the proxy, and the container runtime.
3. The answer is a. Yes. You cannot run standalone containers on a Kubernetes cluster. Pods are the atomic units of deployment in such a cluster.
4. All containers running inside a pod share the same Linux kernel network namespace. Thus, all processes running inside those containers can communicate with each other through localhost in a similar way to how processes or applications directly running on the host can communicate with each other through localhost.
5. The pause container's sole role is to reserve the namespaces of the pod for containers that run in it.
6. This is a bad idea since all containers of a pod are co-located, which means they run on the same cluster node. Also, if multiple containers run in the same pod, they can only be scaled up or down all at once. However, the different components of the application (that is, Web, inventory, and db) usually have very different requirements with regard to scalability or resource consumption. The Web component might need to be scaled up and down depending on the traffic, and the db component, in turn, has special requirements regarding storage that the others do not have. If we do run every component in its own pod, we are much more flexible in this regard.

5. What is the purpose of the so-called pause container in a pod?
6. Bob tells you, "Our application consists of three Docker images: `web`, `inventory`, and `db`. Since we can run multiple containers in a Kubernetes pod, we are going to deploy all the services of our application in a single pod." List three to four reasons why this is a bad idea.
7. Explain in your own words why we need Kubernetes ReplicaSets.
8. Under which circumstances do we need Kubernetes Deployments?
9. List at least three types of Kubernetes services and explain their purposes and their differences.

Answers

1. The Kubernetes master is responsible for managing the cluster. All requests to create objects, reschedule pods, manage ReplicaSets, and more happen on the master. The master does not run the application workload in a production or production-like cluster.
2. On each worker node, we have the kubelet, the proxy, and the container runtime.
3. The answer is a. Yes. You cannot run standalone containers on a Kubernetes cluster. Pods are the atomic units of deployment in such a cluster.
4. All containers running inside a pod share the same Linux kernel network namespace. Thus, all processes running inside those containers can communicate with each other through localhost in a similar way to how processes or applications directly running on the host can communicate with each other through localhost.
5. The pause container's sole role is to reserve the namespaces of the pod for containers that run in it.
6. This is a bad idea since all containers of a pod are co-located, which means they run on the same cluster node. Also, if multiple containers run in the same pod, they can only be scaled up or down all at once. However, the different components of the application (that is, Web, inventory, and db) usually have very different requirements with regard to scalability or resource consumption. The Web component might need to be scaled up and down depending on the traffic, and the db component, in turn, has special requirements regarding storage that the others do not have. If we do run every component in its own pod, we are much more flexible in this regard.

7. We need a mechanism in order to run multiple instances of a pod in a cluster and make sure that the actual number of pods running always corresponds to the desired number, even when individual pods crash or disappear due to network partition or cluster node failures. The ReplicaSet is the mechanism that provides scalability and self-healing to any application service.
8. We need deployment objects whenever we want to update an application service in a Kubernetes cluster without causing downtime to the service. Deployment objects add rolling updates and rollback capabilities to ReplicaSets.
9. Kubernetes service objects are used to make application services participate in service discovery. They provide a stable endpoint to a set of pods (normally governed by a ReplicaSet or a deployment). Kube services are abstractions that define a logical set of pods and a policy regarding how to access them. There are four types of Kube services:
    a. **ClusterIP**: Exposes the service on an IP address that's only accessible from inside the cluster; this is a virtual IP (VIP).
    b. **NodePort**: Publishes a port in the range 30,000–32,767 on every cluster node.
    c. **LoadBalancer**: This type exposes the application service externally using a cloud provider's load balancer, such as ELB on AWS.
    d. **ExternalName**: Used when you need to define a proxy for a cluster's external service, such as a database.

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 reputed financial institution. During the interview, you are confronted with the following questions:

1. Why do we need an orchestrator such as Kubernetes? Can we not just manage a few containers in the cluster manually or with the help of some scripts?
2. What are some container orchestration engines that you are aware of, and which one would you recommend?
3. We are a bit confused about the difference between a Kubernetes `ReplicaSet` and a `Deployment`. What is the difference?

Possible answers to interview questions

Here are some possible answers to the questions:

1. A mission-critical, highly available application that is implemented as a highly distributed system of interconnected application services that are

7. We need a mechanism in order to run multiple instances of a pod in a cluster and make sure that the actual number of pods running always corresponds to the desired number, even when individual pods crash or disappear due to network partition or cluster node failures. The ReplicaSet is the mechanism that provides scalability and self-healing to any application service.
8. We need deployment objects whenever we want to update an application service in a Kubernetes cluster without causing downtime to the service. Deployment objects add rolling updates and rollback capabilities to ReplicaSets.
9. Kubernetes service objects are used to make application services participate in service discovery. They provide a stable endpoint to a set of pods (normally governed by a ReplicaSet or a deployment). Kube services are abstractions that define a logical set of pods and a policy regarding how to access them. There are four types of Kube services:
    a. **ClusterIP**: Exposes the service on an IP address that's only accessible from inside the cluster; this is a virtual IP (VIP).
    b. **NodePort**: Publishes a port in the range 30,000–32,767 on every cluster node.
    c. **LoadBalancer**: This type exposes the application service externally using a cloud provider's load balancer, such as ELB on AWS.
    d. **ExternalName**: Used when you need to define a proxy for a cluster's external service, such as a database.

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 reputed financial institution. During the interview, you are confronted with the following questions:

1. Why do we need an orchestrator such as Kubernetes? Can we not just manage a few containers in the cluster manually or with the help of some scripts?
2. What are some container orchestration engines that you are aware of, and which one would you recommend?
3. We are a bit confused about the difference between a Kubernetes `ReplicaSet` and a `Deployment`. What is the difference?

Possible answers to interview questions

Here are some possible answers to the questions:

1. A mission-critical, highly available application that is implemented as a highly distributed system of interconnected application services that are

just too complex to manually monitor, operate, and manage. Container orchestrators help in this regard. They automate most of the typical tasks, such as reconciling a desired state or collecting and aggregating key metrics of the system. Humans cannot react quick enough to make such an application elastic or self-healing. Software support is needed for this in the form of the mentioned container orchestrators.

2. The winner in the space of container orchestration engines is Kubernetes, which is open-sourced and owned by CNCF. It was originally developed by Google. We also have Docker Swarm, which is proprietary and has been developed by Docker. AWS offers a container service called ECS, which is also proprietary and tightly integrated into the AWS ecosystem. Finally, Microsoft offers AKS, which has the same pros and cons as AWS ECS.

   Given that Kubernetes is by far dominating the space, I would go with it since there is, as a consequence, very much talent on the market that has working experience with it, and the ecosystem is huge. Furthermore, most questions that one has when working with Kubernetes have been answered on StackOverflow.

3. A Kubernetes Deployment offers a few additional services on top of what a ReplicaSet offers. The latter guarantees the availability of a desired set of identical pods. The former also provides update and rollback services on top of that.

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



just too complex to manually monitor, operate, and manage. Container orchestrators help in this regard. They automate most of the typical tasks, such as reconciling a desired state or collecting and aggregating key metrics of the system. Humans cannot react quick enough to make such an application elastic or self-healing. Software support is needed for this in the form of the mentioned container orchestrators.

2. The winner in the space of container orchestration engines is Kubernetes, which is open-sourced and owned by CNCF. It was originally developed by Google. We also have Docker Swarm, which is proprietary and has been developed by Docker. AWS offers a container service called ECS, which is also proprietary and tightly integrated into the AWS ecosystem. Finally, Microsoft offers AKS, which has the same pros and cons as AWS ECS.

   Given that Kubernetes is by far dominating the space, I would go with it since there is, as a consequence, very much talent on the market that has working experience with it, and the ecosystem is huge. Furthermore, most questions that one has when working with Kubernetes have been answered on StackOverflow.

3. A Kubernetes Deployment offers a few additional services on top of what a ReplicaSet offers. The latter guarantees the availability of a desired set of identical pods. The former also provides update and rollback services on top of that.

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 11
Leveraging Docker Logs to Provide Insight into Your Apps

Introduction

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 your 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.

We will be using VS Code and Docker Desktop with Kubernetes as our main tools in this chapter.

Structure

In this chapter, we will discuss the following topics:

- Monitoring your platform
- Collecting container logs using Prometheus
- Monitoring application key metrics using Grafana
- Defining alerts
- Finding the root cause of an issue

Chapter 11
Leveraging Docker Logs to Provide Insight into Your Apps

Introduction

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 your 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.

We will be using VS Code and Docker Desktop with Kubernetes as our main tools in this chapter.

Structure

In this chapter, we will discuss the following topics:

- Monitoring your platform
- Collecting container logs using Prometheus
- Monitoring application key metrics using Grafana
- Defining alerts
- Finding the root cause of an issue

Objectives

After finishing this chapter, you will know why logging, monitoring, and alerting are so important for any mission-critical application that you are running in production. You will be able to collect container logs centrally and use Prometheus to scrape the container metric of your application and also collect them in a central location. You will be able to display key metrics on dashboards in Grafana, which will make it simpler to you to monitor your applications. You will also know how you can define tolerance bands or limits that, when exceeded, will trigger alerts you defined. All this will help you to more easily find the root cause of a production issue shown by your application.

Let us start by defining what monitoring really means in the context of a distributed and mission-critical application such as a container-based application running on Kubernetes in the cloud.

Monitoring your platform

If you have a distributed and mission-critical application running in production, you will want to understand the health of the platform. If you are not able to do that, you are flying blind, and that is a bad position to be in, to say the least. So, what can we do?

There are fundamentally two different angles of view that you can take at your application. The first one is observing or monitoring the whole application from a very technical perspective. This kind of monitoring includes observing the usage of CPU, memory, storage, or network bandwidth by the different components contributing to the overall system. The other perspective we can take is a decidedly business-centric view. Businesses have different ideas about if the application is healthy or not. They are interested if customers are able to use the application if the response times are within the defined limits, and if the application is actually making money, that is, if products are sold or not if the application is an e-commerce platform. Often we also call the former the non-functional view and the latter the functional view.

Next to the two viewing angles, technical or business oriented, we can also distinguish different things we want to monitor. First, let us talk about log entries.

Log entries

A log entry is nothing more than some status text annotated with some metadata, such as the exact date and time when the respective log entry has been created, the name of the application service whose code generated the log entry, or the name of the host on which the respective application service was running, when the log

Objectives

After finishing this chapter, you will know why logging, monitoring, and alerting are so important for any mission-critical application that you are running in production. You will be able to collect container logs centrally and use Prometheus to scrape the container metric of your application and also collect them in a central location. You will be able to display key metrics on dashboards in Grafana, which will make it simpler to you to monitor your applications. You will also know how you can define tolerance bands or limits that, when exceeded, will trigger alerts you defined. All this will help you to more easily find the root cause of a production issue shown by your application.

Let us start by defining what monitoring really means in the context of a distributed and mission-critical application such as a container-based application running on Kubernetes in the cloud.

Monitoring your platform

If you have a distributed and mission-critical application running in production, you will want to understand the health of the platform. If you are not able to do that, you are flying blind, and that is a bad position to be in, to say the least. So, what can we do?

There are fundamentally two different angles of view that you can take at your application. The first one is observing or monitoring the whole application from a very technical perspective. This kind of monitoring includes observing the usage of CPU, memory, storage, or network bandwidth by the different components contributing to the overall system. The other perspective we can take is a decidedly business-centric view. Businesses have different ideas about if the application is healthy or not. They are interested if customers are able to use the application if the response times are within the defined limits, and if the application is actually making money, that is, if products are sold or not if the application is an e-commerce platform. Often we also call the former the non-functional view and the latter the functional view.

Next to the two viewing angles, technical or business oriented, we can also distinguish different things we want to monitor. First, let us talk about log entries.

Log entries

A log entry is nothing more than some status text annotated with some metadata, such as the exact date and time when the respective log entry has been created, the name of the application service whose code generated the log entry, or the name of the host on which the respective application service was running, when the log

entry was produced. Log entries have a few default log levels. The following are the commonly used levels:

Level	Description
Debug	This is information that helps to debug an application. It is very technical information, and normally only developers familiar with the code can understand its meaning
Info	This is—as the name points out—information about what is happening in the application. No action required
Warning	This is a warning that indicates that something happened that could potentially be a problem but is not critical. It is like the warning light in a car telling you that your gasoline tank will soon be empty. You do not have to panic but should take care of it eventually
Error	An error happened in the application. What just happened should not have happened. Often, a well-written application can recover from such an error. Nevertheless, the SREs (system reliability enginners) should absolutely take note of it and try to solve the root cause of the error such as that it does not occur again
Fatal	Something terrible happened. The application should fail fast in such a situation as these types of errors are not recoverable. The immediate reaction of the SREs is needed

***Table 11.1:** Commonly used levels*

If the developers use a logging library for their framework of choice, such as log4j for Java, then it is very easy to generate such log output. The application can then be configured to limit the output of only certain log levels. For example, during development, the developers want to see all log entries. But in production, one would only ever want to see log entries that are at least a warning or higher. Why is that so? If you collect all the log entries for level debug and info, too, you may result with a vast amount of data. And if you use a commercial log aggregator such as Splunk, the price varies with the throughput of data. Keep that in mind.

Log entries can contain any information and are often unstructured. They can help to quickly pinpoint the root cause of a production issue, should one arise. You see, generating log output is important. But be precise and avoid generic statements. The log entry should tell exactly what has happened and why.

Centrally collection logs

When working with containers and running them on Kubernetes, the task of remaining on top of all the information logged by the different containers running

entry was produced. Log entries have a few default log levels. The following are the commonly used levels:

Level	Description
Debug	This is information that helps to debug an application. It is very technical information, and normally only developers familiar with the code can understand its meaning
Info	This is—as the name points out—information about what is happening in the application. No action required
Warning	This is a warning that indicates that something happened that could potentially be a problem but is not critical. It is like the warning light in a car telling you that your gasoline tank will soon be empty. You do not have to panic but should take care of it eventually
Error	An error happened in the application. What just happened should not have happened. Often, a well-written application can recover from such an error. Nevertheless, the SREs (system reliability enginners) should absolutely take note of it and try to solve the root cause of the error such as that it does not occur again
Fatal	Something terrible happened. The application should fail fast in such a situation as these types of errors are not recoverable. The immediate reaction of the SREs is needed

***Table 11.1:** Commonly used levels*

If the developers use a logging library for their framework of choice, such as log4j for Java, then it is very easy to generate such log output. The application can then be configured to limit the output of only certain log levels. For example, during development, the developers want to see all log entries. But in production, one would only ever want to see log entries that are at least a warning or higher. Why is that so? If you collect all the log entries for level debug and info, too, you may result with a vast amount of data. And if you use a commercial log aggregator such as Splunk, the price varies with the throughput of data. Keep that in mind.

Log entries can contain any information and are often unstructured. They can help to quickly pinpoint the root cause of a production issue, should one arise. You see, generating log output is important. But be precise and avoid generic statements. The log entry should tell exactly what has happened and why.

Centrally collection logs

When working with containers and running them on Kubernetes, the task of remaining on top of all the information logged by the different containers running

in a cluster, the container runtime on each node of the cluster, and the cluster itself, is a major undertaken. Applications running inside containers in Kubernetes pods usually produce the log output into stdout or stderr, from where Kubernetes, by default, collects and stores them in files on the cluster node on which the pod runs. As we have learned, containers and pods are often ephemeral. They come and go. A pod can crash and automatically be relaunched by Kubernetes on a different node of the cluster. Kubernetes cleans up the system automatically. When a pod crashes or is stopped and evicted, then its resources, including its logs, are freed from the system. This way, valuable information gets lost.

To remain in control in a distributed system consisting of many parts, it is super important to collect logs in real-time from all sources in a central place, where they can then be aggregated and searched in an effective way. This way also, logs never get lost, even though pods may continue to crash or be evicted and rerun on different cluster nodes.

Many commercials and also free solutions exist to solve this problem of collecting and centrally aggregating important logs.

Now, let us move on to metrics, which are another important ingredient used to monitor a system.

Metrics

Contrary to log entries which can contain simple text, metrics are more specific and structured. The word metric, according to Merriam-Webster, means: "*...quantitative study as by automated methods of data processing...*". Evidently, we are talking of measuring a quantity, such as the number of products bought over a given time period or the total number of payments made onto a given account, and so on. How do we get such metrics? It is some process that needs to do the measurement. Metrics can be more system-level, such as the amount of memory a service is using or the percentage of CPU that is used by all processes running on a server. Metrics can also be more business focused, such as the ones mentioned previously, which were the number of products bought per time period or the number of deposits made into an account.

The former are called non-functional metrics, and the latter consequently are functional metrics. Non-functional metrics are usually measured by some low-level utilities such as Node Exporter or cAdvisor. To collect functional metrics, developers have to instrument their code. Different types of measurements exist. The simplest one is a **counter**, which is steadily increased from zero onwards each time the thing we want to count occurs, that is, for example, each time a customer buys a product. Another measurement type is a **gauge** that can be compared to the gauges you well know from the dashboard of your car. The current speed of the car is displayed as a gauge. A gauge usually goes from a minimal value, say zero, to an upper value, say

in a cluster, the container runtime on each node of the cluster, and the cluster itself, is a major undertaken. Applications running inside containers in Kubernetes pods usually produce the log output into stdout or stderr, from where Kubernetes, by default, collects and stores them in files on the cluster node on which the pod runs. As we have learned, containers and pods are often ephemeral. They come and go. A pod can crash and automatically be relaunched by Kubernetes on a different node of the cluster. Kubernetes cleans up the system automatically. When a pod crashes or is stopped and evicted, then its resources, including its logs, are freed from the system. This way, valuable information gets lost.

To remain in control in a distributed system consisting of many parts, it is super important to collect logs in real-time from all sources in a central place, where they can then be aggregated and searched in an effective way. This way also, logs never get lost, even though pods may continue to crash or be evicted and rerun on different cluster nodes.

Many commercials and also free solutions exist to solve this problem of collecting and centrally aggregating important logs.

Now, let us move on to metrics, which are another important ingredient used to monitor a system.

Metrics

Contrary to log entries which can contain simple text, metrics are more specific and structured. The word metric, according to Merriam-Webster, means: "*...quantitative study as by automated methods of data processing...*". Evidently, we are talking of measuring a quantity, such as the number of products bought over a given time period or the total number of payments made onto a given account, and so on. How do we get such metrics? It is some process that needs to do the measurement. Metrics can be more system-level, such as the amount of memory a service is using or the percentage of CPU that is used by all processes running on a server. Metrics can also be more business focused, such as the ones mentioned previously, which were the number of products bought per time period or the number of deposits made into an account.

The former are called non-functional metrics, and the latter consequently are functional metrics. Non-functional metrics are usually measured by some low-level utilities such as Node Exporter or cAdvisor. To collect functional metrics, developers have to instrument their code. Different types of measurements exist. The simplest one is a **counter**, which is steadily increased from zero onwards each time the thing we want to count occurs, that is, for example, each time a customer buys a product. Another measurement type is a **gauge** that can be compared to the gauges you well know from the dashboard of your car. The current speed of the car is displayed as a gauge. A gauge usually goes from a minimal value, say zero, to an upper value, say

240 km/h, in the case of a car. As you accelerate or decelerate, the current value on the gauge changes. Similarly, gauges work when used to instrument an application. A typical usage scenario would be to measure the current number of users on our application. In more generic terms, *a gauge is a metric that represents a single numerical value that can arbitrarily go up and down.*

The third type of measurement type is **histograms**. A histogram is *sampling observations and counts them in configurable buckets*. An example would be request response times grouped by time ranges. Each time a measurement is made, the counter of the bucket is increased, within which the measured time lies. Similarly, one could measure the sum or the orders made in the system and group them in value ranges.

The fourth and last type of metric is **summaries**. A summary is similar to a histogram, but it calculates quantiles over a sliding time window.

Now if the developers instrument their code and define counters, gauges, histograms, and summaries, those values need to be collected by some process. Interestingly the very popular Prometheus middleware can do exactly this. Prometheus has become a de facto industry standard. It periodically and repeatedly scrapes defined metrics from registered endpoints. By convention services, which have some Prometheus-compatible metrics to expose, do this on the endpoint `/metrics`. For most development languages, libraries exist that export metrics in a Prometheus-compatible format on that default endpoint.

Once we have centrally collected all metrics of the application, we can query them from within Prometheus. The tool has a powerful query language that makes it easy to extract exactly the information we are looking for from a heap of values. Prometheus even allows us to visualize the metrics, but the graphics capabilities are rather limited. We need better tooling for the graphical representation of metrics in the form of dashboards.

Dashboards

Dashboards are widely used to display a group of related metrics in a visually pleasing and meaningful way. The intent of a dashboard is to present essential data in a way that the observer can quickly draw conclusions from what he or she sees. Raw numbers are hard to grasp for humans, specifically if there are vast amounts of numbers. We have a very hard time to spot interesting patterns in a sequence of numbers. A graphical representation in the form of a trend line is much easier to grasp for us observers. And this is the basic idea of why we need dashboards. Many free and commercial tools exist that we can use to build interesting dashboards. The most popular free tools are Kibana and Grafana. Commercial tools are DataDog,

240 km/h, in the case of a car. As you accelerate or decelerate, the current value on the gauge changes. Similarly, gauges work when used to instrument an application. A typical usage scenario would be to measure the current number of users on our application. In more generic terms, *a gauge is a metric that represents a single numerical value that can arbitrarily go up and down.*

The third type of measurement type is **histograms**. A histogram is *sampling observations and counts them in configurable buckets*. An example would be request response times grouped by time ranges. Each time a measurement is made, the counter of the bucket is increased, within which the measured time lies. Similarly, one could measure the sum or the orders made in the system and group them in value ranges.

The fourth and last type of metric is **summaries**. A summary is similar to a histogram, but it calculates quantiles over a sliding time window.

Now if the developers instrument their code and define counters, gauges, histograms, and summaries, those values need to be collected by some process. Interestingly the very popular Prometheus middleware can do exactly this. Prometheus has become a de facto industry standard. It periodically and repeatedly scrapes defined metrics from registered endpoints. By convention services, which have some Prometheus-compatible metrics to expose, do this on the endpoint `/metrics`. For most development languages, libraries exist that export metrics in a Prometheus-compatible format on that default endpoint.

Once we have centrally collected all metrics of the application, we can query them from within Prometheus. The tool has a powerful query language that makes it easy to extract exactly the information we are looking for from a heap of values. Prometheus even allows us to visualize the metrics, but the graphics capabilities are rather limited. We need better tooling for the graphical representation of metrics in the form of dashboards.

Dashboards

Dashboards are widely used to display a group of related metrics in a visually pleasing and meaningful way. The intent of a dashboard is to present essential data in a way that the observer can quickly draw conclusions from what he or she sees. Raw numbers are hard to grasp for humans, specifically if there are vast amounts of numbers. We have a very hard time to spot interesting patterns in a sequence of numbers. A graphical representation in the form of a trend line is much easier to grasp for us observers. And this is the basic idea of why we need dashboards. Many free and commercial tools exist that we can use to build interesting dashboards. The most popular free tools are Kibana and Grafana. Commercial tools are DataDog,

New Relic, Splunk, and others. An interesting dashboard example is given in the following image:



Figure 11.1: Grafana dashboard for a container-based application

In the previous image, we see a dashboard as defined in Grafana. It contains at least two types of metrics, counters, and gauges. Counters are represented as pure numbers, such as the number of CPU Cores. Gauges, on the other hand, are represented as a gauge such as the Memory Load or as trend lines such as the Container CPU Usage graph, which shows the value changing over time.

Alerts

If you do think that collecting logs and metrics and displaying the latter in nice dashboards is enough, then you will be disappointed. If we end with the dashboards, then some support personnel has to sit in front of a plethora of monitors all the time, 24 * 7, 365 days a year, just in case something unforeseen happens. This is a boring task, to say the least. What if the person falls asleep? We need to change our strategy. Let us define alerts that are triggered when something exceptional happens. How we are alerting can be different. It can be a text message or a call to your phone, a

New Relic, Splunk, and others. An interesting dashboard example is given in the following image:



Figure 11.1: Grafana dashboard for a container-based application

In the previous image, we see a dashboard as defined in Grafana. It contains at least two types of metrics, counters, and gauges. Counters are represented as pure numbers, such as the number of CPU Cores. Gauges, on the other hand, are represented as a gauge such as the Memory Load or as trend lines such as the Container CPU Usage graph, which shows the value changing over time.

Alerts

If you do think that collecting logs and metrics and displaying the latter in nice dashboards is enough, then you will be disappointed. If we end with the dashboards, then some support personnel has to sit in front of a plethora of monitors all the time, 24 * 7, 365 days a year, just in case something unforeseen happens. This is a boring task, to say the least. What if the person falls asleep? We need to change our strategy. Let us define alerts that are triggered when something exceptional happens. How we are alerting can be different. It can be a text message or a call to your phone, a

pager message if you are on duty, an e-mail, or even switching on an alarm siren and some flashing alarm lights. It totally depends on the use case. Let us just say that the author has contributed to various applications that used all of the previous means to alert people.

Alerts are based on rules that use metrics as input values. What it means is simply that someone has to define the metrics that are important, and an alert is warranted if the metric exceeds a certain value, either repeatedly or during a predefined time range. As an example, take CPU usage. The SRE wants to be notified when the CPU usage of a node of the Kubernetes cluster is above 95% for a duration of more than 1 minute.

But who has to define the rules, you may ask? In the case of non-functional metrics, it is operations or, more specifically, the SREs that have to define what is noteworthy and in which case they want to be alerted even in the middle of the night. For functional metrics, it is the business that has to define the metrics and the tolerance bands or rules for those metrics that will trigger an alert.

Now that we have discussed alerts let us concentrate on what comes next, the run books.

Run books

Imagine an alert is triggered. What is next? This is where run books come in. A run book defines for each alert who has to be alerted, what this person has to do to address the underlying issue, and, in case the issue cannot be fixed, to whom the issue has to be escalated. Creating run books is a challenging task and should not be underestimated. But they are an essential tool for a company. An SRE can only do so much. Some production issues are so severe that the alert has to be escalated to the C-level management. Imagine, for example, the situation where you are operating an e-commerce application, and your **Payment Service Provider (PSP)** is down, and thus, no payments on your platform can be performed. This means that your application just lost one of the essential must-haves. Basically, you cannot do any business anymore until the issue is resolved; at least your CTO should know about this, don't you think?

Well, after having explained the theory behind monitoring and alerting, it is time to break it down and do some simple examples. Let us begin by looking into how to centrally collect container logs.

Collecting container logs

As an example, let us show how to use SemaText (**https://sematext.com**) to collect logs from a single-node Kubernetes cluster. We are using Docker Desktop for this purpose. As mentioned earlier, there exist numerous ways how one can collect and

pager message if you are on duty, an e-mail, or even switching on an alarm siren and some flashing alarm lights. It totally depends on the use case. Let us just say that the author has contributed to various applications that used all of the previous means to alert people.

Alerts are based on rules that use metrics as input values. What it means is simply that someone has to define the metrics that are important, and an alert is warranted if the metric exceeds a certain value, either repeatedly or during a predefined time range. As an example, take CPU usage. The SRE wants to be notified when the CPU usage of a node of the Kubernetes cluster is above 95% for a duration of more than 1 minute.

But who has to define the rules, you may ask? In the case of non-functional metrics, it is operations or, more specifically, the SREs that have to define what is noteworthy and in which case they want to be alerted even in the middle of the night. For functional metrics, it is the business that has to define the metrics and the tolerance bands or rules for those metrics that will trigger an alert.

Now that we have discussed alerts let us concentrate on what comes next, the run books.

Run books

Imagine an alert is triggered. What is next? This is where run books come in. A run book defines for each alert who has to be alerted, what this person has to do to address the underlying issue, and, in case the issue cannot be fixed, to whom the issue has to be escalated. Creating run books is a challenging task and should not be underestimated. But they are an essential tool for a company. An SRE can only do so much. Some production issues are so severe that the alert has to be escalated to the C-level management. Imagine, for example, the situation where you are operating an e-commerce application, and your **Payment Service Provider (PSP)** is down, and thus, no payments on your platform can be performed. This means that your application just lost one of the essential must-haves. Basically, you cannot do any business anymore until the issue is resolved; at least your CTO should know about this, don't you think?

Well, after having explained the theory behind monitoring and alerting, it is time to break it down and do some simple examples. Let us begin by looking into how to centrally collect container logs.

Collecting container logs

As an example, let us show how to use SematText (**https://sematext.com**) to collect logs from a single-node Kubernetes cluster. We are using Docker Desktop for this purpose. As mentioned earlier, there exist numerous ways how one can collect and

aggregate logs centrally. SemaText was chosen because of the ease of setup. But other options, such as the open-source ELK stack or various commercial solutions, should also be considered.

1. Open the dashboard of Docker Desktop, then follow the following steps:
   a. Go to the settings page (click the cogwheel icon in the upper right corner)
   b. On the left side select the **Kubernetes** tab
   c. Click the checkbox next to **Enable Kubernetes**
2. It may take a while until Kubernetes is ready to go as Docker Desktop has to download many Docker images containing Kubernetes components and then configure the cluster. Thus, be patient.
3. In the meantime, open a terminal and install Helm. If you are on a Mac, you can use the command:

   ```
   $ brew install Helm
   ```

   On Windows, use Chocolatey to do the same.
4. Use the command `helm version` to check that the installation went as planned.
5. Create a trial account on **https://sematext.com** and log in to your new trial account.
6. On the **Getting started** screen of SemaText, click the button **Start shipping logs.**
7. When asked to select the environment, choose **Kubernetes** from the list of available options.
8. You will be asked to create an infra app. Call it **kube-demo.**
9. Follow the instructions and select Helm 3 as the configuration method. This will ask you to execute the following commands:

   ```
   helm repo add sematext https://cdn.sematext.com/helm-charts
   helm repo update
   helm install st-agent \
     --set infraToken=<infra-token> \
     --set region=<region> \
     --namespace=sematext \
     --create-namespace \
     sematext/sematext-agent
   ```

aggregate logs centrally. SemaText was chosen because of the ease of setup. But other options, such as the open-source ELK stack or various commercial solutions, should also be considered.

1. Open the dashboard of Docker Desktop, then follow the following steps:
   a. Go to the settings page (click the cogwheel icon in the upper right corner)
   b. On the left side select the **Kubernetes** tab
   c. Click the checkbox next to **Enable Kubernetes**
2. It may take a while until Kubernetes is ready to go as Docker Desktop has to download many Docker images containing Kubernetes components and then configure the cluster. Thus, be patient.
3. In the meantime, open a terminal and install Helm. If you are on a Mac, you can use the command:

   ```
   $ brew install Helm
   ```

   On Windows, use Chocolatey to do the same.
4. Use the command `helm version` to check that the installation went as planned.
5. Create a trial account on **https://sematext.com** and log in to your new trial account.
6. On the **Getting started** screen of SemaText, click the button **Start shipping logs.**
7. When asked to select the environment, choose **Kubernetes** from the list of available options.
8. You will be asked to create an infra app. Call it **kube-demo.**
9. Follow the instructions and select Helm 3 as the configuration method. This will ask you to execute the following commands:

   ```
   helm repo add sematext https://cdn.sematext.com/helm-charts
   helm repo update
   helm install st-agent \
     --set infraToken=<infra-token> \
     --set region=<region> \
     --namespace=sematext \
     --create-namespace \
     sematext/sematext-agent
   ```

10. The previously mentioned three commands will (1) add a new helm repo from where repos can be loaded, (2) update Helm with the new repo list, and (3) use Helm to download and run the SematText agent, that will be used to collect logs and ship them to your SematText account. Please make sure to replace **<infra-token>** with the token provided to you by SematText and <region> with the region you did select when creating your trial account.

11. After a moment, you should see in SematText that your cluster has been connected:

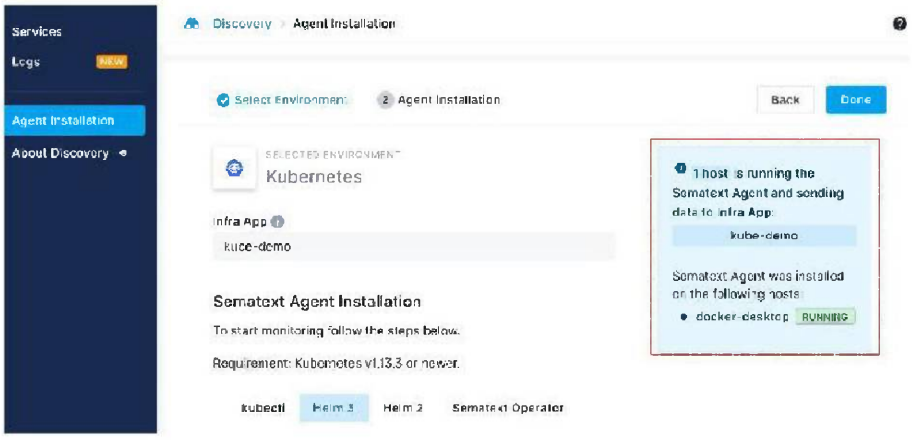


Figure 11.2: Kubernetes cluster is exporting logs to SematText

12. Click on **Done**. To verify that indeed data is flowing to SematText, we are now exploring the collected information and observing what SematText is doing with it.

10. The previously mentioned three commands will (1) add a new helm repo from where repos can be loaded, (2) update Helm with the new repo list, and (3) use Helm to download and run the SematText agent, that will be used to collect logs and ship them to your SematText account. Please make sure to replace **<infra-token>** with the token provided to you by SematText and <region> with the region you did select when creating your trial account.

11. After a moment, you should see in SematText that your cluster has been connected:

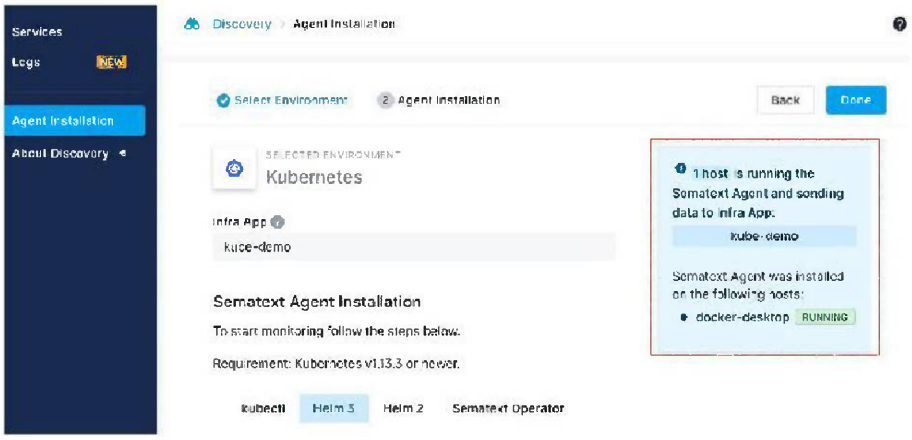


Figure 11.2: Kubernetes cluster is exporting logs to SematText

12. Click on **Done**. To verify that indeed data is flowing to SematText, we are now exploring the collected information and observing what SematText is doing with it.

13. Select the **Infrastructure** tab on the left side of SemaText and then select **Containers Overview**. You should see something similar to the following image:

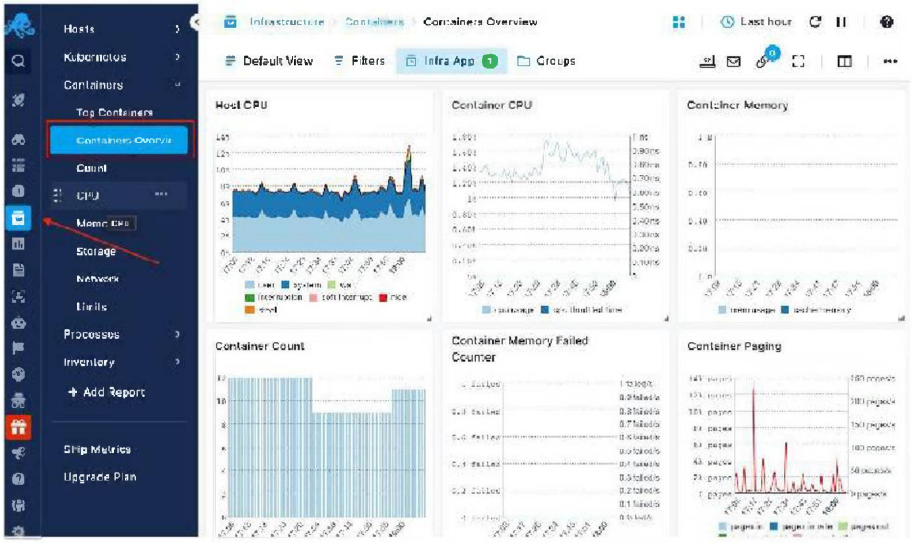


Figure 11.3: Overview of containers in SemaText

As you can see in the previous image, the data that is flowing from our single node Kubernetes cluster to SemaText is used to generate very helpful graphs, such as the host CPU utilization of the container memory usage over time. Please explore on your own the many views into the received data that SemaText offers.

14. So far, we have only collected infrastructure level information through the according logs. We did not collect any container logs. As an exercise, please try to configure your SemaText trial account in a way that it also collects container logs. For this, you may also want to run one or several own pods, such as Nginx, and connect them with SemaText.

Great, you have just seen how important it is to collect log information from your infrastructure, from your containers, and from the applications centrally, and what insight into your system you can gain out of this information with little effort.

In the next section, we are going to use Prometheus to scrape metrics from various container-related resources.

13. Select the **Infrastructure** tab on the left side of SemaText and then select **Containers Overview**. You should see something similar to the following image:

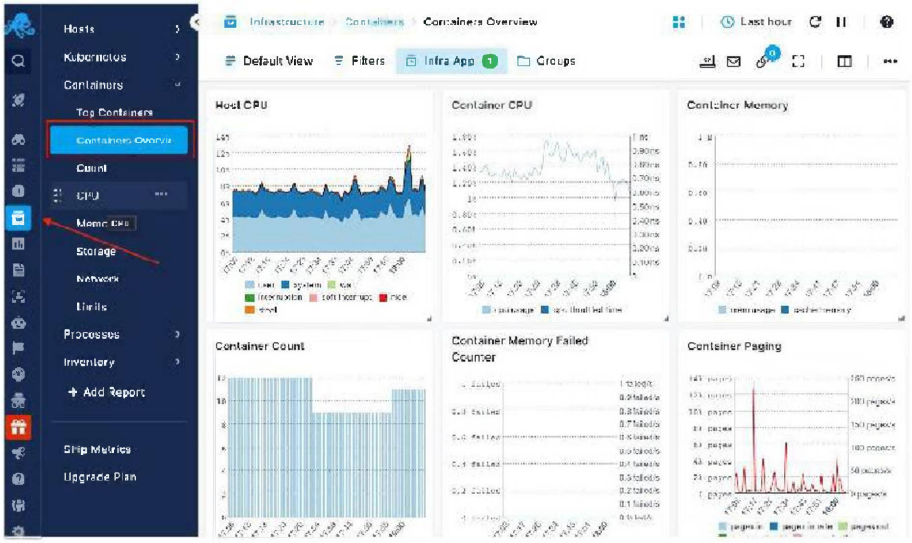


Figure 11.3: Overview of containers in SemaText

As you can see in the previous image, the data that is flowing from our single node Kubernetes cluster to SemaText is used to generate very helpful graphs, such as the host CPU utilization of the container memory usage over time. Please explore on your own the many views into the received data that SemaText offers.

14. So far, we have only collected infrastructure level information through the according logs. We did not collect any container logs. As an exercise, please try to configure your SemaText trial account in a way that it also collects container logs. For this, you may also want to run one or several own pods, such as Nginx, and connect them with SemaText.

Great, you have just seen how important it is to collect log information from your infrastructure, from your containers, and from the applications centrally, and what insight into your system you can gain out of this information with little effort.

In the next section, we are going to use Prometheus to scrape metrics from various container-related resources.

Scraping container metrics using Prometheus

Applications running inside a container produce logs. These logs reside inside the container itself and, if not taken care of, will be lost once the container is gone. Thus, we need a mechanism to collect the logs from all the containers running in on our Docker host or, even better, from all containers running in our cluster, be it Kubernetes, Docker Swarm, or any other container orchestrator-managed cluster. But how can we collect the logs? There are different means to have log entries shipped to a central agent as they are generated. One such means is the use of cAdvisor. cAdvisor stands for container advisor and is an open-source tool for running performance analysis and collecting usage data from the containers. cAdvisor normally runs as a daemon process in the background, collecting, processing, and aggregating useful operational information. Alternatively, we could also use Fluentd for the job.

cAdvisor is often used with Prometheus in the industry. We can then stream the data from cAdvisor into Prometheus, and then, plugin Grafan to visually display the collected metrics on a series of dashboards. The high-level architecture of this approach will look like the following:

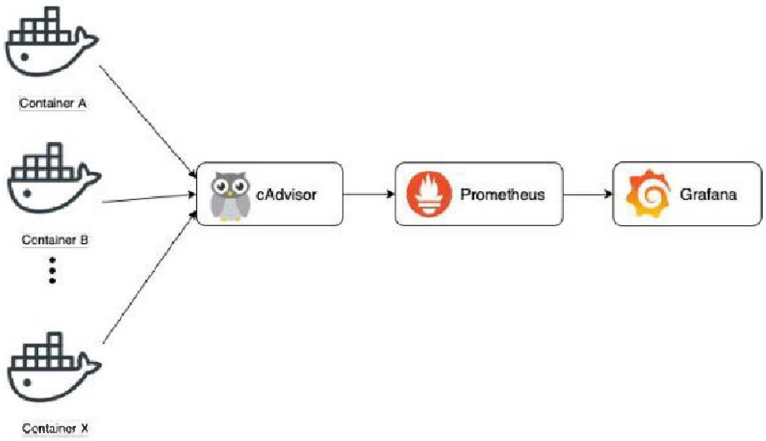


Figure 11.4: *Container monitoring architecture*

Monitoring Docker metrics with cAdvisor

To be able to monitor Docker container metrics with cAdvisor, proceed as follows:

1. Create a configuration file for Prometheus instructing it to scrape metrics from cAdvisor. For this:

Scraping container metrics using Prometheus

Applications running inside a container produce logs. These logs reside inside the container itself and, if not taken care of, will be lost once the container is gone. Thus, we need a mechanism to collect the logs from all the containers running in on our Docker host or, even better, from all containers running in our cluster, be it Kubernetes, Docker Swarm, or any other container orchestrator-managed cluster. But how can we collect the logs? There are different means to have log entries shipped to a central agent as they are generated. One such means is the use of cAdvisor. cAdvisor stands for container advisor and is an open-source tool for running performance analysis and collecting usage data from the containers. cAdvisor normally runs as a daemon process in the background, collecting, processing, and aggregating useful operational information. Alternatively, we could also use Fluentd for the job.

cAdvisor is often used with Prometheus in the industry. We can then stream the data from cAdvisor into Prometheus, and then, plugin Grafan to visually display the collected metrics on a series of dashboards. The high-level architecture of this approach will look like the following:

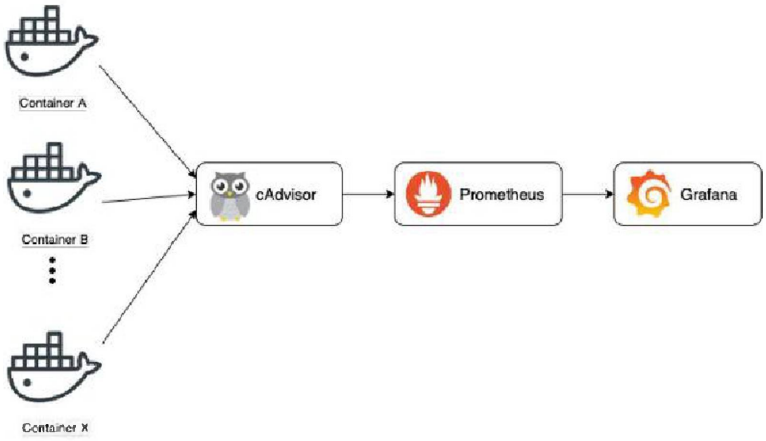


Figure 11.4: *Container monitoring architecture*

Monitoring Docker metrics with cAdvisor

To be able to monitor Docker container metrics with cAdvisor, proceed as follows:

1. Create a configuration file for Prometheus instructing it to scrape metrics from cAdvisor. For this:

a. Create a folder **prometheus** in your source folder for *this chapter* that is in **../src/ch11**

b. In this new folder, create a file called **prometheus.yml** and add the following content to it:

```
scrape_configs:
- job_name: cadvisor
  scrape_interval: 5s
  static_configs:
  - targets: ['cadvisor:8080']
```

2. Create a file called **docker-compose.yml** in the same folder as the previous one and add the following content to it:

```
version: '3.2'
services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    ports:
    - 9090:9090
    command:
    - --config.file=/etc/prometheus/prometheus.yml
    volumes:
    - ./prometheus:/etc/prometheus/prometheus:ro
    depends_on:
    - cadvisor
  cadvisor:
    image: gcr.io/cadvisor/cadvisor:v0.45.0
    container_name: cadvisor
    ports:
    - 8080:8080
    volumes:
    - /:/rootfs:ro
    - /var/run:/var/run:rw
    - /sys:/sys:ro
    - /var/lib/docker/:/var/lib/docker:ro
```

a. Create a folder **prometheus** in your source folder for *this chapter* that is in **../src/ch11**

b. In this new folder, create a file called **prometheus.yml** and add the following content to it:

```
scrape_configs:
- job_name: cadvisor
  scrape_interval: 5s
  static_configs:
  - targets: ['cadvisor:8080']
```

2. Create a file called **docker-compose.yml** in the same folder as the previous one and add the following content to it:

```
version: '3.2'
services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    ports:
    - 9090:9090
    command:
    - --config.file=/etc/prometheus/prometheus.yml
    volumes:
    - ./prometheus:/etc/prometheus/prometheus:ro
    depends_on:
    - cadvisor
  cadvisor:
    image: gcr.io/cadvisor/cadvisor:v0.45.0
    container_name: cadvisor
    ports:
    - 8080:8080
    volumes:
    - /:/rootfs:ro
    - /var/run:/var/run:rw
    - /sys:/sys:ro
    - /var/lib/docker/:/var/lib/docker:ro
```

```
    depends_on:
    - redis
  redis:
    image: redis:latest
    container_name: redis
    ports:
    - 6379:6379
```

The previous compose file instructs Docker to run the following three services:

a. The **prometheus** service uses the local **prometheus/prometheus.yml** configuration file that is imported into the container via the volumes parameter.

b. The **cadvisor** service exposes port **8080**, which is the default port for cAdvisor metrics, and relies on a variety of local volumes (**/, /var/run**, and so on). Note that we took an explicit version of the **cadvisor** image (**v0.45.0**) because earlier versions refused to run on a Mac with the M1 processor.

c. The **redis** service is a standard Redis server. **cAdvisor** will gather container metrics from this container automatically without any further configuration needed.

3. Run the application with the following:

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

First, the three images needed are pulled down from the Docker hub by the previous command. This may take a moment. Finally, in the output, you should see the following:

```
⠿ Network ch11_default  Created                    0.0s
⠿ Container redis       Started                    0.9s
⠿ Container cadvisor    Started                    0.9s
⠿ Container prometheus  Started                    1.1s
```

4. Now you can also check on the logs with the following:

```
docker compose logs prometheus
```

In the output produced by the previous command, you should see something like the following, confirming to you that Prometheus is up and running:

```
    depends_on:
    - redis
  redis:
    image: redis:latest
    container_name: redis
    ports:
    - 6379:6379
```

The previous compose file instructs Docker to run the following three services:

a. The **prometheus** service uses the local **prometheus/prometheus.yml** configuration file that is imported into the container via the volumes parameter.

b. The **cadvisor** service exposes port **8080**, which is the default port for cAdvisor metrics, and relies on a variety of local volumes (**/, /var/run**, and so on). Note that we took an explicit version of the **cadvisor** image (**v0.45.0**) because earlier versions refused to run on a Mac with the M1 processor.

c. The **redis** service is a standard Redis server. **cAdvisor** will gather container metrics from this container automatically without any further configuration needed.

3. Run the application with the following:

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

First, the three images needed are pulled down from the Docker hub by the previous command. This may take a moment. Finally, in the output, you should see the following:

```
⠿ Network ch11_default  Created                    0.0s
⠿ Container redis       Started                    0.9s
⠿ Container cadvisor    Started                    0.9s
⠿ Container prometheus  Started                    1.1s
```

4. Now you can also check on the logs with the following:

```
docker compose logs prometheus
```

In the output produced by the previous command, you should see something like the following, confirming to you that Prometheus is up and running:

```
prometheus  | ts=2022-09-04T09:31:59.390Z caller=main.go:897
level=info msg="Server is ready to receive web requests."
```

5. Verify that the three services are running with the following:

```
$ docker compose ps
NAME         COMMAND                  SERVICE      STATUS          PORTS
cadvisor     "/usr/bin/cadvisor -…"   cadvisor     running (healthy)
prometheus   "/bin/prometheus --c…"   prometheus   running         ...
redis        "docker-entrypoint.s…"   redis        running         ...
```

   You may have to wait a bit until the `cadvisor` service is switching from starting to healthy.

6. Open a new browser tab and navigate to the URL **http://localhost:8080** to open a WEB UI for cAdvisor:

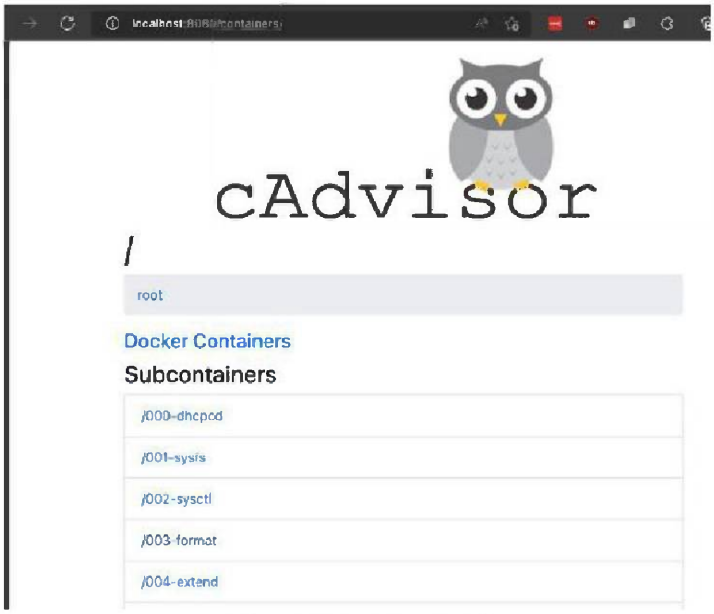


Figure 11.5: cAdvisor Web UI

7. To explore the container metrics in Prometheus, open another browser tab and navigate to `http://localhost:9090`, and you will find yourself in the Prometheus expression browser:

```
prometheus  | ts=2022-09-04T09:31:59.390Z caller=main.go:897
level=info msg="Server is ready to receive web requests."
```

5. Verify that the three services are running with the following:

```
$ docker compose ps
NAME         COMMAND                  SERVICE      STATUS          PORTS
cadvisor     "/usr/bin/cadvisor -…"   cadvisor     running (healthy)
prometheus   "/bin/prometheus --c…"   prometheus   running         ...
redis        "docker-entrypoint.s…"   redis        running         ...
```

   You may have to wait a bit until the `cadvisor` service is switching from starting to healthy.

6. Open a new browser tab and navigate to the URL **http://localhost:8080** to open a WEB UI for cAdvisor:

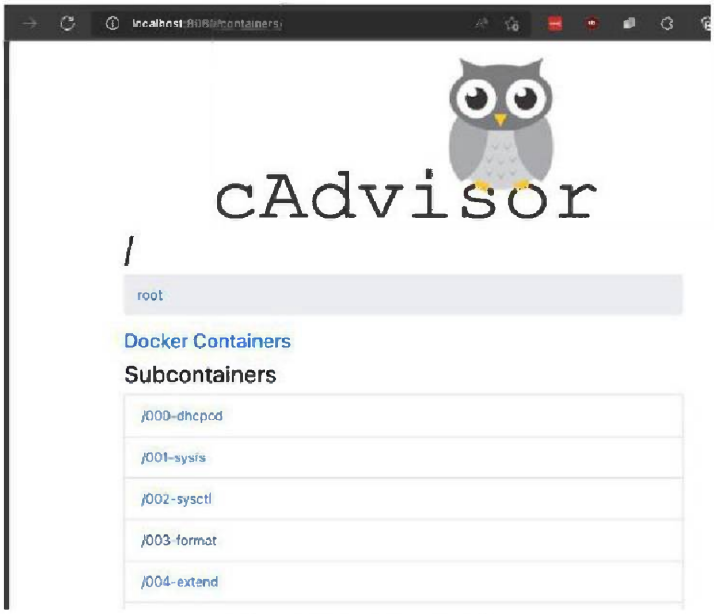


Figure 11.5: cAdvisor Web UI

7. To explore the container metrics in Prometheus, open another browser tab and navigate to `http://localhost:9090`, and you will find yourself in the Prometheus expression browser:

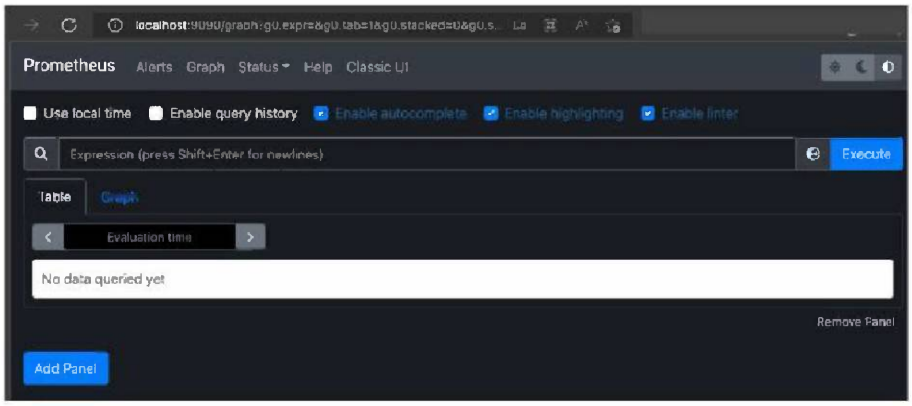


Figure 11.6: Prometheus expression browser

In the query text field, enter something like **`container_start_time_seconds`** to get the value for the startup time of all containers. To refine the query and only get the value for the Redis container, enter this: **`container_start_time_seconds{name="redis"}`**. Note that in the query text field, you get IntelliSense, which is convenient, when you do not remember all the details of a command and its parameters:



Figure 11.7: Intellisense for the Prometheus expression browser

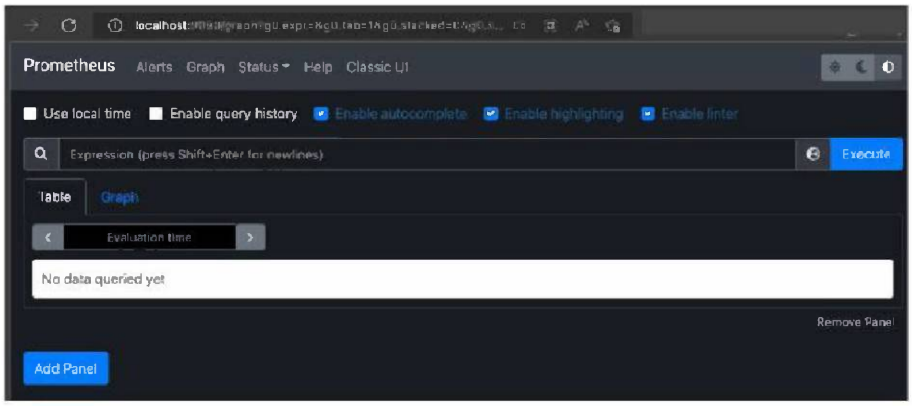


Figure 11.6: Prometheus expression browser

In the query text field, enter something like **`container_start_time_seconds`** to get the value for the startup time of all containers. To refine the query and only get the value for the Redis container, enter this: **`container_start_time_seconds{name="redis"}`**. Note that in the query text field, you get IntelliSense, which is convenient, when you do not remember all the details of a command and its parameters:



Figure 11.7: Intellisense for the Prometheus expression browser

We first started without Grafana and only showed the flow of data up until Prometheus. Now it is time to use Grafana to display the data visually.

Monitoring application key using Grafana

Now, let us add Grafana to the stack and show how to use it. To make the whole thing a bit more interesting, we are also adding **Node Exporter** to the picture. The Prometheus Node Exporter exposes a wide variety of hardware and kernel metrics.

1. Create a new file called **docker-compose-grafana.yml** as a sibling to the existing compose file and add the following code:

```
version: '3.2'
volumes:
    prometheus_data: {}
    grafana_data: {}

services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    volumes:
      - ./prometheus:/etc/prometheus
      - prometheus_data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/etc/prometheus/console_libraries'
      - '--web.console.templates=/etc/prometheus/consoles'
      - '--storage.tsdb.retention.time=200h'
      - '--web.enable-lifecycle'
    ports:
      - 9090:9090

  nodeexporter:
    image: prom/node-exporter:v1.3.1
    container_name: nodeexporter
```

We first started without Grafana and only showed the flow of data up until Prometheus. Now it is time to use Grafana to display the data visually.

Monitoring application key using Grafana

Now, let us add Grafana to the stack and show how to use it. To make the whole thing a bit more interesting, we are also adding **Node Exporter** to the picture. The Prometheus Node Exporter exposes a wide variety of hardware and kernel metrics.

1. Create a new file called **docker-compose-grafana.yml** as a sibling to the existing compose file and add the following code:

```
version: '3.2'
volumes:
    prometheus_data: {}
    grafana_data: {}

services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    volumes:
      - ./prometheus:/etc/prometheus
      - prometheus_data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
     - '--web.console.libraries=/etc/prometheus/console_libraries'
      - '--web.console.templates=/etc/prometheus/consoles'
      - '--storage.tsdb.retention.time=200h'
      - '--web.enable-lifecycle'
    ports:
      - 9090:9090

  nodeexporter:
    image: prom/node-exporter:v1.3.1
    container_name: nodeexporter
```

```
    volumes:
      - /proc:/host/proc:ro
      - /sys:/host/sys:ro
      - /:/rootfs:ro
    command:
      - '--path.procfs=/host/proc'
      - '--path.rootfs=/rootfs'
      - '--path.sysfs=/host/sys'
      -   '--collector.filesystem.mount-points-exclude=^/
(sys|proc|dev|host|etc)($$|/)'
    ports:
      - 9100:9100

  cadvisor:
    image: gcr.io/cadvisor/cadvisor:v0.45.0
    container_name: cadvisor
    privileged: true
    devices:
      - /dev/kmsg:/dev/kmsg
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:ro
      - /sys:/sys:ro
      - /var/lib/docker:/var/lib/docker:ro
      # - /cgroup:/cgroup:ro #doesn't work on MacOS only for Linux
    ports:
      - 8080:8080

  grafana:
    image: grafana/grafana:9.0.5
    container_name: grafana
    volumes:
      - grafana_data:/var/lib/grafana
    - ./grafana/provisioning/dashboards:/etc/grafana/provisioning/
dashboards
```

```
    volumes:
      - /proc:/host/proc:ro
      - /sys:/host/sys:ro
      - /:/rootfs:ro
    command:
      - '--path.procfs=/host/proc'
      - '--path.rootfs=/rootfs'
      - '--path.sysfs=/host/sys'
      -   '--collector.filesystem.mount-points-exclude=^/
(sys|proc|dev|host|etc)($$|/)'
    ports:
      - 9100:9100

  cadvisor:
    image: gcr.io/cadvisor/cadvisor:v0.45.0
    container_name: cadvisor
    privileged: true
    devices:
      - /dev/kmsg:/dev/kmsg
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:ro
      - /sys:/sys:ro
      - /var/lib/docker:/var/lib/docker:ro
      # - /cgroup:/cgroup:ro #doesn't work on MacOS only for Linux
    ports:
      - 8080:8080

  grafana:
    image: grafana/grafana:9.0.5
    container_name: grafana
    volumes:
      - grafana_data:/var/lib/grafana
    - ./grafana/provisioning/dashboards:/etc/grafana/provisioning/
dashboards
```

```
    - ./grafana/provisioning/datasources:/etc/grafana/provisioning/
datasources
    ports:
      - 3000:3000
```

2. We need to adjust the Prometheus configuration file so it has the following content. Note how we have now three scraping tasks compared to before when we only had a single one. Prometheus is instructed to scrape metrics from Node Exporter, cAdvisor, and even from itself:

```
global:
  scrape_interval:     15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: 'nodeexporter'
    scrape_interval: 5s
    static_configs:
      - targets: ['nodeexporter:9100']

  - job_name: 'cadvisor'
    scrape_interval: 5s
    static_configs:
      - targets: ['cadvisor:8080']

  - job_name: 'prometheus'
    scrape_interval: 10s
    static_configs:
      - targets: ['localhost:9090']
```

3. We now need to configure Grafana and tell it from where to source the data. We do this by the following:

    a. Creating a folder **grafana/provisioning** as a sibling to the **prometheus** folder.

    b. Inside the provisioning folder, we create a folder **datasources**.

    c. Inside the **datasources** folder, create a file **datasource.yml** with the following content:

```
    - ./grafana/provisioning/datasources:/etc/grafana/provisioning/
datasources
    ports:
      - 3000:3000
```

2. We need to adjust the Prometheus configuration file so it has the following content. Note how we have now three scraping tasks compared to before when we only had a single one. Prometheus is instructed to scrape metrics from Node Exporter, cAdvisor, and even from itself:

```
global:
  scrape_interval:     15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: 'nodeexporter'
    scrape_interval: 5s
    static_configs:
      - targets: ['nodeexporter:9100']

  - job_name: 'cadvisor'
    scrape_interval: 5s
    static_configs:
      - targets: ['cadvisor:8080']

  - job_name: 'prometheus'
    scrape_interval: 10s
    static_configs:
      - targets: ['localhost:9090']
```

3. We now need to configure Grafana and tell it from where to source the data. We do this by the following:

    a. Creating a folder **grafana/provisioning** as a sibling to the **prometheus** folder.

    b. Inside the provisioning folder, we create a folder **datasources**.

    c. Inside the **datasources** folder, create a file **datasource.yml** with the following content:

```
apiVersion: 1

datasources:
  - name: Prometheus
    type: prometheus
    access: proxy
    orgId: 1
    url: http://prometheus:9090
    basicAuth: false
    isDefault: true
    editable: true
```

4. Next, we need to define some pre-canned dashboards for Grafana, to make our lives a bit easier. Otherwise, we would have to define them ourselves. We can do this by mapping json files containing the dashboard definition into the Grafana container. Since these JSON files are pretty long, please proceed as follows:
    a. Inside the **grafana/provisioning** folder, create a folder **dashboards**.
    b. Into this folder, copy the four files found, as shown in the following figure:



Figure 11.8: Files containing predefine Grafana dashboards

5. With this, we should be ready. To run the stack with Grafana.
    a. Shut down and clean-up the running stack with the following:

```
$ docker compose down -v
```

```
apiVersion: 1

datasources:
  - name: Prometheus
    type: prometheus
    access: proxy
    orgId: 1
    url: http://prometheus:9090
    basicAuth: false
    isDefault: true
    editable: true
```

4. Next, we need to define some pre-canned dashboards for Grafana, to make our lives a bit easier. Otherwise, we would have to define them ourselves. We can do this by mapping json files containing the dashboard definition into the Grafana container. Since these JSON files are pretty long, please proceed as follows:
    a. Inside the **grafana/provisioning** folder, create a folder **dashboards**.
    b. Into this folder, copy the four files found, as shown in the following figure:



Figure 11.8: Files containing predefine Grafana dashboards

5. With this, we should be ready. To run the stack with Grafana.
    a. Shut down and clean-up the running stack with the following:

```
$ docker compose down -v
```

b. Run the new stack with the following:

```
$ docker compose -f docker-compose.yml up -d
```

6. We can open another browser tab and navigate to Grafana **`http://localhost:3000`**. Login with username **`admin`** and password **`admin`**. When asked to change the password, just re-use the same password **`admin`** again, for simplicity.

7. In Grafana, select **Dashboards** and select one of the predefined dashboards:

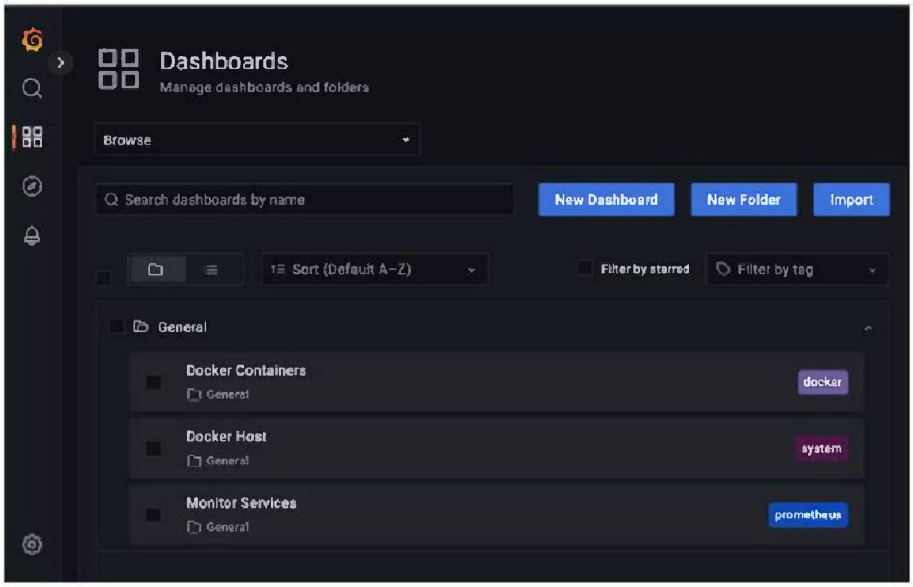


Figure 11.9: Selection of Grafana dashboards

8. You should see something similar to the following picture. As you can see, we are getting metrics from the four services in our stack, cAdvisor, node exporter, Prometheus, and Grafana:

b. Run the new stack with the following:

```
$ docker compose -f docker-compose.yml up -d
```

6. We can open another browser tab and navigate to Grafana **`http://localhost:3000`**. Login with username **`admin`** and password **`admin`**. When asked to change the password, just re-use the same password **`admin`** again, for simplicity.

7. In Grafana, select **Dashboards** and select one of the predefined dashboards:

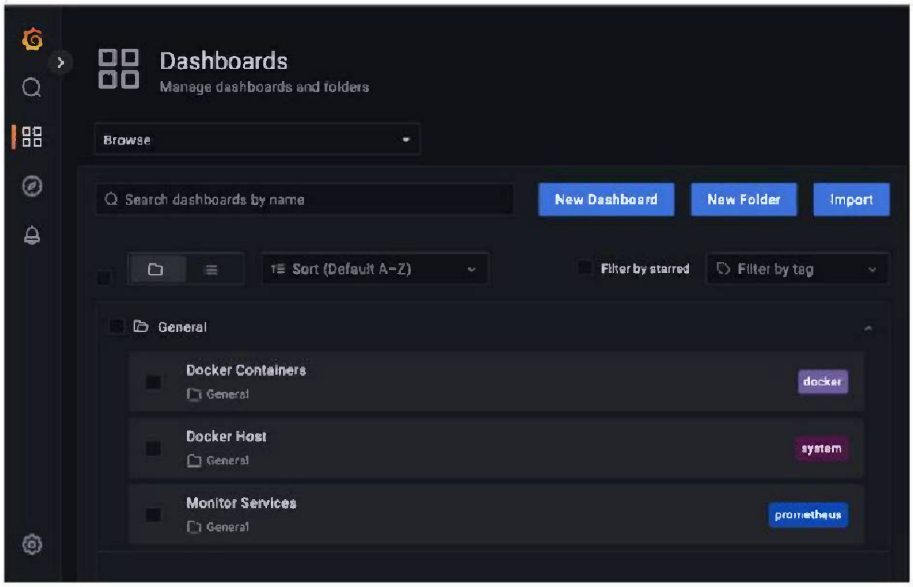


Figure 11.9: Selection of Grafana dashboards

8. You should see something similar to the following picture. As you can see, we are getting metrics from the four services in our stack, cAdvisor, node exporter, Prometheus, and Grafana:



***Figure 11.10:** Container dashboard in Grafana*

Wonderful, we have shown how to collect metrics from our Docker host and the containers running on it via cAdvisor, node exporter, and Prometheus and used Grafana to display key values in a nice graphical way that is pleasing to look at and informative at the same time. We have the most important information right at our fingertips.

Defining alerts

Just collecting and displaying metrics, be they infrastructure or business-relevant metrics, is not enough. We need to identify key metrics that really define the health of the system and then define **Service Level Objectives (SLOs)** and **Service Level Agreements (SLAs)** for those metrics. We then define rules on how often or how long a measured metric can exceed the corresponding SLO or SLA. If such a rule is violated, we trigger an alert.

To get a feeling for this, let us define a couple of candidates for alerts. One sample is a business-relevant metric, and the other is a non-functional, that is, a system-level metric. Can you tell which is which?



***Figure 11.10:** Container dashboard in Grafana*

Wonderful, we have shown how to collect metrics from our Docker host and the containers running on it via cAdvisor, node exporter, and Prometheus and used Grafana to display key values in a nice graphical way that is pleasing to look at and informative at the same time. We have the most important information right at our fingertips.

Defining alerts

Just collecting and displaying metrics, be they infrastructure or business-relevant metrics, is not enough. We need to identify key metrics that really define the health of the system and then define **Service Level Objectives (SLOs)** and **Service Level Agreements (SLAs)** for those metrics. We then define rules on how often or how long a measured metric can exceed the corresponding SLO or SLA. If such a rule is violated, we trigger an alert.

To get a feeling for this, let us define a couple of candidates for alerts. One sample is a business-relevant metric, and the other is a non-functional, that is, a system-level metric. Can you tell which is which?

- In an application offering life insurance, we could define the time it takes to calculate a quote for a customer interested in a quote as a key metric. The SLA for this metric could then be that 99% of all requests for a quote need to be faster than 50 ms. The slowest request cannot take more than 1,000 ms. A rule for an alert could then be that if the SLA is violated more than three times in one hour, an alert should be triggered.
- In a banking application, we identify the percentage of total CPU used as a metric. The **service level objective (SLO)** could be that the percentage should not exceed 99%. The rule could be that if the CPU percentage exceeds 99% for more than 50% of the time during an interval of 1 minute, an alert should be triggered.

Yes, correct, the former is a business metric, and the latter an infrastructure metric.

Alerts can then be routed to identified target persons such as system reliability engineers or developers by various means such as e-mail, text message, automated phone call, slack message, acoustic alarm, optical alarm, or more.

Once we have defined and wired such alarms, service personnel does not have to actively monitor the system anymore and can instead perform other tasks. They have the guarantee to be notified if something important or exceptional happens where they are required to react.

Now let us discuss a hot topic, issues happening on a production system. We want to be able to pinpoint the issue's root cause quickly.

Finding the root cause of an issue

If you are responsible for an application running in production, there will come a time, and it is unavoidable, where you are faced with an issue. Issues are often hindering the production system to perform optimally, or in the worst case, it will cause the system to completely go out of order. In such a situation, we need to react swiftly and do everything possible to get the system up and running again as quickly as possible. But to be able to do so, we need to first identify the root cause of the problem. Since the application is running in production, we do not have direct access to it. We cannot just log in to the servers on which the application is running and do local troubleshooting. For example, it is impossible to inspect the filesystem of the affected server or browse through log files that are stored on the affected server. We need to find other ways how to pinpoint the root cause.

When troubleshooting a problem, we are moving in a multi-dimensional problem space. The dimensions of this space are the factors that can vary in the system, such as the servers on which the application is running. If we are running our application on a Kubernetes cluster, then we are typically talking about many servers to which parts of the application can be deployed and run. Other dimensions are :

- In an application offering life insurance, we could define the time it takes to calculate a quote for a customer interested in a quote as a key metric. The SLA for this metric could then be that 99% of all requests for a quote need to be faster than 50 ms. The slowest request cannot take more than 1,000 ms. A rule for an alert could then be that if the SLA is violated more than three times in one hour, an alert should be triggered.
- In a banking application, we identify the percentage of total CPU used as a metric. The **service level objective (SLO)** could be that the percentage should not exceed 99%. The rule could be that if the CPU percentage exceeds 99% for more than 50% of the time during an interval of 1 minute, an alert should be triggered.

Yes, correct, the former is a business metric, and the latter an infrastructure metric.

Alerts can then be routed to identified target persons such as system reliability engineers or developers by various means such as e-mail, text message, automated phone call, slack message, acoustic alarm, optical alarm, or more.

Once we have defined and wired such alarms, service personnel does not have to actively monitor the system anymore and can instead perform other tasks. They have the guarantee to be notified if something important or exceptional happens where they are required to react.

Now let us discuss a hot topic, issues happening on a production system. We want to be able to pinpoint the issue's root cause quickly.

Finding the root cause of an issue

If you are responsible for an application running in production, there will come a time, and it is unavoidable, where you are faced with an issue. Issues are often hindering the production system to perform optimally, or in the worst case, it will cause the system to completely go out of order. In such a situation, we need to react swiftly and do everything possible to get the system up and running again as quickly as possible. But to be able to do so, we need to first identify the root cause of the problem. Since the application is running in production, we do not have direct access to it. We cannot just log in to the servers on which the application is running and do local troubleshooting. For example, it is impossible to inspect the filesystem of the affected server or browse through log files that are stored on the affected server. We need to find other ways how to pinpoint the root cause.

When troubleshooting a problem, we are moving in a multi-dimensional problem space. The dimensions of this space are the factors that can vary in the system, such as the servers on which the application is running. If we are running our application on a Kubernetes cluster, then we are typically talking about many servers to which parts of the application can be deployed and run. Other dimensions are :

- **Category of issue:** An issue can show up in one of the various categories. It can be a security issue, a performance issue, a resource problem such as lack of RAM or disk space, a network issue, and more.
- **Kind of issue:** It is the issue caused by an infrastructure problem, an application runtime problem, or just by a plain old bug?
- **Type of service:** If our application is consisting of multiple different services, then the dimension is which of those services shows the problem.
- **Time of issue:** One of the most important dimensions of our problem space is the time axis. When exactly did the issue happen?
- **Container or pod**: In a containerized application, we can identify the exact container or pod where the service ran that caused the problem

To quickly pinpoint the root cause, we need to try hard to collapse as many dimensions of the problem space as possible. When we talk about collapsing, we really mean that we need to try to fix a single point or a small interval on a given axis. Collapsing the time dimensions, for example, can be achieved by identifying or pointing out the exact date and time when the issue was first observed. Identifying the server on which the problem was observed first means that we just collapsed another dimension, the server dimension.

Now assuming that we have provisioned our system in a way that it centrally collects all the container and infrastructure logs. Earlier, we discussed how we can do this. In this case, we can now use queries that limit the data returned with filters that correspond to the collapsed dimensions. For example, have a statement that only returns us logs from the server we have identified as the one where the problem was observed. We can add an additional filter that limits the time range of the log entries returned if we know the approximate time of the issue. We can add yet another filter that only returns logs from service A if this is the one, we have pinpointed as the origin of the problem.

The more filters we can add to the query, the less items the query returns and the faster we can come to a conclusion. Now, we need to find suspicious patterns in the remaining data.

If there is still too much log information left that we need to browse through, we can try to use the recorded metrics in the form of dashboards with trend lines as another way of guiding us to the problem spot.

Identifying the root cause of a production issue can be quite difficult, and it requires some experience and an investigative mind to come to quick results. And here, time is of the essence. If we are facing a production down issue, then every second counts, and we need to be very strategic in our approach.

- **Category of issue:** An issue can show up in one of the various categories. It can be a security issue, a performance issue, a resource problem such as lack of RAM or disk space, a network issue, and more.
- **Kind of issue:** It is the issue caused by an infrastructure problem, an application runtime problem, or just by a plain old bug?
- **Type of service:** If our application is consisting of multiple different services, then the dimension is which of those services shows the problem.
- **Time of issue:** One of the most important dimensions of our problem space is the time axis. When exactly did the issue happen?
- **Container or pod**: In a containerized application, we can identify the exact container or pod where the service ran that caused the problem

To quickly pinpoint the root cause, we need to try hard to collapse as many dimensions of the problem space as possible. When we talk about collapsing, we really mean that we need to try to fix a single point or a small interval on a given axis. Collapsing the time dimensions, for example, can be achieved by identifying or pointing out the exact date and time when the issue was first observed. Identifying the server on which the problem was observed first means that we just collapsed another dimension, the server dimension.

Now assuming that we have provisioned our system in a way that it centrally collects all the container and infrastructure logs. Earlier, we discussed how we can do this. In this case, we can now use queries that limit the data returned with filters that correspond to the collapsed dimensions. For example, have a statement that only returns us logs from the server we have identified as the one where the problem was observed. We can add an additional filter that limits the time range of the log entries returned if we know the approximate time of the issue. We can add yet another filter that only returns logs from service A if this is the one, we have pinpointed as the origin of the problem.

The more filters we can add to the query, the less items the query returns and the faster we can come to a conclusion. Now, we need to find suspicious patterns in the remaining data.

If there is still too much log information left that we need to browse through, we can try to use the recorded metrics in the form of dashboards with trend lines as another way of guiding us to the problem spot.

Identifying the root cause of a production issue can be quite difficult, and it requires some experience and an investigative mind to come to quick results. And here, time is of the essence. If we are facing a production down issue, then every second counts, and we need to be very strategic in our approach.

Conclusion

In this chapter, we have shown you how to scrape the logs of all the containers that are part of your application and collect them centrally. We have shown how to display key metrics extracted from those logs in Grafana, and finally, we have learned how to define alerts that are triggered when a key metric exceeds the predefined tolerance band or limits. All this we did to make it easier to localize the root cause of a production issue more quickly.

Job interview questions

You finally got an invitation to a job interview for the role as **system reliability engineer (SRE)** at a very well-established firm in a town close to you. The interviewer is asking you the following questions:

1. In our company, we are not yet collecting any metrics and monitoring them, nor have we defined any alerts yet. Tell us, what are we missing out?
2. Given the situation that we run our mission-critical applications in the cloud and on a hosted Kubernetes cluster, what would be your steps to establish monitoring and alerting? Describe the most important steps on a high level and order them meaningfully.
3. Finding an issue in production is quite challenging for our engineers. Tell us your strategy to quickly pinpoint the root cause of the issue.
4. What are you doing after a sever production issue has been fixed and the system is up and running again?

Possible answers to the job interview questions

Here are possible answers to the previous questions:

1. Not collecting metrics from the application and not having any monitoring and alerting defined based on the collected metrics and logs is like flying blind. Nobody wants to sit in a plane where the pilot is blindfolded. We are totally ignorant about what is really happening inside our application. We do not know and can only guess if the application is healthy from a technical and from business perspective. If we monitor our system but do not define alerts, then we waste too much resources in keeping our system up and running and in a healthy state. SREs need to permanently stare at the dashboards, although most of the time, nothing remarkable happens. It is better to have the system alert the SREs when something noteworthy happens. Then, the SREs can do better things in the meantime instead of sitting around.

Conclusion

In this chapter, we have shown you how to scrape the logs of all the containers that are part of your application and collect them centrally. We have shown how to display key metrics extracted from those logs in Grafana, and finally, we have learned how to define alerts that are triggered when a key metric exceeds the predefined tolerance band or limits. All this we did to make it easier to localize the root cause of a production issue more quickly.

Job interview questions

You finally got an invitation to a job interview for the role as **system reliability engineer (SRE)** at a very well-established firm in a town close to you. The interviewer is asking you the following questions:

1. In our company, we are not yet collecting any metrics and monitoring them, nor have we defined any alerts yet. Tell us, what are we missing out?
2. Given the situation that we run our mission-critical applications in the cloud and on a hosted Kubernetes cluster, what would be your steps to establish monitoring and alerting? Describe the most important steps on a high level and order them meaningfully.
3. Finding an issue in production is quite challenging for our engineers. Tell us your strategy to quickly pinpoint the root cause of the issue.
4. What are you doing after a sever production issue has been fixed and the system is up and running again?

Possible answers to the job interview questions

Here are possible answers to the previous questions:

1. Not collecting metrics from the application and not having any monitoring and alerting defined based on the collected metrics and logs is like flying blind. Nobody wants to sit in a plane where the pilot is blindfolded. We are totally ignorant about what is really happening inside our application. We do not know and can only guess if the application is healthy from a technical and from business perspective. If we monitor our system but do not define alerts, then we waste too much resources in keeping our system up and running and in a healthy state. SREs need to permanently stare at the dashboards, although most of the time, nothing remarkable happens. It is better to have the system alert the SREs when something noteworthy happens. Then, the SREs can do better things in the meantime instead of sitting around.

2. The most important thing is to ship all logs to a central location, be it log information from the cluster nodes, the Kubernetes cluster, the container runtime on each node and the logs from all resources running on Kubernetes, and finally, the application logs produced by applications running inside the containers in the pods. Next, we need to define key metrics (functional and non-functional) that we want to monitor and trigger alerts if the values of those key metrics exceed predefined values. We then display those KPIs on dashboards in a tool such as Kibana or Grafana. We define rules for each alert and wire the alert with a system such as PagerDuty, such as key stakeholders such as operations engineers, DevOps engineers, or developers get notified via Slack message, text message, phone call, or similar, and can take action. We can then also start instrumenting the code such as that it produces additional, mostly business relevant, that is, functional metrics. These metrics we can then scrape with a tool such as Prometheus and use Grafana to display those functional metrics on dashboards in the form of counters, gauges, and trends. Once again, we can select key functional metric and define rules that will trigger alerts if violated.

3. Pinpointing the root cause of a production issue is challenging due to the fact that we are dealing with a multi-dimensional problem space. There are many factors that can contribute to the problem. Each of those factors we call a dimension. The time axis, for example, is a dimension. In a service-oriented architecture, the service is a dimension. On a cluster, the nodes are dimension, and so on. The goal is to reduce the problem space to a small number of dimensions. How can we do that? We need to try to fix a point on as many dimensions as possible. That is, we can, for example, pinpoint the exact time the issue happened. With this, the time dimension collapses to a point, and with it, our problem space. We can locate the cluster node on which the problem happened, and another dimension collapses. Then we can identify the exact service where the problem happened, and so on. Once we have collapsed most dimensions, we can start narrowing the problem on the remaining dimensions until we are sure to get the root cause.

4. It is crucial to perform a post-mortem meeting as soon as possible after the production issue has been fixed. The goal of the post-mortem meeting is to point out what exactly went wrong without trying to blame somebody. We need to describe the steps we did to fix the problem. We need to take action to make sure the same issue cannot happen again. And finally, we need to make sure that, if we did some manual fixes during the production outage, those manual steps are automated and carefully tested. An example could be that we had to manually fix a problem with the database by, e.g., adding a missing file to a table or adding a missing index that is needed to get scalable and fast query responses.

2. The most important thing is to ship all logs to a central location, be it log information from the cluster nodes, the Kubernetes cluster, the container runtime on each node and the logs from all resources running on Kubernetes, and finally, the application logs produced by applications running inside the containers in the pods. Next, we need to define key metrics (functional and non-functional) that we want to monitor and trigger alerts if the values of those key metrics exceed predefined values. We then display those KPIs on dashboards in a tool such as Kibana or Grafana. We define rules for each alert and wire the alert with a system such as PagerDuty, such as key stakeholders such as operations engineers, DevOps engineers, or developers get notified via Slack message, text message, phone call, or similar, and can take action. We can then also start instrumenting the code such as that it produces additional, mostly business relevant, that is, functional metrics. These metrics we can then scrape with a tool such as Prometheus and use Grafana to display those functional metrics on dashboards in the form of counters, gauges, and trends. Once again, we can select key functional metric and define rules that will trigger alerts if violated.

3. Pinpointing the root cause of a production issue is challenging due to the fact that we are dealing with a multi-dimensional problem space. There are many factors that can contribute to the problem. Each of those factors we call a dimension. The time axis, for example, is a dimension. In a service-oriented architecture, the service is a dimension. On a cluster, the nodes are dimension, and so on. The goal is to reduce the problem space to a small number of dimensions. How can we do that? We need to try to fix a point on as many dimensions as possible. That is, we can, for example, pinpoint the exact time the issue happened. With this, the time dimension collapses to a point, and with it, our problem space. We can locate the cluster node on which the problem happened, and another dimension collapses. Then we can identify the exact service where the problem happened, and so on. Once we have collapsed most dimensions, we can start narrowing the problem on the remaining dimensions until we are sure to get the root cause.

4. It is crucial to perform a post-mortem meeting as soon as possible after the production issue has been fixed. The goal of the post-mortem meeting is to point out what exactly went wrong without trying to blame somebody. We need to describe the steps we did to fix the problem. We need to take action to make sure the same issue cannot happen again. And finally, we need to make sure that, if we did some manual fixes during the production outage, those manual steps are automated and carefully tested. An example could be that we had to manually fix a problem with the database by, e.g., adding a missing file to a table or adding a missing index that is needed to get scalable and fast query responses.

Questions

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

1. What type of information would we typically find in a container log?
2. What is the role of Prometheus, and how does it work?
3. Why is it recommended to centrally collect the logs of all containers?
4. What is the purpose of a tool such as Grafana?
5. What do we need to define an alert for our application running in production?
6. What do we mean when we talk about a multi-dimensional problem space when we are dealing with a production issue?

Answers

Here are the answers to the previously mentioned questions:

1. In a container log, we find information of different levels and importance. Generally, we categorize the information into the following five levels.
    a. **Debug**: This is information that helps to debug an application. It is very technical information, and normally, only developers familiar with the code can understand its meaning.
    b. **Info**: This is—as the name points out—information about what is happening in the application. No action is required.
    c. **Warning**: This is information that indicates that something happened that could potentially be a problem but is not critical. It is like the warning light in a car telling you that your gasoline tank will soon be empty. You do not have to panic but should take care of it eventually.
    d. **Error**: An error happened in the application. What just happened should not have happened. Often, a well-written application can recover from such an error. Nevertheless, the SREs should absolutely take note of it and try to solve the root cause of the error such as that it does not occur again.
    e. **Fatal**: something terrible happened. The application should fail fast in such a situation as these types of errors are not recoverable. The immediate reaction of the SREs is needed.
2. Prometheus is a tool that is used to collect logging information and metrics from different components of the system. These can be infrastructure-level components such as servers, VMs, containers, and network infrastructure. It

Questions

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

1. What type of information would we typically find in a container log?
2. What is the role of Prometheus, and how does it work?
3. Why is it recommended to centrally collect the logs of all containers?
4. What is the purpose of a tool such as Grafana?
5. What do we need to define an alert for our application running in production?
6. What do we mean when we talk about a multi-dimensional problem space when we are dealing with a production issue?

Answers

Here are the answers to the previously mentioned questions:

1. In a container log, we find information of different levels and importance. Generally, we categorize the information into the following five levels.
    a. **Debug**: This is information that helps to debug an application. It is very technical information, and normally, only developers familiar with the code can understand its meaning.
    b. **Info**: This is—as the name points out—information about what is happening in the application. No action is required.
    c. **Warning**: This is information that indicates that something happened that could potentially be a problem but is not critical. It is like the warning light in a car telling you that your gasoline tank will soon be empty. You do not have to panic but should take care of it eventually.
    d. **Error**: An error happened in the application. What just happened should not have happened. Often, a well-written application can recover from such an error. Nevertheless, the SREs should absolutely take note of it and try to solve the root cause of the error such as that it does not occur again.
    e. **Fatal**: something terrible happened. The application should fail fast in such a situation as these types of errors are not recoverable. The immediate reaction of the SREs is needed.
2. Prometheus is a tool that is used to collect logging information and metrics from different components of the system. These can be infrastructure-level components such as servers, VMs, containers, and network infrastructure. It

can also be application-level components such as a microservice instrumented by the developer such as that it produces logging information of defines application-level metrics such as counter, gauges, and more. Prometheus is actively scraping the information from components it knows (or has been configured for). It aggregates all the information centrally, such as that it can be used, e.g., for monitoring with a tool such as Grafana or to define alerts that integrate with tools such as PagerDuty.

3. We collect all container logs centrally because it makes finding the root cause of production issues so much simpler. If we leave the logs of each container on the container host, then we would have to access every node of our cluster to find interesting information. This approach would not scale, and it is insecure since we would have to permit access to the cluster to people.

   Also, if an attacker manages to compromise a node of a cluster and the container and other logs are only stored on the node, then the attacker could wipe out any traces he or she left during the attack. If the logs permanently are forwarded to a central location, the attacker cannot wipe out their traces.

4. Grafana can be used to display interesting metrics extracted from logs in a nice graphical form. A picture often says more than a thousand words. If we, e.g., see a trend line on a dashboard, we can immediately spot any unexpected or unwanted deviations from the norm, or we can recognize patterns that we would not be able to spot if just staring at a row of numbers.

5. We should define alerts in a system to avoid having to actively monitor the application(s), which is error-prone and boring to say the least. To be able to define a meaningful alert, we need the following ingredients:

   a. We need to define a **metric** that we want to measure, such as the memory consumption of a service or the number of policies sold per hour in an insurance application. The former is a non-functional metric, and the latter is a functional metric. Business people care about functional metrics, and technical people, such as SREs, care about non-functional metrics.

   b. We need to collect the metric over time in a central location.

   c. We need to define tolerance bands inside which the measured metric can move. If the current value of the metric exceeds the tolerance band, then we have a potential situation where we may want to alert somebody.

   d. We need a system such as Prometheus, where we can define a rule for alerts and integrate it with a system such as PagerDuty, which can be used to contact a responsible person such as an SRE through a pager, SMS, phone call, e-mail, or other means.

can also be application-level components such as a microservice instrumented by the developer such as that it produces logging information of defines application-level metrics such as counter, gauges, and more. Prometheus is actively scraping the information from components it knows (or has been configured for). It aggregates all the information centrally, such as that it can be used, e.g., for monitoring with a tool such as Grafana or to define alerts that integrate with tools such as PagerDuty.

3. We collect all container logs centrally because it makes finding the root cause of production issues so much simpler. If we leave the logs of each container on the container host, then we would have to access every node of our cluster to find interesting information. This approach would not scale, and it is insecure since we would have to permit access to the cluster to people.

   Also, if an attacker manages to compromise a node of a cluster and the container and other logs are only stored on the node, then the attacker could wipe out any traces he or she left during the attack. If the logs permanently are forwarded to a central location, the attacker cannot wipe out their traces.

4. Grafana can be used to display interesting metrics extracted from logs in a nice graphical form. A picture often says more than a thousand words. If we, e.g., see a trend line on a dashboard, we can immediately spot any unexpected or unwanted deviations from the norm, or we can recognize patterns that we would not be able to spot if just staring at a row of numbers.

5. We should define alerts in a system to avoid having to actively monitor the application(s), which is error-prone and boring to say the least. To be able to define a meaningful alert, we need the following ingredients:

   a. We need to define a **metric** that we want to measure, such as the memory consumption of a service or the number of policies sold per hour in an insurance application. The former is a non-functional metric, and the latter is a functional metric. Business people care about functional metrics, and technical people, such as SREs, care about non-functional metrics.

   b. We need to collect the metric over time in a central location.

   c. We need to define tolerance bands inside which the measured metric can move. If the current value of the metric exceeds the tolerance band, then we have a potential situation where we may want to alert somebody.

   d. We need a system such as Prometheus, where we can define a rule for alerts and integrate it with a system such as PagerDuty, which can be used to contact a responsible person such as an SRE through a pager, SMS, phone call, e-mail, or other means.

6. When dealing with a production incident, we need to pinpoint the root cause of the error as quickly as possible and fix it, so the production system can continue to run flawlessly in the shortest amount of time possible. When pinpointing the root cause of the issue, we say that we move in a multi-dimensional problem space. Every dimension of this space corresponds to one factor that is relevant to pinpoint the issue. Typical dimensions are:
    a. **Time**: When did the issue happen
    b. **Location**: Where did the issue happen (e.g., on which node of the Kubernetes cluster)
    c. **Service**: Which service of the many did show the malfunction
    d. **Type**: What is the type of the malfunction

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



6. When dealing with a production incident, we need to pinpoint the root cause of the error as quickly as possible and fix it, so the production system can continue to run flawlessly in the shortest amount of time possible. When pinpointing the root cause of the issue, we say that we move in a multi-dimensional problem space. Every dimension of this space corresponds to one factor that is relevant to pinpoint the issue. Typical dimensions are:
    a. **Time**: When did the issue happen
    b. **Location**: Where did the issue happen (e.g., on which node of the Kubernetes cluster)
    c. **Service**: Which service of the many did show the malfunction
    d. **Type**: What is the type of the malfunction

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 12
Enabling Zero Downtime Deployments

Introduction

In this chapter, we will learn 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.

We will be using VS Code, Docker Desktop with Kubernetes, and AWS-hosted Kubernetes as our main tools in this chapter.

Structure

In this chapter, we will discuss the following topics:

- Establishing a sound versioning strategy
- Using rolling updates and rollbacks
- Enabling blue-green deployments
- Realizing canary releases

Chapter 12
Enabling Zero Downtime Deployments

Introduction

In this chapter, we will learn 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.

We will be using VS Code, Docker Desktop with Kubernetes, and AWS-hosted Kubernetes as our main tools in this chapter.

Structure

In this chapter, we will discuss the following topics:

- Establishing a sound versioning strategy
- Using rolling updates and rollbacks
- Enabling blue-green deployments
- Realizing canary releases

Objectives

After reading this chapter, the reader will be able to apply a well-documented and popular version strategy to their application or application services. The reader will also be able to roll out updated versions of their application services that are not requiring any downtime of the application. Doing so, the reader will be able to choose between three rollout strategies, namely, rolling updates, blue-green deployments, and canary releases.

Let us start with the versioning strategy.

Establishing a sound versioning strategy

In any realistic application, whether or not it is a small Web shop or a highly distributed, mission-critical enterprise application, one needs to have a sound and well-thought-through strategy defining how one will manage updates of the functionality. In general, one can distinguish between three main categories of updates:

1. A new feature has been added to the application service
2. Some technical depth is removed from the service
3. A bug has been fixed

Although on the surface the preceding three categories look quite different, they all require that we ultimately have to deploy a new version of the application or application service into production.

Over time, many subsequent updates may be required. To keep track of the updates, one could introduce a version number, which is just a steadily increasing integer, starting from 1 and increasing by 1 on each new release. If we have several versions of the same service, we would automatically understand that the one version with the highest number is the most up-to-date version.

Alternatively, we could use a timestamp as a means of versioning our application service. The timestamp could be the combination of the date and time of the moment it is decided that the new version is ready to be released to production. Again, if we have several versions of the same service, then automatically, the one with the new most recent timestamp is the last available version.

This type of versioning will work well for smaller applications developed by a small group of developers. The moment our application grows and is composed of many interacting services, and the number of developers also grows, resulting in many teams, we need a more powerful way of versioning our services.

Objectives

After reading this chapter, the reader will be able to apply a well-documented and popular version strategy to their application or application services. The reader will also be able to roll out updated versions of their application services that are not requiring any downtime of the application. Doing so, the reader will be able to choose between three rollout strategies, namely, rolling updates, blue-green deployments, and canary releases.

Let us start with the versioning strategy.

Establishing a sound versioning strategy

In any realistic application, whether or not it is a small Web shop or a highly distributed, mission-critical enterprise application, one needs to have a sound and well-thought-through strategy defining how one will manage updates of the functionality. In general, one can distinguish between three main categories of updates:

1. A new feature has been added to the application service
2. Some technical depth is removed from the service
3. A bug has been fixed

Although on the surface the preceding three categories look quite different, they all require that we ultimately have to deploy a new version of the application or application service into production.

Over time, many subsequent updates may be required. To keep track of the updates, one could introduce a version number, which is just a steadily increasing integer, starting from 1 and increasing by 1 on each new release. If we have several versions of the same service, we would automatically understand that the one version with the highest number is the most up-to-date version.

Alternatively, we could use a timestamp as a means of versioning our application service. The timestamp could be the combination of the date and time of the moment it is decided that the new version is ready to be released to production. Again, if we have several versions of the same service, then automatically, the one with the new most recent timestamp is the last available version.

This type of versioning will work well for smaller applications developed by a small group of developers. The moment our application grows and is composed of many interacting services, and the number of developers also grows, resulting in many teams, we need a more powerful way of versioning our services.

In a scenario such as the one just described, we also want to make sure that each service has an independent release cycle. Team A, owning service 1 must be able to change their service more or less independently from team B, owning service 2. That said, it can well be that after some time, team A had built and released 20% more versions of their service than team B.

Luckily, there is a versioning system available to us all that has been well-adopted in the industry. The system is called semantic versioning. There is even a website (**https://semver.org**) where the interested reader can find more details about the concept.

In a nutshell, semantic versioning tells us that each new version of a service should be tagged with a version number than consist of three parts MAJOR, MINOR, and PATCH, separated by a period. This is a version looking like this: `MAJOR.MINOR.PATCH`

Each of the three is an integer, starting with zero. Given this schema, one should increment

- MAJOR version when one makes incompatible changes to the API of the service
- MINOR version when one adds functionality to the service in a backward-compatible manner
- PATCH version when one implements backward-compatible bug fixes

Now, I admit this is a bit abstract and not easy to understand. Let us try to elaborate on what the preceding means in practice.

If your team is tasked to fix a bug in the code of the service you own, and this fix does not add any new functionality to the service, nor does it change any of the public interfaces or contracts of your service, then you should only increase the PATCH version of the service, compared to the one that is the valid one. That is, if you run version **`1.2.13`** in production and have to fix such a bug, then the new version of the service, after the bug fix, should be **`1.2.14`**.

Now imagine a situation where your team implements a new feature in the service, but no update on the public interfaces or contracts of the service is needed; then you should update the MINOR version. As an example, given that the current version in production is **`1.2.14`**, then your new version would have to be tagged with **`1.3.0`**. Please note that the MINOR version has increased from 12 to 13, and at the same time, the PATCH version has been reset to zero.

Finally, assume that, once again, your team is tasked to implement yet another new feature. This time the team realizes that one of the public contracts or interfaces needs to be changed in a way that is breaking backwards compatibility. In this case, the team has to increase the MAJOR version. So, from say version **`1.3.2`**, your team

In a scenario such as the one just described, we also want to make sure that each service has an independent release cycle. Team A, owning service 1 must be able to change their service more or less independently from team B, owning service 2. That said, it can well be that after some time, team A had built and released 20% more versions of their service than team B.

Luckily, there is a versioning system available to us all that has been well-adopted in the industry. The system is called semantic versioning. There is even a website (**https://semver.org**) where the interested reader can find more details about the concept.

In a nutshell, semantic versioning tells us that each new version of a service should be tagged with a version number than consist of three parts MAJOR, MINOR, and PATCH, separated by a period. This is a version looking like this: `MAJOR.MINOR.PATCH`

Each of the three is an integer, starting with zero. Given this schema, one should increment

- MAJOR version when one makes incompatible changes to the API of the service
- MINOR version when one adds functionality to the service in a backward-compatible manner
- PATCH version when one implements backward-compatible bug fixes

Now, I admit this is a bit abstract and not easy to understand. Let us try to elaborate on what the preceding means in practice.

If your team is tasked to fix a bug in the code of the service you own, and this fix does not add any new functionality to the service, nor does it change any of the public interfaces or contracts of your service, then you should only increase the PATCH version of the service, compared to the one that is the valid one. That is, if you run version **`1.2.13`** in production and have to fix such a bug, then the new version of the service, after the bug fix, should be **`1.2.14`**.

Now imagine a situation where your team implements a new feature in the service, but no update on the public interfaces or contracts of the service is needed; then you should update the MINOR version. As an example, given that the current version in production is **`1.2.14`**, then your new version would have to be tagged with **`1.3.0`**. Please note that the MINOR version has increased from 12 to 13, and at the same time, the PATCH version has been reset to zero.

Finally, assume that, once again, your team is tasked to implement yet another new feature. This time the team realizes that one of the public contracts or interfaces needs to be changed in a way that is breaking backwards compatibility. In this case, the team has to increase the MAJOR version. So, from say version **`1.3.2`**, your team

would now have to go to `2.0.0`. Again note how we have increased the MAJOR version from 1 to 2, and at the same time, the MINOR and PATCH versions have been reset to zero.

If now you asked yourself why we are discussing this version strategy at length, then this is based on the many years of negative experience with many teams in many companies across all industries. A lot of unnecessary friction in the **software development life cycle** (SDLC) was present due to the fact that the companies and teams did not pay enough attention to establish, document, and enforce a versioning strategy such as semantic versioning. This led to misunderstandings, difficult and painful releases, and even resulted in production outages.

Thus remember, before you start dealing with an ever-growing number of application or service updates, make sure you agree on a common versioning strategy. Honestly, there is no good reason why this should not be the strategy of using semantic versions. Also, make sure the versioning is automated and an integral part of your CI/CD pipeline.

Next, we are going to discover how we can roll out a new version of a service without having a downtime in the system.

Using rolling updates and rollbacks

These days, our applications most often live in the cloud and are accessed by people from around the globe, 24 hours a day. If the application is mission-critical to the company, then it is not possible to just shut down the application or take it offline only to be able to deploy some new version of one or many services that make up the application.

Downtimes can negatively impact the reputation of the company behind the application. Downtimes can also be costly due to the fact that opportunities are missed while the application is not accessible. Take an e-shop as an example. While your application is down, potential customers cannot place orders, and thus the company misses out on revenue.

Thus, we need to come up with a deployment mechanism that allows us to release new versions of the application without requiring any downtime. We may even go so far as to not tolerate any degradation in the availability of the application.

would now have to go to `2.0.0`. Again note how we have increased the MAJOR version from 1 to 2, and at the same time, the MINOR and PATCH versions have been reset to zero.

If now you asked yourself why we are discussing this version strategy at length, then this is based on the many years of negative experience with many teams in many companies across all industries. A lot of unnecessary friction in the **software development life cycle** (SDLC) was present due to the fact that the companies and teams did not pay enough attention to establish, document, and enforce a versioning strategy such as semantic versioning. This led to misunderstandings, difficult and painful releases, and even resulted in production outages.

Thus remember, before you start dealing with an ever-growing number of application or service updates, make sure you agree on a common versioning strategy. Honestly, there is no good reason why this should not be the strategy of using semantic versions. Also, make sure the versioning is automated and an integral part of your CI/CD pipeline.

Next, we are going to discover how we can roll out a new version of a service without having a downtime in the system.

Using rolling updates and rollbacks

These days, our applications most often live in the cloud and are accessed by people from around the globe, 24 hours a day. If the application is mission-critical to the company, then it is not possible to just shut down the application or take it offline only to be able to deploy some new version of one or many services that make up the application.

Downtimes can negatively impact the reputation of the company behind the application. Downtimes can also be costly due to the fact that opportunities are missed while the application is not accessible. Take an e-shop as an example. While your application is down, potential customers cannot place orders, and thus the company misses out on revenue.

Thus, we need to come up with a deployment mechanism that allows us to release new versions of the application without requiring any downtime. We may even go so far as to not tolerate any degradation in the availability of the application.

Rolling updates and rollbacks are the default strategies that Kubernetes offers to enable a downtime-free deployment of a new version of an application service. How does this work in principle?

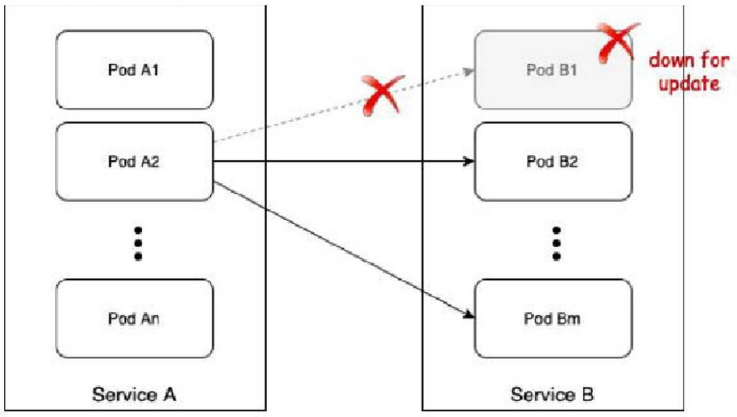


Figure 12.1: Rolling updates—tearing down pod

In the preceding graphic, we can see an application service A running as a Kubernetes ReplicaSet. For high availability reasons, the ReplicaSet has at least a scaling factor of 3. Thus, if one of the service instances fails, we have still two instances that are up and running and can serve the traffic. In our case, application service A runs with a replication factor of *n*.

As we know from *Chapter 10, Orchestrating containers*, Kubernetes Replicasets are used to guarantee, that we always have the number of service instances, or pods, available, that we declared in our Kubernetes YAML file that we used to create the ReplicaSet. If an instance fails, then Kubernetes automatically launches a new pod with a new service instance to reconcile.

Now, the second instance of application service A running in pod A2 requires access to application service B. Kubernetes automatically load balances this request and will direct it to one of the pods B1 to Bm of application service B.

Rolling updates and rollbacks are the default strategies that Kubernetes offers to enable a downtime-free deployment of a new version of an application service. How does this work in principle?

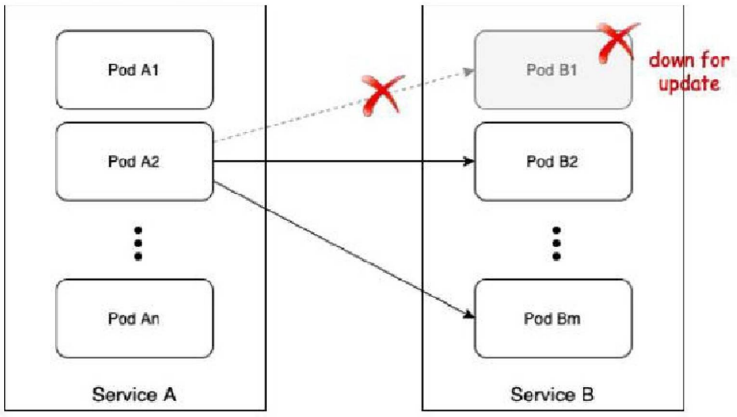


Figure 12.1: Rolling updates—tearing down pod

In the preceding graphic, we can see an application service A running as a Kubernetes ReplicaSet. For high availability reasons, the ReplicaSet has at least a scaling factor of 3. Thus, if one of the service instances fails, we have still two instances that are up and running and can serve the traffic. In our case, application service A runs with a replication factor of *n*.

As we know from *Chapter 10, Orchestrating containers*, Kubernetes Replicasets are used to guarantee, that we always have the number of service instances, or pods, available, that we declared in our Kubernetes YAML file that we used to create the ReplicaSet. If an instance fails, then Kubernetes automatically launches a new pod with a new service instance to reconcile.

Now, the second instance of application service A running in pod A2 requires access to application service B. Kubernetes automatically load balances this request and will direct it to one of the pods B1 to Bm of application service B.

When we now update application service B using the default rolling update strategy of Kubernetes, Kubernetes will tear down the first of the pods, in this case, pod B1. Temporarily this pod will not be available, but this is not a problem since we still have pods B2 until Bm up and running.

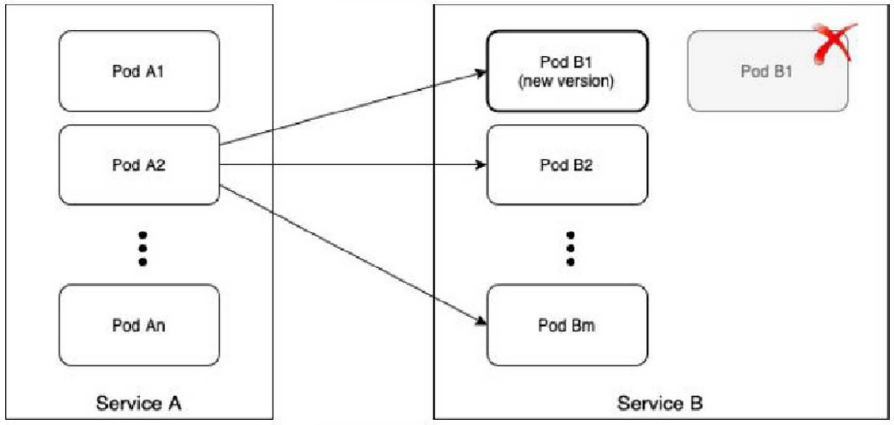


Figure 12.2: Rolling update—running new pod

In the preceding image, you see how Kubernetes instantiated a pod B1 containing the new version of service B to replace the torn down pod.

Kubernetes will repeat this process with pod B2, then pod B3, until it reaches pod Bm. After the last pod has been replaced, the rolling update is finished.

Note that in our example, Kubernetes would first tear down a pod with the old version of application service B and then replace it with an instance of the new version of application service B. But the opposite is also possible. Kubernetes will first start a new instance of application service B and add it to ReplicaSet. Only after the new instance is up and running successfully, Kubernetes will now tear down one of the old instances of application service B. This process is repeated until all old instances are replaced by new instances of application service B.

We should also notice that Kubernetes provides us with many opportunities to fine-tune the rolling update. For example, we can define how many pods shall be replaced at once in a rolling update. This is useful if we have big Replicasets with dozens or hundreds of pods. One would not want to replace them one at a time, as this would take a long time. Please check the official and very detailed documentation of Kubernetes for more insight.

Now let us do an example where we demonstrate a rolling update using Kubernetes Deployment resources. We can use our local single-node Kubernetes cluster for this example. There is no need for a multi-node cluster.

When we now update application service B using the default rolling update strategy of Kubernetes, Kubernetes will tear down the first of the pods, in this case, pod B1. Temporarily this pod will not be available, but this is not a problem since we still have pods B2 until Bm up and running.

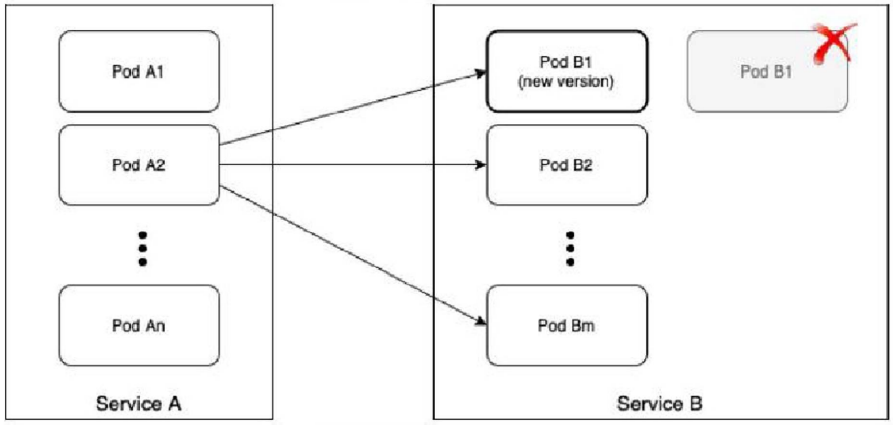


Figure 12.2: Rolling update—running new pod

In the preceding image, you see how Kubernetes instantiated a pod B1 containing the new version of service B to replace the torn down pod.

Kubernetes will repeat this process with pod B2, then pod B3, until it reaches pod Bm. After the last pod has been replaced, the rolling update is finished.

Note that in our example, Kubernetes would first tear down a pod with the old version of application service B and then replace it with an instance of the new version of application service B. But the opposite is also possible. Kubernetes will first start a new instance of application service B and add it to ReplicaSet. Only after the new instance is up and running successfully, Kubernetes will now tear down one of the old instances of application service B. This process is repeated until all old instances are replaced by new instances of application service B.

We should also notice that Kubernetes provides us with many opportunities to fine-tune the rolling update. For example, we can define how many pods shall be replaced at once in a rolling update. This is useful if we have big Replicasets with dozens or hundreds of pods. One would not want to replace them one at a time, as this would take a long time. Please check the official and very detailed documentation of Kubernetes for more insight.

Now let us do an example where we demonstrate a rolling update using Kubernetes Deployment resources. We can use our local single-node Kubernetes cluster for this example. There is no need for a multi-node cluster.

1. Create a new file called **whoami-deployment.yaml** and add the following content to it:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: whoami-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: whoami
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: whoami
    spec:
      containers:
      - name: whoami
        image: gnschenker/whoami:1.0
        ports:
        - containerPort: 80
```

2. Use kubectl to create the deployment on your cluster:

```
kubectl apply -f whoami-deployment.yaml
```

The output should look like the following:

```
deployment.apps/whoami-deployment created
```

1. Create a new file called **whoami-deployment.yaml** and add the following content to it:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: whoami-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: whoami
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: whoami
    spec:
      containers:
      - name: whoami
        image: gnschenker/whoami:1.0
        ports:
        - containerPort: 80
```

2. Use kubectl to create the deployment on your cluster:

```
kubectl apply -f whoami-deployment.yaml
```

The output should look like the following:

```
deployment.apps/whoami-deployment created
```

3. Now to show all resources that are created on the cluster by the preceding command use the following:

   ```
   kubectl get all
   ```

   and you should see an output similar to the following:

   ```
   NAME                                  READY   STATUS    RESTARTS   AGE
   pod/whoami-deployment-64686f77b-5jhvf 1/1 Running   0          7s
   pod/whoami-deployment-64686f77b-l5chj 1/1 Running   0          7s
   pod/whoami-deployment-64686f77b-x7v9r 1/1 Running   0          7s

   NAME             TYPE    CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
   service/kubernetes   ClusterIP  10.96.0.1  <none>  443/TCP   166d

   NAME                              READY   UP-TO-DATE   AVAILABLE   AGE
   deployment.apps/whoami-deployment   3/3  3        0           7s

   NAME                                      DESIRED   CURRENT   READY   AGE
   replicaset.apps/whoami-deployment-64686f77b  3  3     3       7s
   ```

   Notice that we have a single deployment resource called **whoami-deployment** on the cluster. The deployment also implicitly created a ReplicaSet and called it **whoami-deployment-64686f77b**. The hash code **64686f77b** was automatically created to uniquely name the ReplicaSet in case there is more than one for this deployment. And there will be more, as we will see shortly. Finally, we have three pods called **whoami-deployment-64686f77b-<code>**, where **<code>** is one of **5jhvf, l5chj**, and **x7v9r** in our case. In your case, the codes may be different. Once again, these codes are automatically assigned to the pod names by Kubernetes to keep these names unique.

   Great, we have three pods running on our system. But we cannot yet reach the application running inside the containers. For this, we need to add a Kubernetes service.

4. Create a file **whoami-service.yaml** with this content:

   ```
   apiVersion: v1
   kind: Service
   metadata:
     name: whoami-service
   spec:
   ```

3. Now to show all resources that are created on the cluster by the preceding command use the following:

```
kubectl get all
```

and you should see an output similar to the following:

```
NAME                                  READY   STATUS    RESTARTS   AGE
pod/whoami-deployment-64686f77b-5jhvf 1/1 Running   0          7s
pod/whoami-deployment-64686f77b-l5chj 1/1 Running   0          7s
pod/whoami-deployment-64686f77b-x7v9r 1/1 Running   0          7s

NAME             TYPE    CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP  10.96.0.1  <none>  443/TCP   166d

NAME                              READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/whoami-deployment   3/3  3        0           7s

NAME                                      DESIRED   CURRENT   READY   AGE
replicaset.apps/whoami-deployment-64686f77b  3  3     3       7s
```

Notice that we have a single deployment resource called **whoami-deployment** on the cluster. The deployment also implicitly created a ReplicaSet and called it **whoami-deployment-64686f77b**. The hash code **64686f77b** was automatically created to uniquely name the ReplicaSet in case there is more than one for this deployment. And there will be more, as we will see shortly. Finally, we have three pods called **whoami-deployment-64686f77b-<code>**, where **<code>** is one of **5jhvf, l5chj,** and **x7v9r** in our case. In your case, the codes may be different. Once again, these codes are automatically assigned to the pod names by Kubernetes to keep these names unique.

Great, we have three pods running on our system. But we cannot yet reach the application running inside the containers. For this, we need to add a Kubernetes service.

4. Create a file **whoami-service.yaml** with this content:

```
apiVersion: v1
kind: Service
metadata:
  name: whoami-service
spec:
```

```
  selector:
    app: whoami
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
```

5. Use the following command to list all services in the Kubernetes cluster:

   `kubectl get services`

   and you should see an output like this one:

```
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP        166d
whoami-service  NodePort 10.97.45.108 <none>     80:32648/TCP   78s
```

   As you can see, we have a new service called **`whoami-service`** or type **`NodePort`**. Also, note the PORT(S) column. There you can see that Kubernetes automatically assigned node port **`32648`** to this service. This is the port over which we can reach the service and the pods. Attention, in your case, the port number may differ since Kubernetes automatically assigns a free port.

6. We can now try to reach the **`whoami`** application using **`curl`**. Note that we do this in a loop with 10 iterations:

```
for i in {1..10}; do curl localhost:32648; echo '\n'; done
```

   and in our case, the output looks like this:

```
Hostname: whoami-deployment-64686f77b-x7v9r, Version: 1.0
Hostname: whoami-deployment-64686f77b-x7v9r, Version: 1.0
Hostname: whoami-deployment-64686f77b-x7v9r, Version: 1.0
Hostname: whoami-deployment-64686f77b-x7v9r, Version: 1.0
Hostname: whoami-deployment-64686f77b-l5chj, Version: 1.0
Hostname: whoami-deployment-64686f77b-x7v9r, Version: 1.0
Hostname: whoami-deployment-64686f77b-5jhvf, Version: 1.0
Hostname: whoami-deployment-64686f77b-x7v9r, Version: 1.0
```

   Please note that we have hit each of the three pods at least once. You can recognize this by their hostname in the output. The hostname of each pod,

```
  selector:
    app: whoami
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
```

5. Use the following command to list all services in the Kubernetes cluster:

   ```
   kubectl get services
   ```

   and you should see an output like this one:

   ```
   NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
   kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP        166d
   whoami-service  NodePort 10.97.45.108 <none>     80:32648/TCP   78s
   ```

   As you can see, we have a new service called `whoami-service` or type `NodePort`. Also, note the PORT(S) column. There you can see that Kubernetes automatically assigned node port `32648` to this service. This is the port over which we can reach the service and the pods. Attention, in your case, the port number may differ since Kubernetes automatically assigns a free port.

6. We can now try to reach the `whoami` application using `curl`. Note that we do this in a loop with 10 iterations:

   ```
   for i in {1..10}; do curl localhost:32648; echo '\n'; done
   ```

   and in our case, the output looks like this:

   ```
   Hostname: whoami-deployment-64686f77b-x7v9r, Version: 1.0
   Hostname: whoami-deployment-64686f77b-x7v9r, Version: 1.0
   Hostname: whoami-deployment-64686f77b-x7v9r, Version: 1.0
   Hostname: whoami-deployment-64686f77b-x7v9r, Version: 1.0
   Hostname: whoami-deployment-64686f77b-l5chj, Version: 1.0
   Hostname: whoami-deployment-64686f77b-x7v9r, Version: 1.0
   Hostname: whoami-deployment-64686f77b-5jhvf, Version: 1.0
   Hostname: whoami-deployment-64686f77b-x7v9r, Version: 1.0
   ```

   Please note that we have hit each of the three pods at least once. You can recognize this by their hostname in the output. The hostname of each pod,

by default, corresponds to the name of the pod. The service is doing its job and load-balancing the requests to the available pods.

Great, our deployment is working as expected. Now it is time to try an update of the application. Currently, we are using version 1.0 of the app, as shown in the preceding output, but we want to upgrade to version 1.1.

1. Edit the file `whoami-deployment.yaml` and change the image from `gnschenker/whoami:1.0` to `gnschenker/whoami:1.1`. Save the change.
2. Open a second terminal window and execute the following command:

   ```
   kubectl get rs -w
   ```

   This will list all replica sets defined on the cluster and refresh the output periodically. Note that in the command, we used the valid shortcut rs instead of ReplicaSet. Initially, you should only have one ReplicaSet present, and the output should look similar to the following:

   ```
   NAME                          DESIRED   CURRENT   READY   AGE
   whoami-deployment-64686f77b   3         3         3       72m
   ```

3. Now, in your first terminal window, apply the change to the cluster using the following command:

   ```
   kubectl apply -f whoami-deployment.yaml
   ```

   and observe the output in the second terminal window. After a short moment, you should see a second ReplicaSet appearing. This second ReplicaSet is the one governing the pods with the new version of the Docker image. Your output should look similar to the following:

   ```
   NAME                          DESIRED   CURRENT   READY   AGE
   whoami-deployment-64686f77b   3         3         3       72m
   whoami-deployment-664f97c6c9  1         0         0       0s
   whoami-deployment-64686f77b   2         3         3       79m
   whoami-deployment-664f97c6c9  2         0         0       0s
   whoami-deployment-64686f77b   2         3         3       79m
   whoami-deployment-664f97c6c9  2         0         0       0s
   whoami-deployment-64686f77b   2         2         2       79m
   whoami-deployment-664f97c6c9  2         1         0       0s
   whoami-deployment-664f97c6c9  2         2         0       0s
   whoami-deployment-664f97c6c9  2         2         1       1s
   ```

by default, corresponds to the name of the pod. The service is doing its job and load-balancing the requests to the available pods.

Great, our deployment is working as expected. Now it is time to try an update of the application. Currently, we are using version 1.0 of the app, as shown in the preceding output, but we want to upgrade to version 1.1.

1. Edit the file `whoami-deployment.yaml` and change the image from `gnschenker/whoami:1.0` to `gnschenker/whoami:1.1`. Save the change.
2. Open a second terminal window and execute the following command:

   ```
   kubectl get rs -w
   ```

   This will list all replica sets defined on the cluster and refresh the output periodically. Note that in the command, we used the valid shortcut rs instead of ReplicaSet. Initially, you should only have one ReplicaSet present, and the output should look similar to the following:

   ```
   NAME                          DESIRED   CURRENT   READY   AGE
   whoami-deployment-64686f77b   3         3         3       72m
   ```

3. Now, in your first terminal window, apply the change to the cluster using the following command:

   ```
   kubectl apply -f whoami-deployment.yaml
   ```

   and observe the output in the second terminal window. After a short moment, you should see a second ReplicaSet appearing. This second ReplicaSet is the one governing the pods with the new version of the Docker image. Your output should look similar to the following:

   ```
   NAME                          DESIRED   CURRENT   READY   AGE
   whoami-deployment-64686f77b   3         3         3       72m
   whoami-deployment-664f97c6c9  1         0         0       0s
   whoami-deployment-64686f77b   2         3         3       79m
   whoami-deployment-664f97c6c9  2         0         0       0s
   whoami-deployment-64686f77b   2         3         3       79m
   whoami-deployment-664f97c6c9  2         0         0       0s
   whoami-deployment-64686f77b   2         2         2       79m
   whoami-deployment-664f97c6c9  2         1         0       0s
   whoami-deployment-664f97c6c9  2         2         0       0s
   whoami-deployment-664f97c6c9  2         2         1       1s
   ```

```
whoami-deployment-664f97c6c9   2         2         2         1s
whoami-deployment-664f97c6c9   2         2         2         11s
whoami-deployment-64686f77b    0         2         2         79m
whoami-deployment-664f97c6c9   3         2         2         11s
whoami-deployment-64686f77b    0         2         2         79m
whoami-deployment-664f97c6c9   3         2         2         11s
whoami-deployment-64686f77b    0         0         0         79m
whoami-deployment-664f97c6c9   3         3         2         11s
whoami-deployment-664f97c6c9   3         3         3         13s
```

In the preceding output, we can see that the new ReplicaSet is called **whoami-deployment-664f97c6c9**. On the line where it first appears, we can see that its desired state is 1, and both values, **current** and **ready** are still zero. In the next line, we see that the desired state of the old ReplicaSet has been reduced to 2 to compensate for the fact that we have a new pod from the new ReplicaSet, but the total number of pods should be 3.

If you follow along, line by line, you can see that this pattern continues. We add one pod more to the new ReplicaSet and remove one from the old one for compensation. The rolling update is done once the new ReplicaSet has the desired number of pods up and ready, and at the same time, the old ReplicaSet has been reduced to zero pods.

Please note that the old ReplicaSet, once it has been drained, is not automatically deleted. This has a good reason. Kubernetes allows us to execute a rollback in case something goes wrong. In this case, it will reuse the old ReplicaSet.

4. We can get more information about the previous rolling update if we describe the deployment resource as follows:

```
kubectl describe deploy whoami-deployment
```

The resulting output is pretty long and detailed. At this time, we are only interested in the last part of the output, the list of events. It should look similar to the following:

```
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  12m   deployment-controller  Scaled
up replica set whoami-deployment-664f97c6c9 to 1
```

```
whoami-deployment-664f97c6c9   2         2         2         1s
whoami-deployment-664f97c6c9   2         2         2         11s
whoami-deployment-64686f77b    0         2         2         79m
whoami-deployment-664f97c6c9   3         2         2         11s
whoami-deployment-64686f77b    0         2         2         79m
whoami-deployment-664f97c6c9   3         2         2         11s
whoami-deployment-64686f77b    0         0         0         79m
whoami-deployment-664f97c6c9   3         3         2         11s
whoami-deployment-664f97c6c9   3         3         3         13s
```

In the preceding output, we can see that the new ReplicaSet is called **whoami-deployment-664f97c6c9**. On the line where it first appears, we can see that its desired state is 1, and both values, **current** and **ready** are still zero. In the next line, we see that the desired state of the old ReplicaSet has been reduced to 2 to compensate for the fact that we have a new pod from the new ReplicaSet, but the total number of pods should be 3.

If you follow along, line by line, you can see that this pattern continues. We add one pod more to the new ReplicaSet and remove one from the old one for compensation. The rolling update is done once the new ReplicaSet has the desired number of pods up and ready, and at the same time, the old ReplicaSet has been reduced to zero pods.

Please note that the old ReplicaSet, once it has been drained, is not automatically deleted. This has a good reason. Kubernetes allows us to execute a rollback in case something goes wrong. In this case, it will reuse the old ReplicaSet.

4. We can get more information about the previous rolling update if we describe the deployment resource as follows:

```
kubectl describe deploy whoami-deployment
```

The resulting output is pretty long and detailed. At this time, we are only interested in the last part of the output, the list of events. It should look similar to the following:

```
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  12m   deployment-controller  Scaled
up replica set whoami-deployment-664f97c6c9 to 1
```

```
  Normal  ScalingReplicaSet  12m   deployment-controller  Scaled
down replica set whoami-deployment-64686f77b to 2

  Normal  ScalingReplicaSet  12m   deployment-controller  Scaled
up replica set whoami-deployment-664f97c6c9 to 2

  Normal  ScalingReplicaSet  12m   deployment-controller  Scaled
down replica set whoami-deployment-64686f77b to 0

  Normal  ScalingReplicaSet  12m   deployment-controller  Scaled
up replica set whoami-deployment-664f97c6c9 to 3
```

You can see multiple scale-up and scale-down events in the preceding list, as expected.

5. We can check on the replicasets:

   ```
   kubectl get rs
   ```

   and should see the following:

   ```
   NAME                          DESIRED   CURRENT   READY   AGE
   whoami-deployment-64686f77b   0         0         0       103m
   whoami-deployment-664f97c6c9  3         3         3       24m
   ```

   As expected, the old ReplicaSet is still lingering around but has 0 pods associated, whilst the new one has three pods up and running. *Note the old ReplicaSet is still in the system and not automatically deleted by Kubernetes to make a rollback to the previous state possible. Kubernetes will just execute the reverse process as when updating.*

6. We also can use `kubectl` to check the status of the rollout of our new version by using the following command:

   ```
   kubectl rollout status deploy whoami-deployment
   ```

   and, if everything goes well, the output looks like the following:

   ```
   deployment "whoami-deployment" successfully rolled out
   ```

7. Finally, we describe the pods and make sure that they indeed represent the new version. To limit the output to the essential part, we can use this command:

   ```
   kubectl describe pods | grep Image:
   ```

   and indeed, we get the expected image for all three pods:

   ```
   Image:          gnschenker/whoami:1.1
   Image:          gnschenker/whoami:1.1
   Image:          gnschenker/whoami:1.1
   ```

```
  Normal  ScalingReplicaSet  12m   deployment-controller  Scaled
down replica set whoami-deployment-64686f77b to 2

  Normal  ScalingReplicaSet  12m   deployment-controller  Scaled
up replica set whoami-deployment-664f97c6c9 to 2

  Normal  ScalingReplicaSet  12m   deployment-controller  Scaled
down replica set whoami-deployment-64686f77b to 0

  Normal  ScalingReplicaSet  12m   deployment-controller  Scaled
up replica set whoami-deployment-664f97c6c9 to 3
```

You can see multiple scale-up and scale-down events in the preceding list, as expected.

5. We can check on the replicasets:

   ```
   kubectl get rs
   ```

   and should see the following:

   ```
   NAME                          DESIRED   CURRENT   READY   AGE
   whoami-deployment-64686f77b   0         0         0       103m
   whoami-deployment-664f97c6c9  3         3         3       24m
   ```

   As expected, the old ReplicaSet is still lingering around but has 0 pods associated, whilst the new one has three pods up and running. *Note the old ReplicaSet is still in the system and not automatically deleted by Kubernetes to make a rollback to the previous state possible. Kubernetes will just execute the reverse process as when updating.*

6. We also can use `kubectl` to check the status of the rollout of our new version by using the following command:

   ```
   kubectl rollout status deploy whoami-deployment
   ```

   and, if everything goes well, the output looks like the following:

   ```
   deployment "whoami-deployment" successfully rolled out
   ```

7. Finally, we describe the pods and make sure that they indeed represent the new version. To limit the output to the essential part, we can use this command:

   ```
   kubectl describe pods | grep Image:
   ```

   and indeed, we get the expected image for all three pods:

   ```
   Image:          gnschenker/whoami:1.1
   Image:          gnschenker/whoami:1.1
   Image:          gnschenker/whoami:1.1
   ```

With that, we have shown how to perform a rolling update of Kubernetes deployment from one version to the next.

Next, we want to show how we can rollback an upgrade to the previous version in case something goes wrong.

1. We start by analyzing the history of our deployment resource:

   ```
   kubectl rollout history deploy whoami-deployment
   ```

   we should see something like the following:

   ```
   deployment.apps/whoami-deployment
   REVISION  CHANGE-CAUSE
   1         <none>
   2         <none>
   ```

   Evidently, we have two revisions so far: the initial deployment of version 1.0 and the update to version 1.1.

2. If we want a bit more detailed information about the last revision, we can use the following command:

   ```
   kubectl rollout history deploy whoami-deployment --revision=2
   ```

   we have simply added the `--revision=2` command line parameter. We will see the following:

   ```
   deployment.apps/whoami-deployment with revision #2
   Pod Template:
     Labels:       app=whoami
           pod-template-hash=664f97c6c9
     Containers:
      whoami:
       Image:      gnschenker/whoami:1.1
       Port:       80/TCP
       Host Port:  0/TCP
       Environment:        <none>
       Mounts:     <none>
     Volumes:      <none>
   ```

With that, we have shown how to perform a rolling update of Kubernetes deployment from one version to the next.

Next, we want to show how we can rollback an upgrade to the previous version in case something goes wrong.

1. We start by analyzing the history of our deployment resource:

   ```
   kubectl rollout history deploy whoami-deployment
   ```

   we should see something like the following:

   ```
   deployment.apps/whoami-deployment
   REVISION  CHANGE-CAUSE
   1         <none>
   2         <none>
   ```

   Evidently, we have two revisions so far: the initial deployment of version 1.0 and the update to version 1.1.

2. If we want a bit more detailed information about the last revision, we can use the following command:

   ```
   kubectl rollout history deploy whoami-deployment --revision=2
   ```

   we have simply added the `--revision=2` command line parameter. We will see the following:

   ```
   deployment.apps/whoami-deployment with revision #2
   Pod Template:
     Labels:       app=whoami
           pod-template-hash=664f97c6c9
     Containers:
      whoami:
       Image:      gnschenker/whoami:1.1
       Port:       80/TCP
       Host Port:  0/TCP
       Environment:        <none>
       Mounts:     <none>
     Volumes:      <none>
   ```

3. Now using the following command, we can roll back to the previous revision:

   ```
   kubectl rollout undo deploy whoami-deployment
   ```

   and we will find ourselves back to revision 1.

4. Please check it out by using the various commands that you have used previously to gather details about the status of the rollout, and so on. For example, we can use the following:

   ```
   kubectl describe pods | grep Image:
   ```

   and should get the following:

   ```
   Image:          gnschenker/whoami:1.0
   Image:          gnschenker/whoami:1.0
   Image:          gnschenker/whoami:1.0
   ```

   proving the fact that we are back where we started.

To clean-up the system before you continue, use the following commands:

```
$ kubectl delete svc whoami-service
$ kubectl delete deploy whoami-deploy
```

Wonderful, now you can rollout new versions of your applications running on Kubernetes with confidence. Since the rollout is a rolling upgrade, you will at no time face an outage of your system. Pods of the service are gradually replaced with their new version.

It is important to remember that during the rollout phase, part of the traffic is handled by pods running the old version of the software, and part of the traffic is already handled by pods with the new version at the same time. This is not a problem at all if the new version of your application service is backward compatible with the old one.

It is now time to get to know another important way how you can upgrade your application services. This alternative rollout strategy is called blue-green deployment.

Enabling blue-green deployments

There are situations where a rolling update, as described in the previous section, is not enough. One would rather want to have the current version of the service continue running undisturbed in production while installing a new version in parallel. Once the new version has been deployed and tested, one can switch over the whole application traffic at once, from the old to the new version of the service. Ideally, this happens in a fully automated way triggered by a simple click of a button or similar.

3. Now using the following command, we can roll back to the previous revision:

   ```
   kubectl rollout undo deploy whoami-deployment
   ```

   and we will find ourselves back to revision 1.

4. Please check it out by using the various commands that you have used previously to gather details about the status of the rollout, and so on. For example, we can use the following:

   ```
   kubectl describe pods | grep Image:
   ```

   and should get the following:

   ```
   Image:          gnschenker/whoami:1.0
   Image:          gnschenker/whoami:1.0
   Image:          gnschenker/whoami:1.0
   ```

   proving the fact that we are back where we started.

To clean-up the system before you continue, use the following commands:

```
$ kubectl delete svc whoami-service
$ kubectl delete deploy whoami-deploy
```

Wonderful, now you can rollout new versions of your applications running on Kubernetes with confidence. Since the rollout is a rolling upgrade, you will at no time face an outage of your system. Pods of the service are gradually replaced with their new version.

It is important to remember that during the rollout phase, part of the traffic is handled by pods running the old version of the software, and part of the traffic is already handled by pods with the new version at the same time. This is not a problem at all if the new version of your application service is backward compatible with the old one.

It is now time to get to know another important way how you can upgrade your application services. This alternative rollout strategy is called blue-green deployment.

Enabling blue-green deployments

There are situations where a rolling update, as described in the previous section, is not enough. One would rather want to have the current version of the service continue running undisturbed in production while installing a new version in parallel. Once the new version has been deployed and tested, one can switch over the whole application traffic at once, from the old to the new version of the service. Ideally, this happens in a fully automated way triggered by a simple click of a button or similar.

If something goes wrong, and it turns out that the new version of the service shows unexpected and wrong behavior, one can easily switch back to the previous version at once. It is again just a click of a button or a turning of a switch that is needed.

Such a deployment is called a blue-green deployment. At times one also encounters the name red-black deployment for the same kind of deployment. This name is just a synonym for blue-green deployment.

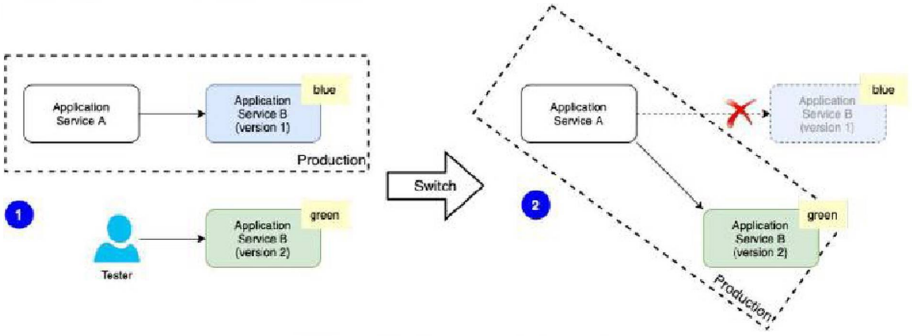


***Figure 12.3**: Blue-green deployment*

In the preceding image, we can see the following:

1. We have an application service A in production, which accesses an application service B. The latter is currently in version 1. This version 1 of application service B we call the blue deployment. We also see that version 2 of application service B has been deployed to the production system. It is not wired to the other services yet, though. We call this version the green version. Testers using manual or automated tests can internally access the green version and make sure it runs as expected.
2. After switching the routing from application service A to version 2 of application service B—the green version—the blue version is now unused and disconnected from the system. In case an unexpected bug is discovered in the green version of B, one can just switch back to the blue version.

Once the green version has been in production for a while and working flawlessly, the blue version can be removed from the system. The next upgrade of the application service B will now take the place of blue, and the switch will be from green to blue this time.

If we want to perform a blue-green deployment of our whoami container, then we can do so by using labels creatively. Remember, labels are used by Kubernetes services and other resources to determine which pods they serve.

If something goes wrong, and it turns out that the new version of the service shows unexpected and wrong behavior, one can easily switch back to the previous version at once. It is again just a click of a button or a turning of a switch that is needed.

Such a deployment is called a blue-green deployment. At times one also encounters the name red-black deployment for the same kind of deployment. This name is just a synonym for blue-green deployment.

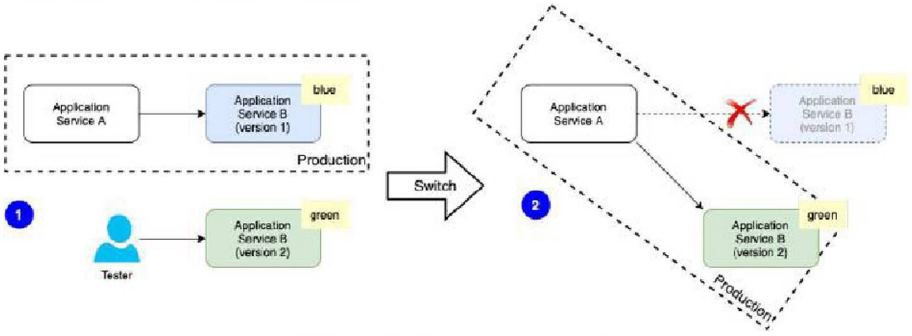


***Figure 12.3**: Blue-green deployment*

In the preceding image, we can see the following:

1. We have an application service A in production, which accesses an application service B. The latter is currently in version 1. This version 1 of application service B we call the blue deployment. We also see that version 2 of application service B has been deployed to the production system. It is not wired to the other services yet, though. We call this version the green version. Testers using manual or automated tests can internally access the green version and make sure it runs as expected.
2. After switching the routing from application service A to version 2 of application service B—the green version—the blue version is now unused and disconnected from the system. In case an unexpected bug is discovered in the green version of B, one can just switch back to the blue version.

Once the green version has been in production for a while and working flawlessly, the blue version can be removed from the system. The next upgrade of the application service B will now take the place of blue, and the switch will be from green to blue this time.

If we want to perform a blue-green deployment of our whoami container, then we can do so by using labels creatively. Remember, labels are used by Kubernetes services and other resources to determine which pods they serve.

Before we start, let us describe how such a deployment works in principle:

1. Deploy version 1 of your application, e.g., the `whoami` application as blue. We can do this by adding a `color` label to the pod. The label will have the value of blue.
2. Next, deploy the Kubernetes service for the application and define the (pod) selector such as that it contains the label `color: blue`
3. Now create a new version, say version 1.1, of the application. But this time, assign a label `color: green` to the pod
4. Test the green version of the application and assert that it works as expected
5. Finally, switch the traffic from the blue version of the application to the green version by updating the Kubernetes service. You need to modify the selector, such as the `color` label now having a value of green. Once the service is updated, all traffic will automatically be routed to the new version of the application having a label of `color: green`
6. Run the new green version 1.1 for some time and observe it. If the new version works as expected, you can now decommission, that is, uninstall the old, blue version 1.0.
7. Create the next version of the application. This time it will have the label `color: blue`
8. Test it.
9. Switch the color label in the service from green to blue, and so on.

As you can see, it is a continuous switch from blue to green and back to blue as we progress with the versions of the application by, for example, adding new features to it.

Let us now test this in practice with our simple whoami application.

1. Create a file `whoami-deploy-blue.yaml` containing the Kubernetes deployment. It has the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: whoami-deploy-blue
spec:
  replicas: 3
  selector:
    matchLabels:
```

Before we start, let us describe how such a deployment works in principle:

1. Deploy version 1 of your application, e.g., the `whoami` application as blue. We can do this by adding a `color` label to the pod. The label will have the value of blue.
2. Next, deploy the Kubernetes service for the application and define the (pod) selector such as that it contains the label `color: blue`
3. Now create a new version, say version 1.1, of the application. But this time, assign a label `color: green` to the pod
4. Test the green version of the application and assert that it works as expected
5. Finally, switch the traffic from the blue version of the application to the green version by updating the Kubernetes service. You need to modify the selector, such as the `color` label now having a value of green. Once the service is updated, all traffic will automatically be routed to the new version of the application having a label of `color: green`
6. Run the new green version 1.1 for some time and observe it. If the new version works as expected, you can now decommission, that is, uninstall the old, blue version 1.0.
7. Create the next version of the application. This time it will have the label `color: blue`
8. Test it.
9. Switch the color label in the service from green to blue, and so on.

As you can see, it is a continuous switch from blue to green and back to blue as we progress with the versions of the application by, for example, adding new features to it.

Let us now test this in practice with our simple whoami application.

1. Create a file `whoami-deploy-blue.yaml` containing the Kubernetes deployment. It has the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: whoami-deploy-blue
spec:
  replicas: 3
  selector:
    matchLabels:
```

```
      app: whoami
      color: blue
  template:
    metadata:
      labels:
        app: whoami
        color: blue
    spec:
      containers:
      - name: whoami
        image: gnschenker/whoami:1.0
        ports:
        - containerPort: 80
```

Notice that the content is pretty much the same as in the previous example that we used to demonstrate rolling updates. There are two differences, though. The first is the definition of the color label in the selector for the deployment object and the metadata of the pod. The second is the removal of the update strategy that we had defined in the rolling update example. We do not really need a special strategy here, as we are not deploying in place but rather in the background.

2. Create a file `whoami-svc-bg.yaml` to define our Kubernetes service with the following content:

```
apiVersion: v1
kind: Service
metadata:
  name: whoami-svc-bg
spec:
  selector:
    app: whoami
    color: blue
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
```

```
      app: whoami
      color: blue
  template:
    metadata:
      labels:
        app: whoami
        color: blue
    spec:
      containers:
      - name: whoami
        image: gnschenker/whoami:1.0
        ports:
        - containerPort: 80
```

Notice that the content is pretty much the same as in the previous example that we used to demonstrate rolling updates. There are two differences, though. The first is the definition of the color label in the selector for the deployment object and the metadata of the pod. The second is the removal of the update strategy that we had defined in the rolling update example. We do not really need a special strategy here, as we are not deploying in place but rather in the background.

2. Create a file **whoami-svc-bg.yaml** to define our Kubernetes service with the following content:

```
apiVersion: v1
kind: Service
metadata:
  name: whoami-svc-bg
spec:
  selector:
    app: whoami
    color: blue
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
```

It looks very familiar. The only difference in regards to the service we used to demonstrate the rolling update is the new name on line 4 and the addition of the label `color:blue` on Line 8.

3. Deploy the blue version of the application using the following:

```
$ kubectl create -f whoami-deploy-blue.yaml
deployment.apps/whoami-deploy-blue created
$ kubectl create -f whoami-svc-bg.yaml
service/whoami-svc-bg
```

4. List all services to discover to which port the service got assigned by Kubernetes

```
$ kubectl get service
```

you should see something like the following:

```
NAME         TYPE     CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes ClusterIP   10.96.0.1     <none>        443/TCP        168d
whoami-svc-bg   NodePort  10.107.124.94  <none>  80:31820/TCP   20s
```

In our case, the port number is `31820`. In your case, it may be a different number; thus, pay attention.

5. Test this application using curl:

```
$ curl localhost:31820
```

do not forget to replace the preceding port number with the one of your system. The response should be similar to the following:

```
Hostname: whoami-deploy-blue-7d57579d9c-kkb8j, Version: 1.0
```

And indeed, we have access to version `1.0` of the application. You may want to execute the curl command a few more times to observe how the Kubernetes service load balances the request to the three different pods running the `whoami` application.

Now, we are ready to deploy the next version of the `whoami` application.

1. Create a copy of the deployment file:

```
$ cp whoami-deploy-blue.yaml whoami-deploy-green.yaml
```

2. Modify the content of the new file `whoami-deploy-green.yaml` and

   a) Change the name of the deployment to `whoami-deploy-green` (Line 4)

It looks very familiar. The only difference in regards to the service we used to demonstrate the rolling update is the new name on line 4 and the addition of the label `color:blue` on Line 8.

3. Deploy the blue version of the application using the following:

```
$ kubectl create -f whoami-deploy-blue.yaml
deployment.apps/whoami-deploy-blue created
$ kubectl create -f whoami-svc-bg.yaml
service/whoami-svc-bg
```

4. List all services to discover to which port the service got assigned by Kubernetes

```
$ kubectl get service
```

you should see something like the following:

```
NAME         TYPE     CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes ClusterIP   10.96.0.1     <none>        443/TCP        168d
whoami-svc-bg   NodePort  10.107.124.94  <none>  80:31820/TCP   20s
```

In our case, the port number is `31820`. In your case, it may be a different number; thus, pay attention.

5. Test this application using curl:

```
$ curl localhost:31820
```

do not forget to replace the preceding port number with the one of your system. The response should be similar to the following:

```
Hostname: whoami-deploy-blue-7d57579d9c-kkb8j, Version: 1.0
```

And indeed, we have access to version `1.0` of the application. You may want to execute the curl command a few more times to observe how the Kubernetes service load balances the request to the three different pods running the `whoami` application.

Now, we are ready to deploy the next version of the `whoami` application.

1. Create a copy of the deployment file:

```
$ cp whoami-deploy-blue.yaml whoami-deploy-green.yaml
```

2. Modify the content of the new file `whoami-deploy-green.yaml` and

   a) Change the name of the deployment to `whoami-deploy-green` (Line 4)

b) Change the value of the label `color` from `blue` to `green` (Lines 10 and 15)

c) change the name of the Docker image to `gnschenker/whoami:1.1` (Line 19)

3. Now create the deployment for the green version with the following:

```
$ kubectl create -f whoami-deploy-green.yaml
```

4. List all deployments

```
$ kubectl get deployments
```

you should see something like the following:

```
NAME                 READY   UP-TO-DATE   AVAILABLE   AGE
whoami-deploy-blue   3/3     3            3           21m
whoami-deploy-green  3/3     3            3           9
```

As you can see, we have now two deployments, each having 3 pods.

5. Let us list all pods then:

```
$ kubectl get pods
```

and you should get the following:

```
NAME                                 READY   STATUS    RESTARTS
AGE
whoami-deploy-blue-7d57579d9c-dbssc  1/1     Running   0        22m
whoami-deploy-blue-7d57579d9c-dfrzx  1/1     Running   0        22m
whoami-deploy-blue-7d57579d9c-kkb8j  1/1     Running   0        22m
whoami-deploy-green-5d8949d449-ljnj2 1/1     Running   0        86s
whoami-deploy-green-5d8949d449-pk8pf 1/1     Running   0        86s
whoami-deploy-green-5d8949d449-sp94j 1/1     Running   0        86s
```

We have six pods up and running, three for each color, as you can see in the name of the pods.

6. We need to make sure, though, that the service still only serves the blue pods. We can do so by again using our friend the curl command and executing it in a loop:

```
$ for i in {1..20}; do curl localhost:31820; echo '\n'; done
```

b) Change the value of the label `color` from `blue` to `green` (Lines 10 and 15)

c) change the name of the Docker image to `gnschenker/whoami:1.1` (Line 19)

3. Now create the deployment for the green version with the following:

```
$ kubectl create -f whoami-deploy-green.yaml
```

4. List all deployments

```
$ kubectl get deployments
```

you should see something like the following:

```
NAME                 READY   UP-TO-DATE   AVAILABLE   AGE
whoami-deploy-blue   3/3     3            3           21m
whoami-deploy-green  3/3     3            3           9
```

As you can see, we have now two deployments, each having 3 pods.

5. Let us list all pods then:

```
$ kubectl get pods
```

and you should get the following:

```
NAME                                 READY   STATUS    RESTARTS
AGE
whoami-deploy-blue-7d57579d9c-dbssc  1/1     Running   0        22m
whoami-deploy-blue-7d57579d9c-dfrzx  1/1     Running   0        22m
whoami-deploy-blue-7d57579d9c-kkb8j  1/1     Running   0        22m
whoami-deploy-green-5d8949d449-ljnj2 1/1     Running   0        86s
whoami-deploy-green-5d8949d449-pk8pf 1/1     Running   0        86s
whoami-deploy-green-5d8949d449-sp94j 1/1     Running   0        86s
```

We have six pods up and running, three for each color, as you can see in the name of the pods.

6. We need to make sure, though, that the service still only serves the blue pods. We can do so by again using our friend the curl command and executing it in a loop:

```
$ for i in {1..20}; do curl localhost:31820; echo '\n'; done
```

Note we have abbreviated the output of the preceding command for clarity. It should look like the following:

```
Hostname: whoami-deploy-blue-7d57579d9c-dbssc, Version: 1.0
Hostname: whoami-deploy-blue-7d57579d9c-kkb8j, Version: 1.0
Hostname: whoami-deploy-blue-7d57579d9c-dfrzx, Version: 1.0
...
Hostname: whoami-deploy-blue-7d57579d9c-dfrzx, Version: 1.0
Hostname: whoami-deploy-blue-7d57579d9c-dbssc, Version: 1.0
```

Make sure that in the output, you only see blue pods!

7. Next, we want to define a helper service that makes it easy to test the green version of the application. Thus, add a new file called **whoami-svc-bg-test.yaml** and add the following content to it:

```
apiVersion: v1
kind: Service
metadata:
  name: whoami-svc-bg-test
spec:
  selector:
    app: whoami
    color: green
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
```

Note, the content is pretty much the same as in the other service. Just the value of the **color** label is now **green** – since we want to test the green deployment—and the service has a different name **whoami-svc-bg-test**, containing the word **test** for clarity.

8. Create this temporary test service with:

```
$ kubectl create -f whoami-svc-bg-test.yaml
```

Note we have abbreviated the output of the preceding command for clarity. It should look like the following:

```
Hostname: whoami-deploy-blue-7d57579d9c-dbssc, Version: 1.0
Hostname: whoami-deploy-blue-7d57579d9c-kkb8j, Version: 1.0
Hostname: whoami-deploy-blue-7d57579d9c-dfrzx, Version: 1.0
...
Hostname: whoami-deploy-blue-7d57579d9c-dfrzx, Version: 1.0
Hostname: whoami-deploy-blue-7d57579d9c-dbssc, Version: 1.0
```

Make sure that in the output, you only see blue pods!

7. Next, we want to define a helper service that makes it easy to test the green version of the application. Thus, add a new file called **whoami-svc-bg-test.yaml** and add the following content to it:

```
apiVersion: v1
kind: Service
metadata:
  name: whoami-svc-bg-test
spec:
  selector:
    app: whoami
    color: green
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
```

Note, the content is pretty much the same as in the other service. Just the value of the **color** label is now **green** – since we want to test the green deployment—and the service has a different name **whoami-svc-bg-test**, containing the word **test** for clarity.

8. Create this temporary test service with:

```
$ kubectl create -f whoami-svc-bg-test.yaml
```

9. List all services:

```
$ kubectl get service
```

you should see something along the line of:

```
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP  PORT(S)        AGE
kubernetes    ClusterIP   10.96.0.1       <none>       443/TCP        168d
whoami-svc-bg NodePort    10.102.114.211  <none> 80:31820/TCP    30m
whoami-svc-bg-test NodePort 10.104.122.227 <none> 80:31668/TCP    4s
```

We can easily see that we have now two services for the whoami application, one mapped to port 31820 and the second one that we are going to use for testing to port 31668.

10. Test the newly deployed green version of the application. Here for simplicity, we just use the curl command once again:

```
$ for i in {1..10}; do curl localhost:31668; echo '\n'; done
```

Make sure you replace the port in the preceding command with the value you get for your test service. The output looks similar to the following:

```
Hostname: whoami-deploy-green-5d8949d449-sp94j, Version: 1.1

...

Hostname: whoami-deploy-green-5d8949d449-ljnj2, Version: 1.1

Hostname: whoami-deploy-green-5d8949d449-pk8pf, Version: 1.1

...
```

And we make sure we only see green pods. This is indeed the case, and we can, thus, continue with confidence.

11. Now that we have tested the new green version of the application, we can finally modify our existing productive service. Open the file `whoami-svc-bg.yaml` and modify the value of the `color` label to `green` (Line 8)

12. Apply the change to the service using the following:

```
$ kubectl apply -f whoami-svc-bg.yaml
```

Do not worry about the warning; all is good:

```
Warning: resource services/whoami-svc-bg is missing the kubectl.
kubernetes.io/last-applied-configuration annotation which is
required by kubectl apply. kubectl apply should only be used on
```

9. List all services:

```
$ kubectl get service
```

you should see something along the line of:

```
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP  PORT(S)        AGE
kubernetes    ClusterIP   10.96.0.1       <none>       443/TCP        168d
whoami-svc-bg NodePort    10.102.114.211  <none> 80:31820/TCP    30m
whoami-svc-bg-test NodePort 10.104.122.227 <none> 80:31668/TCP    4s
```

We can easily see that we have now two services for the whoami application, one mapped to port 31820 and the second one that we are going to use for testing to port 31668.

10. Test the newly deployed green version of the application. Here for simplicity, we just use the curl command once again:

```
$ for i in {1..10}; do curl localhost:31668; echo '\n'; done
```

Make sure you replace the port in the preceding command with the value you get for your test service. The output looks similar to the following:

```
Hostname: whoami-deploy-green-5d8949d449-sp94j, Version: 1.1

...

Hostname: whoami-deploy-green-5d8949d449-ljnj2, Version: 1.1

Hostname: whoami-deploy-green-5d8949d449-pk8pf, Version: 1.1

...
```

And we make sure we only see green pods. This is indeed the case, and we can, thus, continue with confidence.

11. Now that we have tested the new green version of the application, we can finally modify our existing productive service. Open the file `whoami-svc-bg.yaml` and modify the value of the `color` label to `green` (Line 8)

12. Apply the change to the service using the following:

```
$ kubectl apply -f whoami-svc-bg.yaml
```

Do not worry about the warning; all is good:

```
Warning: resource services/whoami-svc-bg is missing the kubectl.
kubernetes.io/last-applied-configuration annotation which is
required by kubectl apply. kubectl apply should only be used on
```

```
resources created declaratively by either kubectl create --save-
config or kubectl apply. The missing annotation will be patched
automatically

service/whoami-svc-bg configured
```

13. Test this application using `curl`:

    ```
    $ curl localhost:31820
    ```

    and the response should be similar to this:

    ```
    Hostname: whoami-deploy-green-5d8949d449-ljnj2, Version: 1.1
    ```

    And indeed, we have now the green version put live.

Great, you have learned how to do a blue-green deployment on Kubernetes by leveraging labels. Before you continue, you may want to clean up your environment. Use the following commands to do so:

```
$ kubectl delete service whoami-svc-bg

$ kubectl delete service whoami-svc-bg-test

$ kubectl delete deploy whoami-deploy-blue

$ kubectl delete deploy whoami-deploy-green
```

Next, we want to explore a more sophisticated version of the blue-green deployment, also called canary release.

Realizing canary releases

Canary releases are used whenever we want to achieve a gradual and controlled switch over from one version of the application to the next one. You may now say that this sounds exactly like what we did when using the standard rolling updates with Kubernetes. And to a certain extent, you are right. But rolling updates are rather limited in how you can control the gradual switch over from the old to the new version of the application. Rolling updates happen in an automated way, and we have limited influence in the process.

Canary releases, on the other hand, allow us to do a blue-green deployment and then provide us all the means to first test the new version in isolation and afterwards define a custom process on how we want the switch over to happen. Often we first funnel a small percentage of the traffic to the new version. The majority of the traffic continues to be routed through the old version. We observe the system, and if everything behaves as expected, we slowly increase the traffic that we funnel to the new version until we reach 100%. Then we can decommission the old version.

Well, let us see how we do this in practice. We first start by using all the Kubernetes concepts you already know, such as labels and deployments.

```
resources created declaratively by either kubectl create --save-
config or kubectl apply. The missing annotation will be patched
automatically

service/whoami-svc-bg configured
```

13. Test this application using `curl`:

    ```
    $ curl localhost:31820
    ```

    and the response should be similar to this:

    ```
    Hostname: whoami-deploy-green-5d8949d449-ljnj2, Version: 1.1
    ```

    And indeed, we have now the green version put live.

Great, you have learned how to do a blue-green deployment on Kubernetes by leveraging labels. Before you continue, you may want to clean up your environment. Use the following commands to do so:

```
$ kubectl delete service whoami-svc-bg

$ kubectl delete service whoami-svc-bg-test

$ kubectl delete deploy whoami-deploy-blue

$ kubectl delete deploy whoami-deploy-green
```

Next, we want to explore a more sophisticated version of the blue-green deployment, also called canary release.

Realizing canary releases

Canary releases are used whenever we want to achieve a gradual and controlled switch over from one version of the application to the next one. You may now say that this sounds exactly like what we did when using the standard rolling updates with Kubernetes. And to a certain extent, you are right. But rolling updates are rather limited in how you can control the gradual switch over from the old to the new version of the application. Rolling updates happen in an automated way, and we have limited influence in the process.

Canary releases, on the other hand, allow us to do a blue-green deployment and then provide us all the means to first test the new version in isolation and afterwards define a custom process on how we want the switch over to happen. Often we first funnel a small percentage of the traffic to the new version. The majority of the traffic continues to be routed through the old version. We observe the system, and if everything behaves as expected, we slowly increase the traffic that we funnel to the new version until we reach 100%. Then we can decommission the old version.

Well, let us see how we do this in practice. We first start by using all the Kubernetes concepts you already know, such as labels and deployments.

1. Copy the deployment file for blue, `whoami-deploy-blue.yaml` using the following command:
   ```
   $ cp whoami-deploy-blue.yaml whoami-deploy-blue-canary.yaml
   ```
2. Copy the deployment file for green, `whoami-deploy-green.yaml` using the following command:
   ```
   $ cp whoami-deploy-green.yaml whoami-deploy-green-canary.yaml
   ```
3. Copy the service for the blue-green deployment, whoami-svc-bg.yaml using the following command:
   ```
   $ cp whoami-svc-bg.yaml whoami-svc-canary.yaml
   ```
4. Edit the file `whoami-deploy-blue-canary.yaml` as follows:
   a) change the name of the deployment to `whoami-deploy-blue-canary`
   b) set the number of replicas to `5`
5. Edit the file `whoami-deploy-green-canary.yaml` as follows:
   a) change the name of the deployment to `whoami-deploy-green-canary`
   b) set the number of replicas to `1`
6. Edit the file `whoami-svc-canary.yaml` as follows:
   a) change the name of the service to `whoami-svc-canary`
   b) make sure the value of the label `color` is set to `blue`
7. Deploy the blue pods like the following:
   ```
   $ kubectl create -f whoami-deploy-blue-canary.yaml
   ```
   The output should be:
   ```
   deployment.apps/whoami-deploy-blue-canary created
   ```
8. Deploy the service as follows:
   ```
   $ kubectl create -f whoami-svc-canary.yaml
   ```
   The output of the preceding command is as follows:
   ```
   service/whoami-svc-canary created
   ```
9. List all services and write down the port number to which your service is mapped:
   ```
   $ kubectl get service
   ```
   in our case it looks like this:

1. Copy the deployment file for blue, `whoami-deploy-blue.yaml` using the following command:
   ```
   $ cp whoami-deploy-blue.yaml whoami-deploy-blue-canary.yaml
   ```
2. Copy the deployment file for green, `whoami-deploy-green.yaml` using the following command:
   ```
   $ cp whoami-deploy-green.yaml whoami-deploy-green-canary.yaml
   ```
3. Copy the service for the blue-green deployment, whoami-svc-bg.yaml using the following command:
   ```
   $ cp whoami-svc-bg.yaml whoami-svc-canary.yaml
   ```
4. Edit the file `whoami-deploy-blue-canary.yaml` as follows:
   a) change the name of the deployment to `whoami-deploy-blue-canary`
   b) set the number of replicas to `5`
5. Edit the file `whoami-deploy-green-canary.yaml` as follows:
   a) change the name of the deployment to `whoami-deploy-green-canary`
   b) set the number of replicas to `1`
6. Edit the file `whoami-svc-canary.yaml` as follows:
   a) change the name of the service to `whoami-svc-canary`
   b) make sure the value of the label `color` is set to `blue`
7. Deploy the blue pods like the following:
   ```
   $ kubectl create -f whoami-deploy-blue-canary.yaml
   ```
   The output should be:
   ```
   deployment.apps/whoami-deploy-blue-canary created
   ```
8. Deploy the service as follows:
   ```
   $ kubectl create -f whoami-svc-canary.yaml
   ```
   The output of the preceding command is as follows:
   ```
   service/whoami-svc-canary created
   ```
9. List all services and write down the port number to which your service is mapped:
   ```
   $ kubectl get service
   ```
   in our case it looks like this:

```
NAME          TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
kubernetes    ClusterIP   10.96.0.1      <none>        443/TCP        169d
whoami-svc-canary  NodePort   10.101.142.42  <none>  80:32250/TCP   2m1s
```

Apparently, our port number is **32250**.

10. Test the application:

    ```
    $ for i in {1..20}; do curl localhost:32250; echo '\n'; done
    ```

    Make sure you replace the port in the preceding command with the one Kubernetes assigned to your service. Make sure you only see blue pods and that you see five different instances.

So far, so good. This is nothing new for us at this time. But now the interesting part comes. We want to run approximately 20% of the traffic through the new version of the **whoami** application. If all goes well, then we gradually will increase the traffic to green by scaling up the number of pods for green and, at the same time, scaling down the number of blue pods.

1. Deploy the green version of the pods:

    ```
    $ kubectl create -f whoami-deploy-green-canary.yaml
    ```

    This will deploy a single green pod due to the fact that we set the number of replicas in the file to one.

    At this time, the green pod is not accessible through the service since the selector of the service only selects blue pods.

2. Edit the service **whoami-svc-canary** with this command:

    ```
    $ kubectl edit svc whoami-svc-canary
    ```

    and remove the label **color** by deleting Line 8. Save your changes. The changes will be applied immediately.

3. Test the application:

    ```
    $ for i in {1..20}; do curl localhost:32250; echo '\n'; done
    ```

    Observe that now part of the traffic is funneled through the green pod, as shown in the following snippet of our output:

    ```
    ...
    Hostname: whoami-deploy-blue-canary-7d57579d9c-8454g, Version: 1.0
    Hostname: whoami-deploy-green-canary-5d8949d449-tkbdb, Version:
    1.1
    ```

```
NAME          TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
kubernetes    ClusterIP   10.96.0.1      <none>        443/TCP        169d
whoami-svc-canary  NodePort   10.101.142.42  <none>  80:32250/TCP   2m1s
```

Apparently, our port number is **32250**.

10. Test the application:

    ```
    $ for i in {1..20}; do curl localhost:32250; echo '\n'; done
    ```

    Make sure you replace the port in the preceding command with the one Kubernetes assigned to your service. Make sure you only see blue pods and that you see five different instances.

So far, so good. This is nothing new for us at this time. But now the interesting part comes. We want to run approximately 20% of the traffic through the new version of the **whoami** application. If all goes well, then we gradually will increase the traffic to green by scaling up the number of pods for green and, at the same time, scaling down the number of blue pods.

1. Deploy the green version of the pods:

    ```
    $ kubectl create -f whoami-deploy-green-canary.yaml
    ```

    This will deploy a single green pod due to the fact that we set the number of replicas in the file to one.

    At this time, the green pod is not accessible through the service since the selector of the service only selects blue pods.

2. Edit the service **whoami-svc-canary** with this command:

    ```
    $ kubectl edit svc whoami-svc-canary
    ```

    and remove the label **color** by deleting Line 8. Save your changes. The changes will be applied immediately.

3. Test the application:

    ```
    $ for i in {1..20}; do curl localhost:32250; echo '\n'; done
    ```

    Observe that now part of the traffic is funneled through the green pod, as shown in the following snippet of our output:

    ```
    ...
    Hostname: whoami-deploy-blue-canary-7d57579d9c-8454g, Version: 1.0
    Hostname: whoami-deploy-green-canary-5d8949d449-tkbdb, Version:
    1.1
    ```

```
Hostname: whoami-deploy-blue-canary-7d57579d9c-j67vj, Version: 1.0
Hostname: whoami-deploy-blue-canary-7d57579d9c-88nbd, Version: 1.0
...
```

4. Scale down the blue deployment using this imperative command:

   ```
   $ kubectl scale deploy whoami-deploy-blue-canary --replicas=4
   ```

   The output is:

   ```
   deployment.apps/whoami-deploy-blue-canary scaled
   ```

5. List all deployments:

   ```
   $ kubectl get deployment
   ```

   and you should see the following:

   ```
   NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
   whoami-deploy-blue-canary   4/4     4            4           28m
   whoami-deploy-green-canary  1/1     1            1           13m
   ```

   indicating that we have four blue pods and one green pod. That now makes 20% of the traffic flow through the green.

6. Next, we want to funnel even more traffic through the new version of the application. For this, we have to scale up the green pods and, at the same time, scale down the blue pods. Do this:

   ```
   $ kubectl scale deploy whoami-deploy-green-canary --replicas=2
   $ kubectl scale deploy whoami-deploy-blue-canary --replicas=3
   ```

   and double check that the pods scale as expected with:

   ```
   $ kubectl get deployment
   ```

7. Make sure approximately 40% of the traffic is funneled to the green version and about 60% to the blue version:

   ```
   $ for i in {1..20}; do curl localhost:32250; echo '\n'; done
   ```

8. Continue to scale up green and scale down blue until you reach zero pods for blue.

9. When you are done, clean up the system with the following commands:

   ```
   $ kubectl delete svc/whoami-svc-canary
   $ kubectl delete deploy whoami-deploy-blue-canary
   $ kubectl delete deploy whoami-deploy-green-canary
   ```

OK, we have managed to create a canary release just with the standard tools of Kubernetes. But we can do better. We will show, as an example, how much benefit

```
Hostname: whoami-deploy-blue-canary-7d57579d9c-j67vj, Version: 1.0
Hostname: whoami-deploy-blue-canary-7d57579d9c-88nbd, Version: 1.0
...
```

4. Scale down the blue deployment using this imperative command:

   ```
   $ kubectl scale deploy whoami-deploy-blue-canary --replicas=4
   ```

   The output is:

   ```
   deployment.apps/whoami-deploy-blue-canary scaled
   ```

5. List all deployments:

   ```
   $ kubectl get deployment
   ```

   and you should see the following:

   ```
   NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
   whoami-deploy-blue-canary   4/4     4            4           28m
   whoami-deploy-green-canary  1/1     1            1           13m
   ```

   indicating that we have four blue pods and one green pod. That now makes 20% of the traffic flow through the green.

6. Next, we want to funnel even more traffic through the new version of the application. For this, we have to scale up the green pods and, at the same time, scale down the blue pods. Do this:

   ```
   $ kubectl scale deploy whoami-deploy-green-canary --replicas=2
   $ kubectl scale deploy whoami-deploy-blue-canary --replicas=3
   ```

   and double check that the pods scale as expected with:

   ```
   $ kubectl get deployment
   ```

7. Make sure approximately 40% of the traffic is funneled to the green version and about 60% to the blue version:

   ```
   $ for i in {1..20}; do curl localhost:32250; echo '\n'; done
   ```

8. Continue to scale up green and scale down blue until you reach zero pods for blue.

9. When you are done, clean up the system with the following commands:

   ```
   $ kubectl delete svc/whoami-svc-canary
   $ kubectl delete deploy whoami-deploy-blue-canary
   $ kubectl delete deploy whoami-deploy-green-canary
   ```

OK, we have managed to create a canary release just with the standard tools of Kubernetes. But we can do better. We will show, as an example, how much benefit

we can achieve by using a service mesh such as Gloo. Let us have a look.

Canary releases with Gloo

In the previous example, we enabled canary releases just with the means provided by Kubernetes out of the box, namely, labels and deployments. This works but introduces a strong coupling between the amount of traffic funneled through a particular version of the application service and the scaling factor of the blue and green versions of the application service. In this section, we want to show how a canary release can be enabled more elegantly by using a service mesh such as Gloo. Let us start.

1. First, we install Gloo on our Kubernetes cluster. In the author's case, it is the Kubernetes cluster of Docker Desktop on a MacBook Air with the M1 processor

    a. Download and install Gloo and add the binaries folder to the path with the following command:

    ```
    $ curl -sL https://run.solo.io/gloo/install | sh
    $ export PATH=$HOME/.gloo/bin:$PATH
    ```

    b. Make sure Gloo is installed with the following:

    ```
    $ glooctl version
    ```

    You should see something like the following:

    ```
    Client: {"version":"1.11.7"}
    Server: version undefined, could not find any version of gloo
    running
    ```

    c. With the following command, we install the gateway for our cluster:

    ```
    $ glooctl install gateway
    ```

    and after a moment or so, you should see this output in your terminal:

    ```
    Creating namespace gloo-system... Done.
    Starting Gloo installation...

    Gloo was successfully installed!
    ```

    d. Finally, we can double-check the installation, e.g., with the following:

    ```
    $ kubectl get pod -n gloo-system
    ```

    and we should see the following:

    ```
    NAME                          READY   STATUS    RESTARTS   AGE
    ```

we can achieve by using a service mesh such as Gloo. Let us have a look.

Canary releases with Gloo

In the previous example, we enabled canary releases just with the means provided by Kubernetes out of the box, namely, labels and deployments. This works but introduces a strong coupling between the amount of traffic funneled through a particular version of the application service and the scaling factor of the blue and green versions of the application service. In this section, we want to show how a canary release can be enabled more elegantly by using a service mesh such as Gloo. Let us start.

1. First, we install Gloo on our Kubernetes cluster. In the author's case, it is the Kubernetes cluster of Docker Desktop on a MacBook Air with the M1 processor

    a. Download and install Gloo and add the binaries folder to the path with the following command:

    ```
    $ curl -sL https://run.solo.io/gloo/install | sh
    $ export PATH=$HOME/.gloo/bin:$PATH
    ```

    b. Make sure Gloo is installed with the following:

    ```
    $ glooctl version
    ```

    You should see something like the following:

    ```
    Client: {"version":"1.11.7"}
    Server: version undefined, could not find any version of gloo
    running
    ```

    c. With the following command, we install the gateway for our cluster:

    ```
    $ glooctl install gateway
    ```

    and after a moment or so, you should see this output in your terminal:

    ```
    Creating namespace gloo-system... Done.
    Starting Gloo installation...

    Gloo was successfully installed!
    ```

    d. Finally, we can double-check the installation, e.g., with the following:

    ```
    $ kubectl get pod -n gloo-system
    ```

    and we should see the following:

    ```
    NAME                          READY   STATUS    RESTARTS   AGE
    ```

```
discovery-667cb696cc-7ksdm    1/1     Running   0          32m
gateway-569c6c7f79-zh2f6      1/1     Running   0          32m
gateway-proxy-99c66dc8-fff6z  1/1     Running   0          32m
gloo-86b4b8b498-ljxd8         1/1     Running   0          32m
```

We are now ready to install the application.

2. Create a new file called **whoami-gloo-deploy.yaml** with the following content:

```
apiVersion: v1
kind: Service
metadata:
  name: whoami
spec:
  selector:
    app: whoami
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: whoami-blue
spec:
  replicas: 1
  selector:
    matchLabels:
      app: whoami
      color: blue
  template:
    metadata:
      labels:
```

```
discovery-667cb696cc-7ksdm     1/1     Running   0          32m
gateway-569c6c7f79-zh2f6       1/1     Running   0          32m
gateway-proxy-99c66dc8-fff6z   1/1     Running   0          32m
gloo-86b4b8b498-ljxd8          1/1     Running   0          32m
```

We are now ready to install the application.

2. Create a new file called **whoami-gloo-deploy.yaml** with the following content:

```
apiVersion: v1
kind: Service
metadata:
  name: whoami
spec:
  selector:
    app: whoami
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: whoami-blue
spec:
  replicas: 1
  selector:
    matchLabels:
      app: whoami
      color: blue
  template:
    metadata:
      labels:
```

```
        app: whoami
        color: blue
    spec:
      containers:
      - name: whoami
        image: gnschenker/whoami:1.0
        ports:
        - containerPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: whoami-green
spec:
  replicas: 1
  selector:
    matchLabels:
      app: whoami
      color: green
  template:
    metadata:
      labels:
        app: whoami
        color: green
    spec:
      containers:
      - name: whoami
        image: gnschenker/whoami:1.1
        ports:
        - containerPort: 80
```

Note that we have added the manifests for the service and the deployments for version 1.0 and version 1.1 to the same file to simplify things a bit. The definitions are pretty much the same as the ones we used in the previous examples in this chapter. We "only" adjusted the names of the service and

```
        app: whoami
        color: blue
    spec:
      containers:
      - name: whoami
        image: gnschenker/whoami:1.0
        ports:
        - containerPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: whoami-green
spec:
  replicas: 1
  selector:
    matchLabels:
      app: whoami
      color: green
  template:
    metadata:
      labels:
        app: whoami
        color: green
    spec:
      containers:
      - name: whoami
        image: gnschenker/whoami:1.1
        ports:
        - containerPort: 80
```

Note that we have added the manifests for the service and the deployments for version 1.0 and version 1.1 to the same file to simplify things a bit. The definitions are pretty much the same as the ones we used in the previous examples in this chapter. We "only" adjusted the names of the service and

the deployments. Also, note how we separate individual manifest with three dashes (---). This is the Kubernetes standard.

Also, note that we have requested only a single replica for both deployments. We do this purposefully to show that the number of replicas is completely decoupled from the amount of traffic that we will send to blue or green. Stay tuned.

3. Now create another file and call it **`whoami-gloo-upstream.yaml`**, and add the following content to it:

```
apiVersion: gloo.solo.io/v1
kind: Upstream
metadata:
  name: whoami
  namespace: gloo-system
spec:
  kube:
    selector:
      app: whoami
    serviceName: whoami
    serviceNamespace: default
    servicePort: 80
    subsetSpec:
      selectors:
        - keys:
            - color
```

With this, we model the application as a Gloo **`Upstream`**, which is Gloo's abstraction for a traffic destination.

Explaining all the details about Gloo is beyond the scope of this book. If you want to know more about Gloo, we encourage you to read the external resources about it listed at the end of this chapter.

4. Next, create a file called **`whoami-gloo-virtual-service.yaml`** with the following content:

```
apiVersion: gateway.solo.io/v1
kind: VirtualService
metadata:
```

the deployments. Also, note how we separate individual manifest with three dashes (---). This is the Kubernetes standard.

Also, note that we have requested only a single replica for both deployments. We do this purposefully to show that the number of replicas is completely decoupled from the amount of traffic that we will send to blue or green. Stay tuned.

3. Now create another file and call it **`whoami-gloo-upstream.yaml`**, and add the following content to it:

```
apiVersion: gloo.solo.io/v1
kind: Upstream
metadata:
  name: whoami
  namespace: gloo-system
spec:
  kube:
    selector:
      app: whoami
    serviceName: whoami
    serviceNamespace: default
    servicePort: 80
    subsetSpec:
      selectors:
        - keys:
            - color
```

With this, we model the application as a Gloo **`Upstream`**, which is Gloo's abstraction for a traffic destination.

Explaining all the details about Gloo is beyond the scope of this book. If you want to know more about Gloo, we encourage you to read the external resources about it listed at the end of this chapter.

4. Next, create a file called **`whoami-gloo-virtual-service.yaml`** with the following content:

```
apiVersion: gateway.solo.io/v1
kind: VirtualService
metadata:
```

```
  name: whoami
  namespace: gloo-system
spec:
  virtualHost:
    domains:
      - "*"
    routes:
      - matchers:
          - headers:
              - name: stage
                value: canary
            prefix: /
        routeAction:
          single:
            upstream:
              name: whoami
              namespace: gloo-system
            subset:
              values:
                color: green
      - matchers:
          - prefix: /
        routeAction:
          single:
            upstream:
              name: whoami
              namespace: gloo-system
            subset:
              values:
                color: blue
```

In the preceding manifest we define a route to the upstream in Gloo with a **VirtualService**. Note the **routes** part of this manifest. It contains two **matchers** entries. These are the filters that help you define to which destination the traffic shall be routed.

```
  name: whoami
  namespace: gloo-system
spec:
  virtualHost:
    domains:
      - "*"
    routes:
      - matchers:
          - headers:
              - name: stage
                value: canary
            prefix: /
        routeAction:
          single:
            upstream:
              name: whoami
              namespace: gloo-system
            subset:
              values:
                color: green
      - matchers:
          - prefix: /
        routeAction:
          single:
            upstream:
              name: whoami
              namespace: gloo-system
            subset:
              values:
                color: blue
```

In the preceding manifest we define a route to the upstream in Gloo with a **VirtualService**. Note the **routes** part of this manifest. It contains two **matchers** entries. These are the filters that help you define to which destination the traffic shall be routed.

In the first `matchers` entry, we define that traffic that arrives with a header of `stage: canary` is matching this filter and will be routed to the destination defined in the `routeAction`. In this case, it will be directed to the whoami upstream that has the label `color: green` defined. That is, the traffic will be funneled to pods running the whoami application with version 1.1.

The second matchers entry applies to everything else and will be directed to the whoami upstream that has the label `color: blue` defined. That is, the traffic will be funneled to pods running the whoami application with version 1.0.

Attention: the order of the matchers entries matters. If you switch them in our example, all traffic will always be directed to the blue version.

5. It is now time to deploy the application. Use the following commands to do so:

   a) Deploy the two versions of the whoami application:

   ```
   $ kubectl apply -f whoami-gloo-deploy.yaml
   ```

   resulting in the following output:

   ```
   service/whoami created
   deployment.apps/whoami-blue created
   deployment.apps/whoami-green created
   ```

   b) Deploy the Gloo Upstream resource:

   ```
   $ kubectl apply -f whoami-gloo-upstream.yaml
   ```

   which gives the following output:

   ```
   upstream.gloo.solo.io/whoami created
   ```

   c) Finally, deploy the VirtualService resource:

   ```
   $ kubectl apply -f whoami-gloo-virtual-service.yaml
   ```

   resulting in this output:

   ```
   virtualservice.gateway.solo.io/whoami created
   ```

6. Let us see if we can run the application. First, we need to find out the target URL. We can do this with the following command:

   ```
   $ glooctl proxy url
   ```

   In our case, the output looks like the following:

   ```
   http://localhost:80
   ```

7. Let us now try to access the application. We can use the preceding URL:

In the first `matchers` entry, we define that traffic that arrives with a header of `stage: canary` is matching this filter and will be routed to the destination defined in the `routeAction`. In this case, it will be directed to the whoami upstream that has the label `color: green` defined. That is, the traffic will be funneled to pods running the whoami application with version 1.1.

The second matchers entry applies to everything else and will be directed to the whoami upstream that has the label `color: blue` defined. That is, the traffic will be funneled to pods running the whoami application with version 1.0.

Attention: the order of the matchers entries matters. If you switch them in our example, all traffic will always be directed to the blue version.

5. It is now time to deploy the application. Use the following commands to do so:

   a) Deploy the two versions of the whoami application:

   ```
   $ kubectl apply -f whoami-gloo-deploy.yaml
   ```

   resulting in the following output:

   ```
   service/whoami created
   deployment.apps/whoami-blue created
   deployment.apps/whoami-green created
   ```

   b) Deploy the Gloo Upstream resource:

   ```
   $ kubectl apply -f whoami-gloo-upstream.yaml
   ```

   which gives the following output:

   ```
   upstream.gloo.solo.io/whoami created
   ```

   c) Finally, deploy the VirtualService resource:

   ```
   $ kubectl apply -f whoami-gloo-virtual-service.yaml
   ```

   resulting in this output:

   ```
   virtualservice.gateway.solo.io/whoami created
   ```

6. Let us see if we can run the application. First, we need to find out the target URL. We can do this with the following command:

   ```
   $ glooctl proxy url
   ```

   In our case, the output looks like the following:

   ```
   http://localhost:80
   ```

7. Let us now try to access the application. We can use the preceding URL:

```
$ curl http://localhost:80/
```

and should get an output similar to the following:

```
Hostname: whoami-blue-7d57579d9c-bl4zn, Version: 1.0%
```

This indicates that we are indeed reaching the blue version of the application.

Alternatively, instead of using the hardcoded preceding URL, we can continue using `glooctl proxy` command and remain more generic:

```
$ curl $(glooctl proxy url)/
```

The output should be the same.

8. Repeat the preceding command a few times to make sure that you only ever reach the blue version of the application.

9. Now, let us add the header `stage: canary`, to our request, which should be the criteria to route us to the green version of the application:

   ```
   $ curl $(glooctl proxy url)/ -H "stage: canary"
   ```

   And indeed, we should get an answer like this:

   ```
   Hostname: whoami-green-5d8949d449-7rcrw, Version: 1.1%
   ```

   With this, we have shown how one can deploy and test a staging version using Gloo. By the way, instead of only using a request header to access the staging version of the software, we could also use a more sophisticated filter. The commercial version of Gloo, for example, would allow us to only route authenticated users to the staging version that has a well-defined claim in their JWT token.

10. Great, but now we want to slowly move over traffic from the blue version to the green version. For this, we can slightly modify the second `matchers` section in the `whoami-gloo-virtual-service.yaml` file. Please replace it with this code snippet:

```
- matchers:
    - prefix: /
  routeAction:
    multi:
      destinations:
        - destination:
            upstream:
              name: whoami
              namespace: gloo-system
```

```
$ curl http://localhost:80/
```

and should get an output similar to the following:

```
Hostname: whoami-blue-7d57579d9c-bl4zn, Version: 1.0%
```

This indicates that we are indeed reaching the blue version of the application.

Alternatively, instead of using the hardcoded preceding URL, we can continue using `glooctl proxy` command and remain more generic:

```
$ curl $(glooctl proxy url)/
```

The output should be the same.

8. Repeat the preceding command a few times to make sure that you only ever reach the blue version of the application.

9. Now, let us add the header `stage: canary`, to our request, which should be the criteria to route us to the green version of the application:

   ```
   $ curl $(glooctl proxy url)/ -H "stage: canary"
   ```

   And indeed, we should get an answer like this:

   ```
   Hostname: whoami-green-5d8949d449-7rcrw, Version: 1.1%
   ```

   With this, we have shown how one can deploy and test a staging version using Gloo. By the way, instead of only using a request header to access the staging version of the software, we could also use a more sophisticated filter. The commercial version of Gloo, for example, would allow us to only route authenticated users to the staging version that has a well-defined claim in their JWT token.

10. Great, but now we want to slowly move over traffic from the blue version to the green version. For this, we can slightly modify the second `matchers` section in the `whoami-gloo-virtual-service.yaml` file. Please replace it with this code snippet:

```
- matchers:
    - prefix: /
  routeAction:
    multi:
      destinations:
        - destination:
            upstream:
              name: whoami
              namespace: gloo-system
```

```
      subset:
        values:
          color: blue
    weight: 75
  - destination:
      upstream:
        name: whoami
        namespace: gloo-system
      subset:
        values:
          color: green
    weight: 25
```

We have replaced the `routeAction` from single to multi. Now, we have two destinations instead of one, blue and green. We are using a `weight` to decide how much traffic goes to blue versus how much goes to green. In our case, we funnel 75% of the traffic to blue and the remaining 25% to green.

11. Apply the changes with the following command:

    ```
    $ kubectl apply -f whoami-gloo-virtual-service.yaml
    ```

    giving this output:

    ```
    virtualservice.gateway.solo.io/whoami configured
    ```

12. Test the result by running the following command multiple times:

    ```
    $ curl $(glooctl proxy url)/
    ```

    Make sure that you hit the blue version 3 times more often than the green version.

13. **Optional:** Change the weights to, say, 50 and 50 and apply the changes. Then try to access the whoami application and make sure both services are accessed with the same frequency.

14. When you are done, clean up your cluster with the following commands:

    ```
    $ kubectl delete service/whoami
    $ kubectl delete deploy/whoami-blue
    $ kubectl delete deploy/whoami-green
    $ kubectl delete -f whoami-gloo-upstream.yaml
    ```

```
      subset:
        values:
          color: blue
    weight: 75
  - destination:
      upstream:
        name: whoami
        namespace: gloo-system
      subset:
        values:
          color: green
    weight: 25
```

We have replaced the `routeAction` from single to multi. Now, we have two destinations instead of one, blue and green. We are using a `weight` to decide how much traffic goes to blue versus how much goes to green. In our case, we funnel 75% of the traffic to blue and the remaining 25% to green.

11. Apply the changes with the following command:

    ```
    $ kubectl apply -f whoami-gloo-virtual-service.yaml
    ```

    giving this output:

    ```
    virtualservice.gateway.solo.io/whoami configured
    ```

12. Test the result by running the following command multiple times:

    ```
    $ curl $(glooctl proxy url)/
    ```

    Make sure that you hit the blue version 3 times more often than the green version.

13. **Optional:** Change the weights to, say, 50 and 50 and apply the changes. Then try to access the whoami application and make sure both services are accessed with the same frequency.

14. When you are done, clean up your cluster with the following commands:

    ```
    $ kubectl delete service/whoami
    $ kubectl delete deploy/whoami-blue
    $ kubectl delete deploy/whoami-green
    $ kubectl delete -f whoami-gloo-upstream.yaml
    ```

```
$ kubectl delete -f whoami-gloo-virtual-service.yaml
```

In this section, we showed you how one can create a canary release, either by using the native tools of Kubernetes, such as labels and deployments, or more elegantly, by using a service mesh, such as Gloo. Other well-known service meshes offer similar capabilities. Note that at the time of this writing, Istio, one of the most popular service mesh for Kubernetes, did not run on a Mac with an M1 processor. This is a documented bug.

Conclusion

In this chapter, we have recommended and discussed the use of a sound versioning strategy for your artefacts, such as applications and application services that you run in Kubernetes. The strategy most used in the IT industry is semantic versioning, which we recommend.

Next, we have introduced rolling updates, provided to you by Kubernetes as a default update strategy out of the box. In this strategy, when updating an application service from, say, version 1 to version 1.1, Kubernetes replaces one pod at a time with the new version. The application service remains available all the time if it is running with a replication factor of 2 or higher. High availability requirements anyways dictate that you run your application services with a replication factor of 3 or higher.

Next, we introduced blue-green deployments, another strategy to update an application service. It is easy to achieve on Kubernetes by using labels. We have shown step by step how this works in this chapter.

Finally, we have presented two ways of how to enable canary releases on Kubernetes. Canary releases are a more sophisticated variant of rolling updates. Often canary releases are not just used to upgrade an application service, but canary releases are a popular marketing instrument, allowing to test new features with a small target group and slowly increasing the target group in a controlled way.

The first variant of the canary release implementation we presented just uses out-of-the-box features of Kubernetes, such as labels. The second variant used a service mesh to enable the gradual switching from the old version of the application service to the new one. In our example, we used the Gloo service mesh. But many other service meshes support canary releases too; for example, Istio is another very popular service mesh.

References

- Documentation to Gloo Mesh: **https://www.solo.io/products/gloo-mesh/**

```
$ kubectl delete -f whoami-gloo-virtual-service.yaml
```

In this section, we showed you how one can create a canary release, either by using the native tools of Kubernetes, such as labels and deployments, or more elegantly, by using a service mesh, such as Gloo. Other well-known service meshes offer similar capabilities. Note that at the time of this writing, Istio, one of the most popular service mesh for Kubernetes, did not run on a Mac with an M1 processor. This is a documented bug.

Conclusion

In this chapter, we have recommended and discussed the use of a sound versioning strategy for your artefacts, such as applications and application services that you run in Kubernetes. The strategy most used in the IT industry is semantic versioning, which we recommend.

Next, we have introduced rolling updates, provided to you by Kubernetes as a default update strategy out of the box. In this strategy, when updating an application service from, say, version 1 to version 1.1, Kubernetes replaces one pod at a time with the new version. The application service remains available all the time if it is running with a replication factor of 2 or higher. High availability requirements anyways dictate that you run your application services with a replication factor of 3 or higher.

Next, we introduced blue-green deployments, another strategy to update an application service. It is easy to achieve on Kubernetes by using labels. We have shown step by step how this works in this chapter.

Finally, we have presented two ways of how to enable canary releases on Kubernetes. Canary releases are a more sophisticated variant of rolling updates. Often canary releases are not just used to upgrade an application service, but canary releases are a popular marketing instrument, allowing to test new features with a small target group and slowly increasing the target group in a controlled way.

The first variant of the canary release implementation we presented just uses out-of-the-box features of Kubernetes, such as labels. The second variant used a service mesh to enable the gradual switching from the old version of the application service to the new one. In our example, we used the Gloo service mesh. But many other service meshes support canary releases too; for example, Istio is another very popular service mesh.

References

- Documentation to Gloo Mesh: **https://www.solo.io/products/gloo-mesh/**

Questions

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

1. Which Kubernetes object gives you the ability to execute rolling updates of your applications and application services?
2. Explain what the following snippet from a Kubernetes deployment manifest means:

```
strategy:
type: RollingUpdate
rollingUpdate:
maxUnavailable: 2
maxSurge: 2
```

3. How many Replicasets does Kubernetes use during a rolling update? And what is each of those Replicasets used for?
4. After Kubernetes has finished a rolling update, what does it do with the ReplicaSet that did govern the pods running the old version of the application service?

Answers

Here are the answers to the preceding questions:

1. The Kubernetes `Deployment` object provides the ability to execute rolling updates of an application or an application service.
2. This is a snippet from the definition of a Kubernetes Deployment resource. It defines that Kubernetes should use the rolling update strategy to update the service when requested. Furthermore, the snippet defines that the rolling update shall happen in batches of two pods at a time. `maxSurge: 2` indicates that Kubernetes is allowed to add maximum two new pods containing the new version of the application service to the set of pods. `maxUnavailable: 2`, on the other hand, indicates that Kubernetes shall tear down two pods at max, containing the old version of the application service at a time.
3. Kubernetes uses two Replicasets during a rolling update. The first one is used to control the pods of the old version of the application service, whereas the second ReplicaSet is used to govern the set of pods running the new version of the application service.
4. The ReplicaSet that did govern the set of pods containing the old version of the application set is not automatically deleted by Kubernetes and remains

Questions

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

1. Which Kubernetes object gives you the ability to execute rolling updates of your applications and application services?
2. Explain what the following snippet from a Kubernetes deployment manifest means:

```
strategy:
type: RollingUpdate
rollingUpdate:
maxUnavailable: 2
maxSurge: 2
```

3. How many Replicasets does Kubernetes use during a rolling update? And what is each of those Replicasets used for?
4. After Kubernetes has finished a rolling update, what does it do with the ReplicaSet that did govern the pods running the old version of the application service?

Answers

Here are the answers to the preceding questions:

1. The Kubernetes `Deployment` object provides the ability to execute rolling updates of an application or an application service.
2. This is a snippet from the definition of a Kubernetes Deployment resource. It defines that Kubernetes should use the rolling update strategy to update the service when requested. Furthermore, the snippet defines that the rolling update shall happen in batches of two pods at a time. `maxSurge: 2` indicates that Kubernetes is allowed to add maximum two new pods containing the new version of the application service to the set of pods. `maxUnavailable: 2`, on the other hand, indicates that Kubernetes shall tear down two pods at max, containing the old version of the application service at a time.
3. Kubernetes uses two Replicasets during a rolling update. The first one is used to control the pods of the old version of the application service, whereas the second ReplicaSet is used to govern the set of pods running the new version of the application service.
4. The ReplicaSet that did govern the set of pods containing the old version of the application set is not automatically deleted by Kubernetes and remains

in the system. The number of pods it governs is zero, though. The reason why the ReplicaSet is left there is to make it simple to rollback an update and return to the previous version of the application service with a simple command.

Job Interview sample questions

So, you finally got a chance for an onsite interview for a job as a senior software developer. The job is offered by an established enterprise with a good name in the market. You are excited to have been given a chance for this interview. During the interview, you are confronted with the following questions:

1. Why are rolling updates a good deployment strategy for a mission-critical line of business application?
2. You are tasked to deploy a new version of an application to Kubernetes. The upgrade process needs to be "zero downtime". In a few short sentences, explain what options you have and how you would do this.
3. Please tell us how you would realize a blue-green deployment on Kubernetes. Please list the main steps needed to implement such a blue-green deployment for a simple application service. Avoid going into too much detail.
4. Why would we use canary releases as a deployment strategy? Does the rolling update strategy not cover everything a canary release is about?

Possible answers to interview questions

Here are possible answers to the preceding questions:

1. A mission-critical application cannot have any downtime these days anymore. Thus we need a deployment strategy that guarantees us zero downtime updates of our applications and/or application services. Rolling updates is a deployment strategy that does exactly that, allowing us to rollout a new version whilst the application as such remains operational at all the time.
2. There are various rollout strategies for new versions of application services that do not require to take down the whole platform or application. The three most popular versions of this are as follows:
    a) Rolling updates
    b) Blue-green deployments
    c) Canary releases

in the system. The number of pods it governs is zero, though. The reason why the ReplicaSet is left there is to make it simple to rollback an update and return to the previous version of the application service with a simple command.

Job Interview sample questions

So, you finally got a chance for an onsite interview for a job as a senior software developer. The job is offered by an established enterprise with a good name in the market. You are excited to have been given a chance for this interview. During the interview, you are confronted with the following questions:

1. Why are rolling updates a good deployment strategy for a mission-critical line of business application?
2. You are tasked to deploy a new version of an application to Kubernetes. The upgrade process needs to be "zero downtime". In a few short sentences, explain what options you have and how you would do this.
3. Please tell us how you would realize a blue-green deployment on Kubernetes. Please list the main steps needed to implement such a blue-green deployment for a simple application service. Avoid going into too much detail.
4. Why would we use canary releases as a deployment strategy? Does the rolling update strategy not cover everything a canary release is about?

Possible answers to interview questions

Here are possible answers to the preceding questions:

1. A mission-critical application cannot have any downtime these days anymore. Thus we need a deployment strategy that guarantees us zero downtime updates of our applications and/or application services. Rolling updates is a deployment strategy that does exactly that, allowing us to rollout a new version whilst the application as such remains operational at all the time.
2. There are various rollout strategies for new versions of application services that do not require to take down the whole platform or application. The three most popular versions of this are as follows:
    a) Rolling updates
    b) Blue-green deployments
    c) Canary releases

Rolling updates are the default strategy of Kubernetes. We can achieve this by using Kubernetes Deployment resources and configuring them accordingly

Blue-green deployments can easily be achieved in Kubernetes by using labels, e.g., `color: blue` versus `color: green`. The Kubernetes service that does proxy the incoming requests to the pods can then be instructed to select either the blue or the green pods with a simple update.

3. As mentioned in answer 2, a blue-green deployment in Kubernetes can be achieved as follows:
   a) Use a Deployment object for each blue and green version of the application service
   b) Provide a label `color` with the value `blue` and `green`, respectively, in the definition of Deployment
   c) Define the Kubernetes service working as a proxy for the deployment. In the selector of the service, definition set one of the criteria to be the label `color: blue`. This means that the Kubernetes service will funnel all traffic to the old—that is blue—version of the application service. When ready, modify the service and set the value of the `color` filter to `green`. The Kubernetes service will now funnel all traffic to the new, green version of the application service.
4. Rolling updates are controlled by Kubernetes and leave us limited space for external control. Canary releases are basically also rolling updates, but they give us more fine-grained control over what shall happen during the update. Specifically, we can define our own criteria that determine how the gradual switch over of the traffic from blue to green shall happen. We can develop a feedback loop that measures some KPIs over some time, and in case of success, we can turn the know a notch forward.

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



Rolling updates are the default strategy of Kubernetes. We can achieve this by using Kubernetes Deployment resources and configuring them accordingly

Blue-green deployments can easily be achieved in Kubernetes by using labels, e.g., `color: blue` versus `color: green`. The Kubernetes service that does proxy the incoming requests to the pods can then be instructed to select either the blue or the green pods with a simple update.

3. As mentioned in answer 2, a blue-green deployment in Kubernetes can be achieved as follows:
   a) Use a Deployment object for each blue and green version of the application service
   b) Provide a label `color` with the value `blue` and `green`, respectively, in the definition of Deployment
   c) Define the Kubernetes service working as a proxy for the deployment. In the selector of the service, definition set one of the criteria to be the label `color: blue`. This means that the Kubernetes service will funnel all traffic to the old—that is blue—version of the application service. When ready, modify the service and set the value of the `color` filter to `green`. The Kubernetes service will now funnel all traffic to the new, green version of the application service.
4. Rolling updates are controlled by Kubernetes and leave us limited space for external control. Canary releases are basically also rolling updates, but they give us more fine-grained control over what shall happen during the update. Specifically, we can define our own criteria that determine how the gradual switch over of the traffic from blue to green shall happen. We can develop a feedback loop that measures some KPIs over some time, and in case of success, we can turn the know a notch forward.

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 13
Securing Containers

Introduction

Containers are enablers of a secure software supply chain based on encapsulation (using Linux namespaces, and cgroups), secrets, content trust, and image vulnerability scanning.

In this chapter, we will introduce Kubernetes secrets as a means 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. Next, 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. We will also provide a short list of Docker security best practices. Last but not least, the reader is presented a list of possible container security-related job interview questions and good responses to each of those questions.

We will be using Docker Desktop, Kubernetes, the terminal, and VS Code in this chapter. Furthermore, we will use **Docker Content Trust (DCT)** to trust and Snyk to scan images.

CHAPTER 13
Securing Containers

Introduction

Containers are enablers of a secure software supply chain based on encapsulation (using Linux namespaces, and cgroups), secrets, content trust, and image vulnerability scanning.

In this chapter, we will introduce Kubernetes secrets as a means 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. Next, 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. We will also provide a short list of Docker security best practices. Last but not least, the reader is presented a list of possible container security-related job interview questions and good responses to each of those questions.

We will be using Docker Desktop, Kubernetes, the terminal, and VS Code in this chapter. Furthermore, we will use **Docker Content Trust** (DCT) to trust and Snyk to scan images.

Structure

In this chapter, we will discuss the following topics:

- Using Kubernetes secrets
- Trusting containers
- Scanning container images for vulnerabilities and exposures
- Firewalling applications running in containers
- Discussing Docker security best practices

Objectives

After reading this chapter, you will be able to define and use secrets, such as passwords or API key secrets, with applications running in containers on Kubernetes. You will also be able to trust the containers that you will be running in production and thus making man-in-the-middle type of attacks impossible, that try to alter an existing Docker image in transit. You will also learn how you can further secure your applications by establishing software-defined networks and by mapping ports and protocols according to your need. Additionally, you will understand how you can scan your Docker images for common exposure and vulnerabilities, in short called CVEs. Finally, you will be able to create a list of at least three to four Docker security best practices.

Let us start with using secrets.

Using Kubernetes secrets

Modern applications often require access to protected resources such as databases, blob storage, or third-party APIs. In this case, the application needs to authenticate itself to the resource provider. If the resource provider is a database, then the application usually does this by using a connection string. This connection string may contain a secret, such as a password. We all know that it is an anti-pattern to provide such a secret in clear text to the application, no matter if the secret is hardcoded in the source code of the application or if it is provided in clear text through a configuration file or an environment variable.

Ok, that said, what are we supposed to do instead? The answer is to use Kubernetes secrets.

Explaining Kubernetes secrets

When we want to use passwords, API key secrets or certificates on Kubernetes, we want to make sure that this sensitive information can only ever be seen and accessed

Structure

In this chapter, we will discuss the following topics:

- Using Kubernetes secrets
- Trusting containers
- Scanning container images for vulnerabilities and exposures
- Firewalling applications running in containers
- Discussing Docker security best practices

Objectives

After reading this chapter, you will be able to define and use secrets, such as passwords or API key secrets, with applications running in containers on Kubernetes. You will also be able to trust the containers that you will be running in production and thus making man-in-the-middle type of attacks impossible, that try to alter an existing Docker image in transit. You will also learn how you can further secure your applications by establishing software-defined networks and by mapping ports and protocols according to your need. Additionally, you will understand how you can scan your Docker images for common exposure and vulnerabilities, in short called CVEs. Finally, you will be able to create a list of at least three to four Docker security best practices.

Let us start with using secrets.

Using Kubernetes secrets

Modern applications often require access to protected resources such as databases, blob storage, or third-party APIs. In this case, the application needs to authenticate itself to the resource provider. If the resource provider is a database, then the application usually does this by using a connection string. This connection string may contain a secret, such as a password. We all know that it is an anti-pattern to provide such a secret in clear text to the application, no matter if the secret is hardcoded in the source code of the application or if it is provided in clear text through a configuration file or an environment variable.

Ok, that said, what are we supposed to do instead? The answer is to use Kubernetes secrets.

Explaining Kubernetes secrets

When we want to use passwords, API key secrets or certificates on Kubernetes, we want to make sure that this sensitive information can only ever be seen and accessed

by the authorized or dedicated service. All the other services that are running on the same cluster should not be able to access these secrets.

This is why the folks around Kubernetes decided to introduce Kubernetes secrets. On Kubernetes, a secret is a key-value pair, where the key is a unique name for the secret, and the value contains the actual sensitive data. Kubernetes will store the secrets in etcd. Remember, etcd is a distributed key-value store that is used to manage all data about the Kubernetes cluster.

Kubernetes can be configured in a way that the secrets are encrypted at rest (in etcd) and in transit, that is, when the secret is transmitted over the wire from a master node to the worker nodes where the pods run that consume the secret.

We have two ways to define a secret, declaratively and imperatively.

Declaratively defining secrets

We can create a Kubernetes secret declaratively using a JSON or YAML file, the same way as we define any other object on Kubernetes. Here is a sample YAML file that can be used to define a secret called **my-secret**:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: am9obi5kb2UK
  password: c0VjcmV0LXBhc1N3MHJECg==
```

Let us decipher the values and describe what they are. In the **metadata** block, we have the **name** of the secret, in this case, **my-secret**. Then, we have the **data** block, which contains the actual data of the secret, that is, the sensitive part. In our case, it is a short list of key-value pairs where the value part looks as if it had been encrypted. But unfortunately, this is not the case. The values are just the base64-encoded values of the original values. It is easy to prove this by decoding those values using the base64 tool on your laptop:

```
$ echo "am9obi5kb2UK" | base64 --decode
john.doe
$ echo "c0VjcmV0LXBhc1N3MHJECg==" | base64 --decode
sEcret-pasSw0rD
```

by the authorized or dedicated service. All the other services that are running on the same cluster should not be able to access these secrets.

This is why the folks around Kubernetes decided to introduce Kubernetes secrets. On Kubernetes, a secret is a key-value pair, where the key is a unique name for the secret, and the value contains the actual sensitive data. Kubernetes will store the secrets in etcd. Remember, etcd is a distributed key-value store that is used to manage all data about the Kubernetes cluster.

Kubernetes can be configured in a way that the secrets are encrypted at rest (in etcd) and in transit, that is, when the secret is transmitted over the wire from a master node to the worker nodes where the pods run that consume the secret.

We have two ways to define a secret, declaratively and imperatively.

Declaratively defining secrets

We can create a Kubernetes secret declaratively using a JSON or YAML file, the same way as we define any other object on Kubernetes. Here is a sample YAML file that can be used to define a secret called **my-secret**:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: am9obi5kb2UK
  password: c0VjcmV0LXBhc1N3MHJECg==
```

Let us decipher the values and describe what they are. In the **metadata** block, we have the **name** of the secret, in this case, **my-secret**. Then, we have the **data** block, which contains the actual data of the secret, that is, the sensitive part. In our case, it is a short list of key-value pairs where the value part looks as if it had been encrypted. But unfortunately, this is not the case. The values are just the base64-encoded values of the original values. It is easy to prove this by decoding those values using the base64 tool on your laptop:

```
$ echo "am9obi5kb2UK" | base64 --decode
john.doe
$ echo "c0VjcmV0LXBhc1N3MHJECg==" | base64 --decode
sEcret-pasSw0rD
```

Given this, it is not recommended to use the declarative definition of secrets in production but rather rely on a more secure way, such as imperatively defining secrets. Before we do that, though, let us provide the complete recipe for how to create a secret on Kubernetes using this declarative method:

1. Use a tool such as `base64` to encode the values for the username and password:

```
$ echo "sue.hunter" | base64
c3VlLmh1bnRlcgo=
$ echo "Top-Secret" | base64
VG9wLVNlY3JldAo=
```

2. Add the generated values to your YAML file. Call the file `my-secret.yaml`:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: c3VlLmh1bnRlcgo=
  password: VG9wLVNlY3JldAo=
```

3. Create the secret using `kubectl`:

```
$ kubectl create -f my-secret.yaml
secret/my-secret created
```

4. Describe the secret, also using `kubectl`:

```
$ kubectl describe secret/my-secret
Name:         my-secret
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
```

Given this, it is not recommended to use the declarative definition of secrets in production but rather rely on a more secure way, such as imperatively defining secrets. Before we do that, though, let us provide the complete recipe for how to create a secret on Kubernetes using this declarative method:

1. Use a tool such as `base64` to encode the values for the username and password:

```
$ echo "sue.hunter" | base64
c3VlLmh1bnRlcgo=
$ echo "Top-Secret" | base64
VG9wLVNlY3JldAo=
```

2. Add the generated values to your YAML file. Call the file `my-secret.yaml`:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: c3VlLmh1bnRlcgo=
  password: VG9wLVNlY3JldAo=
```

3. Create the secret using `kubectl`:

```
$ kubectl create -f my-secret.yaml
secret/my-secret created
```

4. Describe the secret, also using `kubectl`:

```
$ kubectl describe secret/my-secret
Name:         my-secret
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
```

```
====
password:  11 bytes
username:  11 bytes
```

5. In the description of the secret, the values are hidden, and only their length is given. So, maybe the secrets are safe now? No, not really. We can easily decode this secret using the `kubectl get` command (do not forget the output parameter `-o yaml`):

   ```
   $ kubectl get secret/my-secret -o yaml
   ```

   Here is the output:

```
apiVersion: v1
data:
  password: VG9wLVNlY3JldAo=
  username: c3VlLmh1bnRlcgo=
kind: Secret
metadata:
  creationTimestamp: "2022-07-28T08:53:37Z"
  name: my-secret
  namespace: default
  resourceVersion: "116368"
  uid: e90a0ca0-84a7-4618-a188-0711c17d8729
type: Opaque
```

Figure 13.1: Output of the kubectl get command

6. Decode the values using `base64`, as described earlier.
7. Once you are done experimenting with the secret, you can delete it as follows:

   ```
   $ kubectl delete secret/my-secret
   secret "my-secret" deleted
   ```

Now let us have a look at how we can imperatively define a secret on Kubernetes.

Imperatively defining secrets

To imperatively define secrets, we can use our trusted `kubectl` command line tool. This method is often used while experimenting with secrets and also by system admins to define secrets on a production cluster. This way, only the admin will have to know the sensitive data of the secret.

Let us see how this works:

1. Create the values for the secrets you want to define, in this case, once again `username` and `password`, and store them each in a file:

   ```
   $ echo -n "john.doe" > ./username.txt
   $ echo -n "c0nFident1al" > ./password.txt
   ```

```
====
password:  11 bytes
username:  11 bytes
```

5. In the description of the secret, the values are hidden, and only their length is given. So, maybe the secrets are safe now? No, not really. We can easily decode this secret using the `kubectl get` command (do not forget the output parameter `-o yaml`):

   ```
   $ kubectl get secret/my-secret -o yaml
   ```

   Here is the output:

```
apiVersion: v1
data:
  password: VG9wLVNlY3JldAo=
  username: c3VlLmh1bnRlcgo=
kind: Secret
metadata:
  creationTimestamp: "2022-07-28T08:53:37Z"
  name: my-secret
  namespace: default
  resourceVersion: "116368"
  uid: e90a0ca0-84a7-4618-a188-0711c17d8729
type: Opaque
```

Figure 13.1: Output of the kubectl get command

6. Decode the values using `base64`, as described earlier.
7. Once you are done experimenting with the secret, you can delete it as follows:

   ```
   $ kubectl delete secret/my-secret
   secret "my-secret" deleted
   ```

Now let us have a look at how we can imperatively define a secret on Kubernetes.

Imperatively defining secrets

To imperatively define secrets, we can use our trusted `kubectl` command line tool. This method is often used while experimenting with secrets and also by system admins to define secrets on a production cluster. This way, only the admin will have to know the sensitive data of the secret.

Let us see how this works:

1. Create the values for the secrets you want to define, in this case, once again `username` and `password`, and store them each in a file:

   ```
   $ echo -n "john.doe" > ./username.txt
   $ echo -n "c0nFident1al" > ./password.txt
   ```

2. Use `kubectl` to create a secret named `my-other-secret` using the values in the files:

```
$ kubectl create secret generic my-other-secret \
  --from-file=username=./username.txt \
  --from-file=password=./password.txt
secret/my-other-secret created
```

   Note the `--from-file` parameter that allows you to add 1 to many key-value pairs to a secret, where the value is retrieved from a file.

3. If we want to use literal values instead of files, then we can use a slightly modified version of the command, using the parameter `--from-literal` to define the key-value pairs:

```
$ kubectl create secret generic my-third-secret \
  --from-literal=username=sally.king \
  --from-literal=password='SillyPassw0rd123'
secret/my-third-secret created
```

4. Use the following command to list all secrets defined in the cluster:

```
$ kubectl get secrets
NAME              TYPE     DATA   AGE
my-other-secret   Opaque   2      6m1s
my-third-secret   Opaque   2      61s
```

5. Use the describe command to get the details of a secret, such as:

```
$ kubectl describe secrets/my-other-secret
Name:         my-other-secret
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
password:  12 bytes
username:  8 bytes
```

2. Use `kubectl` to create a secret named `my-other-secret` using the values in the files:

```
$ kubectl create secret generic my-other-secret \
  --from-file=username=./username.txt \
  --from-file=password=./password.txt
secret/my-other-secret created
```

   Note the `--from-file` parameter that allows you to add 1 to many key-value pairs to a secret, where the value is retrieved from a file.

3. If we want to use literal values instead of files, then we can use a slightly modified version of the command, using the parameter `--from-literal` to define the key-value pairs:

```
$ kubectl create secret generic my-third-secret \
  --from-literal=username=sally.king \
  --from-literal=password='SillyPassw0rd123'
secret/my-third-secret created
```

4. Use the following command to list all secrets defined in the cluster:

```
$ kubectl get secrets
NAME              TYPE     DATA   AGE
my-other-secret   Opaque   2      6m1s
my-third-secret   Opaque   2      61s
```

5. Use the describe command to get the details of a secret, such as:

```
$ kubectl describe secrets/my-other-secret
Name:         my-other-secret
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
password:  12 bytes
username:  8 bytes
```

6. And use the get command to retrieve even the base64 encoded values of the secret:

```
$ kubectl get secrets/my-other-secret -o yaml
apiVersion: v1
data:
  password: YzBuRmlkZW50MWFs
  username: am9obi5kb2U=
kind: Secret
metadata:
  creationTimestamp: "2022-08-28T16:04:12Z"
  name: my-other-secret
  namespace: default
  resourceVersion: "285113"
  uid: 67c885f1-2ef1-4ff2-9e93-8500a65b7307
type: Opaque
```

7. When done, delete your secrets with the following:

```
$ kubectl delete secrets/my-other-secret
$ kubectl delete secrets/my-third-secret
```

And that is it about how to define secrets in a Kubernetes cluster. But let me add one word of caution, though. Earlier, we said that system administrators often use `kubectl` to define secrets in the production Kubernetes cluster. Although this is a somewhat safe method, it is by no means the recommended one. These days configuration management should be fully automated. And defining and managing secrets is part of configuration management. Keep that in mind.

Next, we are going to show how you can use those secrets with applications running inside a pod.

Using secrets in a pod

Now that we know how to imperatively and declaratively define secrets on a Kubernetes cluster, let us discuss how to actually get those secret values to the application running inside a container in a pod. Let us start by showing how we can map secrets to the container's filesystem.

6. And use the get command to retrieve even the base64 encoded values of the secret:

```
$ kubectl get secrets/my-other-secret -o yaml
apiVersion: v1
data:
  password: YzBuRmlkZW50MWFs
  username: am9obi5kb2U=
kind: Secret
metadata:
  creationTimestamp: "2022-08-28T16:04:12Z"
  name: my-other-secret
  namespace: default
  resourceVersion: "285113"
  uid: 67c885f1-2ef1-4ff2-9e93-8500a65b7307
type: Opaque
```

7. When done, delete your secrets with the following:

```
$ kubectl delete secrets/my-other-secret
$ kubectl delete secrets/my-third-secret
```

And that is it about how to define secrets in a Kubernetes cluster. But let me add one word of caution, though. Earlier, we said that system administrators often use `kubectl` to define secrets in the production Kubernetes cluster. Although this is a somewhat safe method, it is by no means the recommended one. These days configuration management should be fully automated. And defining and managing secrets is part of configuration management. Keep that in mind.

Next, we are going to show how you can use those secrets with applications running inside a pod.

Using secrets in a pod

Now that we know how to imperatively and declaratively define secrets on a Kubernetes cluster, let us discuss how to actually get those secret values to the application running inside a container in a pod. Let us start by showing how we can map secrets to the container's filesystem.

Mapping secrets to the filesystem

Let us say we want to create a Deployment object where the Web component uses our secret, `my-secret`, that we introduced in the preceding section. We can use the following command to create the secret in the cluster:

```
$ kubectl apply -f my-secret.yaml
```

In the `.../ch13/whoami-deployment-secret.yaml` file, we can find the definition of the Deployment object. We had to add the part starting from Line 23 to the original definition of the Deployment object from *Chapter 12, Enabling zero downtime deployments*.

whoami-deployment-secret.yml > {} spec > {} ten

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: whoami-deployment
5  spec:
6    replicas: 3
7    selector:
8      matchLabels:
9        app: whoami
10   template:
11     metadata:
12       labels:
13         app: whoami
14     spec:
15       containers:
16       - name: whoami
17         image: gnschenker/whoami:1.0
18         ports:
19         - containerPort: 80
20         volumeMounts:
21         - name: secrets
22           mountPath: "/etc/secrets"
23           readOnly: true
24       volumes:
25       - name: secrets
26         secret:
27           secretName: my-secret
```

Figure 13.2: Mounting a secret with a volume mount

In Lines 24–27, we define a volume called `secrets` from our secret, `my-secret`. Then, we use this volume in the container, as described in Lines 20–23. We mount the secrets in the container filesystem at `/etc/secrets`, and we mount the volume in `read-only` mode. Thus, the secret values will be available to the container as files in the chosen folder. The names of the files will correspond to the key names, and the content of the files will be the values of the corresponding keys. The values will be provided in unencrypted form to the application running inside the container.

Mapping secrets to the filesystem

Let us say we want to create a Deployment object where the Web component uses our secret, `my-secret`, that we introduced in the preceding section. We can use the following command to create the secret in the cluster:

```
$ kubectl apply -f my-secret.yaml
```

In the `.../ch13/whoami-deployment-secret.yaml` file, we can find the definition of the Deployment object. We had to add the part starting from Line 23 to the original definition of the Deployment object from *Chapter 12, Enabling zero downtime deployments*.

whoami-deployment-secret.yml > {} spec > {} ten

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: whoami-deployment
5  spec:
6    replicas: 3
7    selector:
8      matchLabels:
9        app: whoami
10   template:
11     metadata:
12       labels:
13         app: whoami
14     spec:
15       containers:
16       - name: whoami
17         image: gnschenker/whoami:1.0
18         ports:
19         - containerPort: 80
20         volumeMounts:
21         - name: secrets
22           mountPath: "/etc/secrets"
23           readOnly: true
24       volumes:
25       - name: secrets
26         secret:
27           secretName: my-secret
```

Figure 13.2: Mounting a secret with a volume mount

In Lines 24–27, we define a volume called `secrets` from our secret, `my-secret`. Then, we use this volume in the container, as described in Lines 20–23. We mount the secrets in the container filesystem at `/etc/secrets`, and we mount the volume in `read-only` mode. Thus, the secret values will be available to the container as files in the chosen folder. The names of the files will correspond to the key names, and the content of the files will be the values of the corresponding keys. The values will be provided in unencrypted form to the application running inside the container.

In our case, since we have the **username** and **password** keys in secret, we will find two files, named **username** and **password**, in the **/etc/secrets** folder in the container's filesystem. The **username** file should contain the value **john.doe**, and the **password** file should contain the value **Top-Secret**. Let us try this out:

1. Create the secret **my-secret** with the following:

```
$ kubectl apply -f my-secret.yaml
secret/my-secret created
```

2. Create the deployment using the secret with the following:

```
$ kubectl apply -f whoami-deployment-secret.yaml
deployment.apps/whoami-deployment created
```

3. List all the running pods:

```
$ kubectl get pods
NAME                                READY   STATUS    RESTARTS      AGE
whoami-deployment-695f85cbd9-5rfdr  1/1     Running   0             4m52s
whoami-deployment-695f85cbd9-sh4kj  1/1     Running   0             4m52s
whoami-deployment-695f85cbd9-vck9v  1/1     Running   0             4m52s
```

4. Select one of the pods, in our case, the first one and use its name in the following command:

5. Run a Linux shell inside the first container with the following:

```
$ kubectl exec -it whoami-deployment-695f85cbd9-5rfdr -- /bin/sh
/app #
```

6. List all files in the secrets folder:

```
$ ls -al /etc/secrets
total 0
lrwxrwxrwx    1 root     root      15 Aug 27 10:05 password ->
..data/password
lrwxrwxrwx    1 root     root      15 Aug 27 10:05 username ->
..data/username
```

7. Uncover the content of the two files:

```
$ cat /etc/secrets/username && cat /etc/secrets/password
sue.hunter
Top-Secret
```

In our case, since we have the **username** and **password** keys in secret, we will find two files, named **username** and **password**, in the **/etc/secrets** folder in the container's filesystem. The **username** file should contain the value **john.doe**, and the **password** file should contain the value **Top-Secret**. Let us try this out:

1. Create the secret **my-secret** with the following:

```
$ kubectl apply -f my-secret.yaml
secret/my-secret created
```

2. Create the deployment using the secret with the following:

```
$ kubectl apply -f whoami-deployment-secret.yaml
deployment.apps/whoami-deployment created
```

3. List all the running pods:

```
$ kubectl get pods
NAME                                READY   STATUS    RESTARTS      AGE
whoami-deployment-695f85cbd9-5rfdr  1/1     Running   0             4m52s
whoami-deployment-695f85cbd9-sh4kj  1/1     Running   0             4m52s
whoami-deployment-695f85cbd9-vck9v  1/1     Running   0             4m52s
```

4. Select one of the pods, in our case, the first one and use its name in the following command:

5. Run a Linux shell inside the first container with the following:

```
$ kubectl exec -it whoami-deployment-695f85cbd9-5rfdr -- /bin/sh
/app #
```

6. List all files in the secrets folder:

```
$ ls -al /etc/secrets
total 0
lrwxrwxrwx    1 root     root      15 Aug 27 10:05 password ->
..data/password
lrwxrwxrwx    1 root     root      15 Aug 27 10:05 username ->
..data/username
```

7. Uncover the content of the two files:

```
$ cat /etc/secrets/username && cat /etc/secrets/password
sue.hunter
Top-Secret
```

As you can see previously, the files contain the expected values in clear form, ready to be used by the application running inside the container

8. To leave the shell inside the pod, hit *CTRL-d*.

9. To clean up the environment when done, use the following:

```
$ kubectl delete deployments/whoami-deployment

deployment.apps "whoami-deployment" deleted

$ kubectl delete secrets/my-secret
```

secret "my-secret" deleted

Now that we have seen how to map secrets as files into a pod, we will learn how to do the same using environment variables.

Secrets and environment variables

Let us say our **whoami** component from the previous chapter expects the **username** in the environment variable, **WHOAMI_USERNAME**, and the **password** in **WHOAMI_PASSWORD**. If this is the case, we can modify our deployment YAML so that it looks as follows:

whoami-deployment.yml > apiVersion

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: whoami-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: whoami
  template:
    metadata:
      labels:
        app: whoami
    spec:
      containers:
      - name: whoami
        image: gnschenker/whoami:1.0
        ports:
        - containerPort: 80
        env:
        - name: WHOAMI_USERNAME
          valueFrom:
            secretKeyRef:
              name: my-secret
              key: username
        - name: WHOAMI_PASSWORD
          valueFrom:
            secretKeyRef:
              name: my-secret
              key: password
```

Figure 13.3: Secrets through environment variables

As you can see previously, the files contain the expected values in clear form, ready to be used by the application running inside the container

8. To leave the shell inside the pod, hit *CTRL-d*.

9. To clean up the environment when done, use the following:

```
$ kubectl delete deployments/whoami-deployment

deployment.apps "whoami-deployment" deleted

$ kubectl delete secrets/my-secret
```

secret "my-secret" deleted

Now that we have seen how to map secrets as files into a pod, we will learn how to do the same using environment variables.

Secrets and environment variables

Let us say our **whoami** component from the previous chapter expects the **username** in the environment variable, **WHOAMI_USERNAME**, and the **password** in **WHOAMI_PASSWORD**. If this is the case, we can modify our deployment YAML so that it looks as follows:

whoami-deployment.yml > apiVersion

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: whoami-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: whoami
  template:
    metadata:
      labels:
        app: whoami
    spec:
      containers:
      - name: whoami
        image: gnschenker/whoami:1.0
        ports:
        - containerPort: 80
        env:
        - name: WHOAMI_USERNAME
          valueFrom:
            secretKeyRef:
              name: my-secret
              key: username
        - name: WHOAMI_PASSWORD
          valueFrom:
            secretKeyRef:
              name: my-secret
              key: password
```

Figure 13.3: Secrets through environment variables

In Lines 20–30, we define the two environment variables, **WHOAMI_USERNAME** and **WHOAMI_PASSWORD**, and map the corresponding key-value pair of **my-secret** to them.

We have stored the previous *yaml* in a file called **.../ch13/whoami-deployment-env-secret.yaml**. Note that we do not need a volume anymore; instead, we directly map the individual keys of our **my-secret** into the corresponding environment variables that are valid inside the container. The following sequence of commands shows that the secret values are indeed available inside the container in the respective environment variables:

10. Create the secret my-secret with the following:

```
$ kubectl apply -f my-secret.yaml
secret/my-secret created
```

11. Create the deployment using the secret with the following:

```
$ kubectl apply -f whoami-deployment-env-secret.yaml
deployment.apps/whoami-deployment created
```

12. List all the running pods:

```
$ kubectl get pods
NAME                                READY   STATUS    RESTARTS      AGE
whoami-deployment-57954b8859-fbfw7  1/1     Running   0             4m52s
whoami-deployment-57954b8859-rk96l  1/1     Running   0             4m52s
whoami-deployment-57954b8859-xzvj2  1/1     Running   0             4m52s
```

13. Select one of the pods, in our case, the first one and use its name in the following command:

14. Run a Linux shell inside the first container with the following:

```
$ kubectl exec -it whoami-deployment-57954b8859-fbfw7 -- /bin/sh
/app #
```

15. Use the echo command to display the content of the two environment variables **WHOAMI_USERNAME** and **WHOAMI_PASSWORD**:

```
/app # echo $WHOAMI_USERNAME && echo $WHOAMI_PASSWORD
sue.hunter
Top-Secret
```

In Lines 20–30, we define the two environment variables, **WHOAMI_USERNAME** and **WHOAMI_PASSWORD**, and map the corresponding key-value pair of **my-secret** to them.

We have stored the previous *yaml* in a file called **.../ch13/whoami-deployment-env-secret.yaml**. Note that we do not need a volume anymore; instead, we directly map the individual keys of our **my-secret** into the corresponding environment variables that are valid inside the container. The following sequence of commands shows that the secret values are indeed available inside the container in the respective environment variables:

10. Create the secret my-secret with the following:

```
$ kubectl apply -f my-secret.yaml
secret/my-secret created
```

11. Create the deployment using the secret with the following:

```
$ kubectl apply -f whoami-deployment-env-secret.yaml
deployment.apps/whoami-deployment created
```

12. List all the running pods:

```
$ kubectl get pods
NAME                                READY   STATUS    RESTARTS      AGE
whoami-deployment-57954b8859-fbfw7  1/1     Running   0             4m52s
whoami-deployment-57954b8859-rk96l  1/1     Running   0             4m52s
whoami-deployment-57954b8859-xzvj2  1/1     Running   0             4m52s
```

13. Select one of the pods, in our case, the first one and use its name in the following command:

14. Run a Linux shell inside the first container with the following:

```
$ kubectl exec -it whoami-deployment-57954b8859-fbfw7 -- /bin/sh
/app #
```

15. Use the echo command to display the content of the two environment variables **WHOAMI_USERNAME** and **WHOAMI_PASSWORD**:

```
/app # echo $WHOAMI_USERNAME && echo $WHOAMI_PASSWORD
sue.hunter
Top-Secret
```

16. As you can see, the environment variables contain the expected values, and thus, your application running inside the container can use them.

17. Leave the shell by pressing *CTRL-d*.

18. Clean up Kubernetes with the following:

```
$ kubectl delete deployments/whoami-deployment
deployment.apps "whoami-deployment" deleted
$ kubectl delete secrets/my-secret
secret "my-secret" deleted
```

In the next section, we are going to learn what we can do to trust containers.

Trusting containers

For additional security, we want to make sure that only trusted images run in our production cluster. Some orchestrators allow us to configure a cluster so that it can only ever run signed images. Content trust and signing images are all about making sure that the authors of the image are the ones that we expect them to be, namely, our trusted developers or, even better, our trusted deployment server. Furthermore, with content trust, we want to guarantee that the image that we get is fresh and is not an old one and thus potentially a vulnerable image. And finally, we want to make sure that the image cannot be compromised by malicious hackers in transit. The latter is often called a **man-in-the-middle (MITM)** attack.

By signing images at the source and validating the signature at the target, we can guarantee that the images that we want to run are not compromised.

Container images can be signed by individuals and organizations and also often by an automation agent that is part of the fully automated CI/CD pipeline of a company. This agent is usually associated with a technical user defined in the system.

When working with containers, we can rely on **Docker Content Trust (DCT)** to do the heavy lifting for us. The publisher of a Docker image is normally the one that is signing it. As mentioned previously, this can be individuals, organizations, or automation tasks. Let us look at how a Docker image is usually defined. Its full name consists of the following parts:

```
<registry-url>/<organization>/<repository-name>:<tag>
```

When we use content trust with containers, we are digitally signing the last part of the previous, namely a `tag`. Thus, in a repository, we can have versions of the image that are digitally signed and others that are not.

16. As you can see, the environment variables contain the expected values, and thus, your application running inside the container can use them.

17. Leave the shell by pressing *CTRL-d*.

18. Clean up Kubernetes with the following:

```
$ kubectl delete deployments/whoami-deployment
deployment.apps "whoami-deployment" deleted
$ kubectl delete secrets/my-secret
secret "my-secret" deleted
```

In the next section, we are going to learn what we can do to trust containers.

Trusting containers

For additional security, we want to make sure that only trusted images run in our production cluster. Some orchestrators allow us to configure a cluster so that it can only ever run signed images. Content trust and signing images are all about making sure that the authors of the image are the ones that we expect them to be, namely, our trusted developers or, even better, our trusted deployment server. Furthermore, with content trust, we want to guarantee that the image that we get is fresh and is not an old one and thus potentially a vulnerable image. And finally, we want to make sure that the image cannot be compromised by malicious hackers in transit. The latter is often called a **man-in-the-middle (MITM)** attack.

By signing images at the source and validating the signature at the target, we can guarantee that the images that we want to run are not compromised.

Container images can be signed by individuals and organizations and also often by an automation agent that is part of the fully automated CI/CD pipeline of a company. This agent is usually associated with a technical user defined in the system.

When working with containers, we can rely on **Docker Content Trust (DCT)** to do the heavy lifting for us. The publisher of a Docker image is normally the one that is signing it. As mentioned previously, this can be individuals, organizations, or automation tasks. Let us look at how a Docker image is usually defined. Its full name consists of the following parts:

```
<registry-url>/<organization>/<repository-name>:<tag>
```

When we use content trust with containers, we are digitally signing the last part of the previous, namely a `tag`. Thus, in a repository, we can have versions of the image that are digitally signed and others that are not.

To work with DCT on a Docker host, we need to configure the environment variable `DOCKER_CONTENT_TRUST` to 1, for example, like the following:

```
$ export DOCKER_CONTENT_TRUST=1
```

From this moment on, we can only work with signed images. Any attempt to pull an unsigned Docker image from the registry or to run an unsigned image will result in an error.

If somebody tries to temper with a signed image in transit, the Docker client will throw an error and abort the operation when we are, for example, pulling a signed image from the image registry. This is exactly what we want.

The technical details on how this whole DCT works are relatively complicated and are outside of the scope of this book. There is, though, very good and complete documentation about DCT available in the Docker online documentation. Please refer to this page for a start: **https://docs.docker.com/engine/security/trust/**.

Let us now proceed with the next security topic, vulnerabilities and exposures in Docker images, and how to detect and eliminate them.

Scanning container images for vulnerabilities

In this section, we are going to use Snyk to scan a Docker image for vulnerabilities. Snyk should be included with your Docker Desktop installation. Check by using the following command:

```
$ docker scan --version
Version:    v0.17.0
Git commit: 061fe0a
Provider:   Snyk (1.827.0)
```

Note you do not have to be logged in to your Docker account to use Snyk, but in this case, you can only scan a limited number of times. If you are logged in, on the other hand, you can do up to 200 scans per month for free. Anything beyond that number requires a paid account with Snyk.

Now, what are vulnerabilities, or more precisely CVEs, where CVE stands for Common Vulnerabilities and Exposures?

Common Vulnerabilities and Exposures is a database of publicly disclosed information security issues. Each entry in that database has a number, and that number uniquely identifies one vulnerability from the list. Specialists permanently curate this list and add new vulnerabilities or exposures as quickly as they are discovered.

To work with DCT on a Docker host, we need to configure the environment variable `DOCKER_CONTENT_TRUST` to 1, for example, like the following:

```
$ export DOCKER_CONTENT_TRUST=1
```

From this moment on, we can only work with signed images. Any attempt to pull an unsigned Docker image from the registry or to run an unsigned image will result in an error.

If somebody tries to temper with a signed image in transit, the Docker client will throw an error and abort the operation when we are, for example, pulling a signed image from the image registry. This is exactly what we want.

The technical details on how this whole DCT works are relatively complicated and are outside of the scope of this book. There is, though, very good and complete documentation about DCT available in the Docker online documentation. Please refer to this page for a start: **https://docs.docker.com/engine/security/trust/**.

Let us now proceed with the next security topic, vulnerabilities and exposures in Docker images, and how to detect and eliminate them.

Scanning container images for vulnerabilities

In this section, we are going to use Snyk to scan a Docker image for vulnerabilities. Snyk should be included with your Docker Desktop installation. Check by using the following command:

```
$ docker scan --version
Version:    v0.17.0
Git commit: 061fe0a
Provider:   Snyk (1.827.0)
```

Note you do not have to be logged in to your Docker account to use Snyk, but in this case, you can only scan a limited number of times. If you are logged in, on the other hand, you can do up to 200 scans per month for free. Anything beyond that number requires a paid account with Snyk.

Now, what are vulnerabilities, or more precisely CVEs, where CVE stands for Common Vulnerabilities and Exposures?

Common Vulnerabilities and Exposures is a database of publicly disclosed information security issues. Each entry in that database has a number, and that number uniquely identifies one vulnerability from the list. Specialists permanently curate this list and add new vulnerabilities or exposures as quickly as they are discovered.

We can now use specialized software such as Snyk to scan the various layers of our Docker images and uncover software libraries that are known to include such CVEs. If we discover that our image is vulnerable, then we can and should fix the problem by using a more recent version of the affected library. We then will have to rebuild the image.

But there we are not done. As mentioned previously, security specialists continue to discover new CVEs all the time. Thus, a software library that used to be clean all of a sudden may now be vulnerable due to a new exposed CVE.

This means that we need to make sure all our Docker images in use are scanned periodically and that we alert our security experts and our developers, such as that they can quickly address the problem and create a new version of the infected image.

Let us try to scan our `whoami` application that we were introducing in *Chapter 12, Enabling zero downtime deployments* and using in the previous sections of this chapter.

1. Make sure you have the `whoami` image in your local cache:

   ```
   $ docker image pull gnschenker/whoami:1.0
   ```

2. Scan the image for vulnerabilities:

   ```
   $ docker scan gnschenker/whoami:1.0
   ```

   ```
   docker Scan relies upon access to Snyk, a third party provider, do
   you consent to proceed using Snyk? (y/N)
   ```

   Answer with "y" to the preceding question

3. The result of the previous scan looks like this on the author's computer:

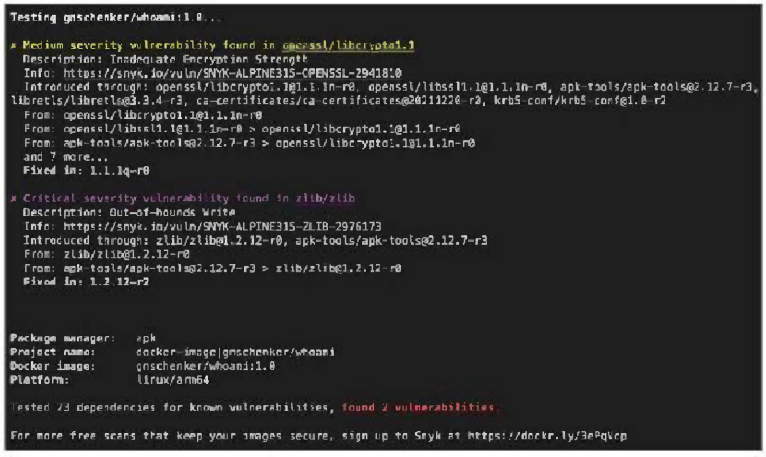


***Figure 13.4:** Scanning the gnschenker/whoami:1.0 Docker image*

We can now use specialized software such as Snyk to scan the various layers of our Docker images and uncover software libraries that are known to include such CVEs. If we discover that our image is vulnerable, then we can and should fix the problem by using a more recent version of the affected library. We then will have to rebuild the image.

But there we are not done. As mentioned previously, security specialists continue to discover new CVEs all the time. Thus, a software library that used to be clean all of a sudden may now be vulnerable due to a new exposed CVE.

This means that we need to make sure all our Docker images in use are scanned periodically and that we alert our security experts and our developers, such as that they can quickly address the problem and create a new version of the infected image.

Let us try to scan our `whoami` application that we were introducing in *Chapter 12, Enabling zero downtime deployments* and using in the previous sections of this chapter.

1. Make sure you have the `whoami` image in your local cache:

   ```
   $ docker image pull gnschenker/whoami:1.0
   ```

2. Scan the image for vulnerabilities:

   ```
   $ docker scan gnschenker/whoami:1.0
   ```

   ```
   docker Scan relies upon access to Snyk, a third party provider, do
   you consent to proceed using Snyk? (y/N)
   ```

   Answer with "y" to the preceding question

3. The result of the previous scan looks like this on the author's computer:

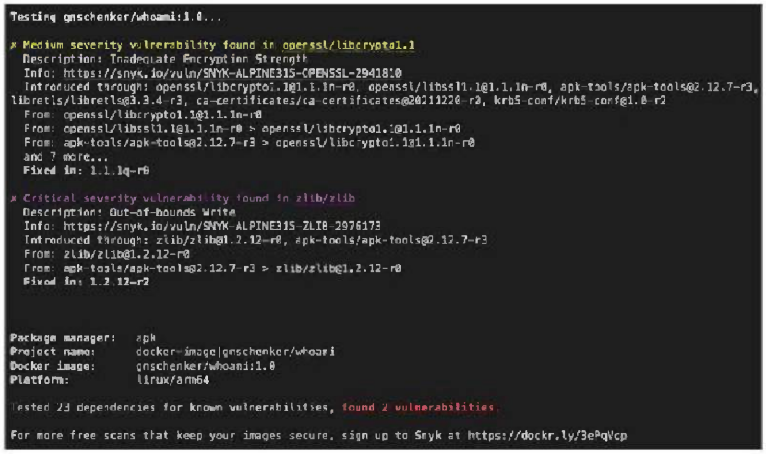


***Figure 13.4:** Scanning the gnschenker/whoami:1.0 Docker image*

As you can see, there were two vulnerabilities found in this version of the image, one of them of medium and one of critical severity. It is clear that we should address the critical vulnerability as soon as possible. Let us do this now.

1. First, we copy over the original **whoami** project, including the Docker file we used to build this image. You can find the copy in your **.../src/ch13/whoami** folder
2. Open the **Dockerfile** and inspect it. We were using version *6.0-alpine* for both the .NET SDK and the runtime. Let us see if Microsoft has updated the vulnerabilities in this version already...
3. Navigate to your **.../src/ch13/whoami** folder
4. Build a new version of the Docker image with the following command:

```
$ docker image build -t gnschenker/whoami:1.0.1 .
...
=> [runtime 2/3] WORKDIR /app                                    0.1s
 => [build 2/5] WORKDIR /app                                     0.5s
 => [build 3/5] COPY . .                                         0.0s
 => [build 4/5] RUN dotnet restore                               0.8s
  => [build 5/5] RUN dotnet publish -o /app/published-app
2.0s
  => [runtime 3/3] COPY --from=build /app/published-app /app
0.0s
 => exporting to image                                           0.0s
 => => exporting layers                                          0.0s
 => => writing image sha256:0c92aac77e1c511dc63c7a71b9ec3d482f1efb-
280219210f98312065cc232b9b   0.0s
 => => naming to docker.io/gnschenker/whoami:1.0.1               0.0s
```

   We omitted most of the output preceding for brevity.

5. Scan the new image:

```
$ docker scan gnschenker/whoami:1.0.1
```

As you can see, there were two vulnerabilities found in this version of the image, one of them of medium and one of critical severity. It is clear that we should address the critical vulnerability as soon as possible. Let us do this now.

1. First, we copy over the original **whoami** project, including the Docker file we used to build this image. You can find the copy in your **.../src/ch13/whoami** folder
2. Open the **Dockerfile** and inspect it. We were using version *6.0-alpine* for both the .NET SDK and the runtime. Let us see if Microsoft has updated the vulnerabilities in this version already...
3. Navigate to your **.../src/ch13/whoami** folder
4. Build a new version of the Docker image with the following command:

```
$ docker image build -t gnschenker/whoami:1.0.1 .
...
=> [runtime 2/3] WORKDIR /app                                    0.1s
 => [build 2/5] WORKDIR /app                                     0.5s
 => [build 3/5] COPY . .                                         0.0s
 => [build 4/5] RUN dotnet restore                               0.8s
  => [build 5/5] RUN dotnet publish -o /app/published-app
2.0s
  => [runtime 3/3] COPY --from=build /app/published-app /app
0.0s
 => exporting to image                                           0.0s
 => => exporting layers                                          0.0s
 => => writing image sha256:0c92aac77e1c511dc63c7a71b9ec3d482f1efb-
280219210f98312065cc232b9b   0.0s
 => => naming to docker.io/gnschenker/whoami:1.0.1               0.0s
```

   We omitted most of the output preceding for brevity.

5. Scan the new image:

```
$ docker scan gnschenker/whoami:1.0.1
```

6. This time the output should look like the following:

```
Testing gnschenker/whoami:1.0.1...

Package manager:   apk
Project name:      docker-image|gnschenker/whoami
Docker image:      gnschenker/whoami:1.0.1
Platform:          linux/arm64

✔ Tested 23 dependencies for known vulnerabilities, no vulnerable paths found.

For more free scans that keep your images secure, sign up to Snyk at https://dockr.ly/3ePqVcp
```

Figure 13.5: 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 it should be just fine, as the application itself did not change.

Firewalling apps running in containers

The best way to secure your applications from malicious attackers is to put them behind secure walls. This technique is also called firewalling an application. The principle is the same as in real life. You want to keep out bad actors from your property by securing the entrance to it. You may have different levels of security, such as door locks, security cameras, intrusion alerts, and more. Similarly, with software, we have a plethora of ways how we can secure our applications. The easiest one is to not connect the application to any external systems, such as the internet. But often, this is not very helpful as, in this case, you are severely crippling the availability of the application. Thus, we need to look out for other means to protect our software.

Software-defined networks

If running containers directly on a Docker host or in a Docker swarm cluster, then we can use **Software-Defined Networks (SDNs)** to limit the reach of applications that are running inside these containers. Only applications that are running on the same SDN can communicate with each other. Thus, if you have an application consisting of three parts, the `web` component, the `domain` component containing the domain logic, and the `database`, then you can define, e.g., two SDNs called `front` and `back`. If you run the `web` and the `domain` container on the `front` network and the `domain` and the `database` container on the `back` network, then the `web` component can only communicate with the `domain` component but not with the `database` component. Only the `domain` component can talk to both the `web` and the `database` component since it is connected to both the `front` and the `back` network. This makes it harder for attackers that succeeded to compromise the `web` component to attack your `database` and steal confidential information.

6. This time the output should look like the following:

```
Testing gnschenker/whoami:1.0.1...

Package manager:   apk
Project name:      docker-image|gnschenker/whoami
Docker image:      gnschenker/whoami:1.0.1
Platform:          linux/arm64

✔ Tested 23 dependencies for known vulnerabilities, no vulnerable paths found.

For more free scans that keep your images secure, sign up to Snyk at https://dockr.ly/3ePqVcp
```

Figure 13.5: 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 it should be just fine, as the application itself did not change.

Firewalling apps running in containers

The best way to secure your applications from malicious attackers is to put them behind secure walls. This technique is also called firewalling an application. The principle is the same as in real life. You want to keep out bad actors from your property by securing the entrance to it. You may have different levels of security, such as door locks, security cameras, intrusion alerts, and more. Similarly, with software, we have a plethora of ways how we can secure our applications. The easiest one is to not connect the application to any external systems, such as the internet. But often, this is not very helpful as, in this case, you are severely crippling the availability of the application. Thus, we need to look out for other means to protect our software.

Software-defined networks

If running containers directly on a Docker host or in a Docker swarm cluster, then we can use **Software-Defined Networks (SDNs)** to limit the reach of applications that are running inside these containers. Only applications that are running on the same SDN can communicate with each other. Thus, if you have an application consisting of three parts, the `web` component, the `domain` component containing the domain logic, and the `database`, then you can define, e.g., two SDNs called `front` and `back`. If you run the `web` and the `domain` container on the `front` network and the `domain` and the `database` container on the `back` network, then the `web` component can only communicate with the `domain` component but not with the `database` component. Only the `domain` component can talk to both the `web` and the `database` component since it is connected to both the `front` and the `back` network. This makes it harder for attackers that succeeded to compromise the `web` component to attack your `database` and steal confidential information.

Port and protocol mappings

Mapping ports of an application running in a container to the outside world, that is, to the host computer on which the application is running, is to open an access gate to the application. This is potentially dangerous, and malicious actors can try to sneak in through these doors. Thus we should carefully limit the number of ports we open and monitor and protect them. If we open a port, then we can limit who can access this door by limiting the protocol that can be used when communicating through this port. Examples are TCP and UDP protocol access only or both.

Here is an example of how to run a Docker container—in this case, an Nginx Web server —and open a port:

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

In the previous command, we are using the command line parameter `-p` (or `--port`) to open the container port 80 and map it to the host port 8080. Note that the container port is the port on which the application running inside the container is listening. We can show the mappings defined for the `web` container by using this command:

```
$ docker container ports web
80/tcp -> 0.0.0.0:8080
```

The output shows us that indeed the container port 80 is mapped to the host (indicated by the 0.0.0.0 IP address) port 8080. We also see that only the TCP protocol is allowed.

We can now try to access the Web server through host port 8080, as is easily proven by using the following command:

```
$ curl http://localhost:8080
```

The output of the preceding should be an HTML snippet representing the welcome page of Nginx.

Ingress and egress rules

A Kubernetes cluster defines a single network. Each pod running on a cluster can thus reach any other pod running on the same cluster. But wait for a second, isn't this dangerous? What if pod A runs a compromised piece of software? A malicious attacker could now try to attack all the other applications running on the cluster through pod A. We can avoid that or at least severely limit that possibility by defining ingress and egress rules for all our applications and application services running in the cluster.

Port and protocol mappings

Mapping ports of an application running in a container to the outside world, that is, to the host computer on which the application is running, is to open an access gate to the application. This is potentially dangerous, and malicious actors can try to sneak in through these doors. Thus we should carefully limit the number of ports we open and monitor and protect them. If we open a port, then we can limit who can access this door by limiting the protocol that can be used when communicating through this port. Examples are TCP and UDP protocol access only or both.

Here is an example of how to run a Docker container—in this case, an Nginx Web server —and open a port:

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

In the previous command, we are using the command line parameter `-p` (or `--port`) to open the container port 80 and map it to the host port 8080. Note that the container port is the port on which the application running inside the container is listening. We can show the mappings defined for the `web` container by using this command:

```
$ docker container ports web
80/tcp -> 0.0.0.0:8080
```

The output shows us that indeed the container port 80 is mapped to the host (indicated by the 0.0.0.0 IP address) port 8080. We also see that only the TCP protocol is allowed.

We can now try to access the Web server through host port 8080, as is easily proven by using the following command:

```
$ curl http://localhost:8080
```

The output of the preceding should be an HTML snippet representing the welcome page of Nginx.

Ingress and egress rules

A Kubernetes cluster defines a single network. Each pod running on a cluster can thus reach any other pod running on the same cluster. But wait for a second, isn't this dangerous? What if pod A runs a compromised piece of software? A malicious attacker could now try to attack all the other applications running on the cluster through pod A. We can avoid that or at least severely limit that possibility by defining ingress and egress rules for all our applications and application services running in the cluster.

The authors of the Calico Project write: From the point of view of a Kubernetes pod, ingress is incoming traffic to the pod, and egress is outgoing traffic from the pod. In Kubernetes network policy, you create ingress and egress allow rules independently (egress, ingress, or both).

You have just read that to define ingress or egress rules, we need to use policies on Kubernetes. We will now try to write some for our **whoami** service. Network policies apply to pods within a specific namespace. Policies can include one or more ingress rules. To specify which pods in the namespace the network policy applies to, we need to use a pod selector. Within the ingress rule, we then use another pod selector to define which pods allow incoming traffic and the ports field to define on which ports this incoming traffic is allowed.

Let us define a policy for an ingress rule for a fictitious API component represented in this example by an Nginx container. Here is the policy:

ch13 > ! ingress-policy.yaml > apiVersion

```
1  apiVersion: networking.k8s.io/v1
2  kind: NetworkPolicy
3  metadata:
4    name: my-ingress-policy
5  spec:
6    podSelector:
7      matchLabels:
8        role: api
9    policyTypes:
10     - Ingress
11   ingress:
12     - from:
13         - podSelector:
14             matchLabels:
15               role: web
16       ports:
17         - protocol: TCP
18           port: 80
```

Figure 13.6: Kubernetes network policy to restrict ingress to a pod

This policy, called **my-ingress-policy**, defines on Lines 6–8 to which pods it applies. In this case, it is valid for all pods in the namespace that have the label **role=api**. In Lines 9 to 10, we define what kind of policy this is. In our case, we define an ingress policy. Finally, in Lines 11 to 18, we define from where the ingress traffic can originate from. In our case, this is all pods in the namespace that have a label **role=web** and that access port 80 and use the TCP protocol.

Unfortunately, we cannot test this network policy easily on our Docker Desktop with Kubernetes, as the use of network policies is not enabled by default. **NetworkPolicy**

The authors of the Calico Project write: From the point of view of a Kubernetes pod, ingress is incoming traffic to the pod, and egress is outgoing traffic from the pod. In Kubernetes network policy, you create ingress and egress allow rules independently (egress, ingress, or both).

You have just read that to define ingress or egress rules, we need to use policies on Kubernetes. We will now try to write some for our **whoami** service. Network policies apply to pods within a specific namespace. Policies can include one or more ingress rules. To specify which pods in the namespace the network policy applies to, we need to use a pod selector. Within the ingress rule, we then use another pod selector to define which pods allow incoming traffic and the ports field to define on which ports this incoming traffic is allowed.

Let us define a policy for an ingress rule for a fictitious API component represented in this example by an Nginx container. Here is the policy:

ch13 > ! ingress-policy.yaml > apiVersion

```
1  apiVersion: networking.k8s.io/v1
2  kind: NetworkPolicy
3  metadata:
4    name: my-ingress-policy
5  spec:
6    podSelector:
7      matchLabels:
8        role: api
9    policyTypes:
10     - Ingress
11   ingress:
12     - from:
13         - podSelector:
14             matchLabels:
15               role: web
16       ports:
17         - protocol: TCP
18           port: 80
```

Figure 13.6: Kubernetes network policy to restrict ingress to a pod

This policy, called **my-ingress-policy**, defines on Lines 6–8 to which pods it applies. In this case, it is valid for all pods in the namespace that have the label **role=api**. In Lines 9 to 10, we define what kind of policy this is. In our case, we define an ingress policy. Finally, in Lines 11 to 18, we define from where the ingress traffic can originate from. In our case, this is all pods in the namespace that have a label **role=web** and that access port 80 and use the TCP protocol.

Unfortunately, we cannot test this network policy easily on our Docker Desktop with Kubernetes, as the use of network policies is not enabled by default. **NetworkPolicy**

resources need a controller to be enabled that enforces the policies after they have been declared. One way to install the required functionality is by installing a **Container Network Interface (CNI)** such as Calico. But this is outside of the scope of this book.

In the next section, we will discuss some of the Docker security best practices as they have been recommended by Docker experts.

Docker security best practices

Here is a short list of some of the most important security best practices when dealing with Docker containers and images. This list is based on the lived experience of container experts such as the famous Docker captains (**https://www.docker.com/community/captains**).

Check your images

Always scan any image that you intend to run in production for known **Common Vulnerabilities and Exposures (CVEs)**. We have explained how to do so earlier in this chapter. We can use a tool such as Snyk, which is included in Docker Desktop, for this. There exist other tools that are free or commercial.

Also, make sure to use **content trust** for images you intend to use in production.

Never run as privileged

You should strictly avoid to run any container in privileged mode. But as always, every rule has exceptions. Before you start working in privileged mode, make sure you understand how it works. This mode should be exclusively used for very special scenarios and where it is unavoidable. But keep the usage at a strict minimum and have each usage reviewed by the security specialists in your company.

Why is the privileged mode so dangerous? Well, it grants Docker container root capabilities to all devices on the host system. Running a container in privileged mode gives it the capabilities of its host machine.

If you are an admin and want to check if any of your containers run in privileged mode, then you can use the following command:

```
$ docker inspect --format='{{.HostConfig.Privileged}}' <container-id>
```

Replace <container-id> with the ID or name of the container you are looking at. If the answer to the preceding command is `true`, then the container is indeed running in privileged mode.

resources need a controller to be enabled that enforces the policies after they have been declared. One way to install the required functionality is by installing a **Container Network Interface (CNI)** such as Calico. But this is outside of the scope of this book.

In the next section, we will discuss some of the Docker security best practices as they have been recommended by Docker experts.

Docker security best practices

Here is a short list of some of the most important security best practices when dealing with Docker containers and images. This list is based on the lived experience of container experts such as the famous Docker captains (**https://www.docker.com/community/captains**).

Check your images

Always scan any image that you intend to run in production for known **Common Vulnerabilities and Exposures (CVEs)**. We have explained how to do so earlier in this chapter. We can use a tool such as Snyk, which is included in Docker Desktop, for this. There exist other tools that are free or commercial.

Also, make sure to use **content trust** for images you intend to use in production.

Never run as privileged

You should strictly avoid to run any container in privileged mode. But as always, every rule has exceptions. Before you start working in privileged mode, make sure you understand how it works. This mode should be exclusively used for very special scenarios and where it is unavoidable. But keep the usage at a strict minimum and have each usage reviewed by the security specialists in your company.

Why is the privileged mode so dangerous? Well, it grants Docker container root capabilities to all devices on the host system. Running a container in privileged mode gives it the capabilities of its host machine.

If you are an admin and want to check if any of your containers run in privileged mode, then you can use the following command:

```
$ docker inspect --format='{{.HostConfig.Privileged}}' <container-id>
```

Replace <container-id> with the ID or name of the container you are looking at. If the answer to the preceding command is `true`, then the container is indeed running in privileged mode.

Only allow read access to a root file system

Docker containers should be ephemeral and run mostly stateless. That is why you can limit a container's access to the root or host file system to be read-only. This way, the application running inside a container can in no way modify the underlying filesystem.

Read-only filesystem on Kubernetes

Let us say we want to run Nginx on Kubernetes in read-only mode; then, you can use the following `yaml` file with the special `securityContext` setting of `readOnlyRootFilesystem` set to true:

```
apiVersion: v1
kind: Pod
metadata:
  name: web
  labels:
    name: web
spec:
  containers:
  - name: web
    image: nginx:alpine
    securityContext:
      readOnlyRootFilesystem: true
    ports:
      - containerPort: 80
```

You can find the preceding yaml in file `.../src/ch13/web-server-error.yaml`. Run it and observe that it fails. Before you continue, please remove the pod called `web`.

If we set the filesystem to read-only, then, to fix the preceding error, we have to add support for a temporary filesystem instead (also called `tmpfs`). In Kubernetes, we use ephemeral volumes to achieve this. Assuming we want to provide two independent ephemeral volumes `/var/run` and `/var/cache/nginx` to the Web server Nginx, we can extend the `yaml` file like so:

```
apiVersion: v1
kind: Pod
metadata:
```

Only allow read access to a root file system

Docker containers should be ephemeral and run mostly stateless. That is why you can limit a container's access to the root or host file system to be read-only. This way, the application running inside a container can in no way modify the underlying filesystem.

Read-only filesystem on Kubernetes

Let us say we want to run Nginx on Kubernetes in read-only mode; then, you can use the following `yaml` file with the special `securityContext` setting of `readOnlyRootFilesystem` set to true:

```
apiVersion: v1
kind: Pod
metadata:
  name: web
  labels:
    name: web
spec:
  containers:
  - name: web
    image: nginx:alpine
    securityContext:
      readOnlyRootFilesystem: true
    ports:
      - containerPort: 80
```

You can find the preceding yaml in file `.../src/ch13/web-server-error.yaml`. Run it and observe that it fails. Before you continue, please remove the pod called `web`.

If we set the filesystem to read-only, then, to fix the preceding error, we have to add support for a temporary filesystem instead (also called `tmpfs`). In Kubernetes, we use ephemeral volumes to achieve this. Assuming we want to provide two independent ephemeral volumes `/var/run` and `/var/cache/nginx` to the Web server Nginx, we can extend the `yaml` file like so:

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: web
  labels:
    name: web
spec:
  containers:
    - name: web
      image: nginx:alpine
      securityContext:
        readOnlyRootFilesystem: true
      ports:
        - containerPort: 80
      volumeMounts:
        - mountPath: /var/run
          name: tmpfs-01
        - mountPath: /var/cache/nginx
          name: tmpfs-02
  volumes:
    - name: tmpfs-01
      emptyDir: {}
    - name: tmpfs-02
      emptyDir: {}
```

Please note how we define ephemeral volumes at the bottom of the preceding code snippet.

Let us try to run this pod and make sure it works and the Web server can be accessed:

1. Create a new file **web-server.yaml**, and add the code preceding snippet to it
2. Create the Web server on Kubernetes with the following:

   ```
   $ kubectl create -f web-server.yaml

   pod "web-server" created
   ```

3. Show the logs of the Web server to prove all is fine:

   ```
   $ kubectl logs web

   /docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will
   ```

```
  name: web
  labels:
    name: web
spec:
  containers:
    - name: web
      image: nginx:alpine
      securityContext:
        readOnlyRootFilesystem: true
      ports:
        - containerPort: 80
      volumeMounts:
        - mountPath: /var/run
          name: tmpfs-01
        - mountPath: /var/cache/nginx
          name: tmpfs-02
  volumes:
    - name: tmpfs-01
      emptyDir: {}
    - name: tmpfs-02
      emptyDir: {}
```

Please note how we define ephemeral volumes at the bottom of the preceding code snippet.

Let us try to run this pod and make sure it works and the Web server can be accessed:

1. Create a new file **web-server.yaml**, and add the code preceding snippet to it
2. Create the Web server on Kubernetes with the following:

   ```
   $ kubectl create -f web-server.yaml

   pod "web-server" created
   ```

3. Show the logs of the Web server to prove all is fine:

   ```
   $ kubectl logs web

   /docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will
   ```

```
attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-
entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-
on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: can not modify /etc/nginx/
conf.d/default.conf (read-only file system?)
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-
on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-
worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2022/07/28 09:39:22 [notice] 1#1: using the "epoll" event method
2022/07/28 09:39:22 [notice] 1#1: nginx/1.21.1
2022/07/28 09:39:22 [notice] 1#1: built by gcc 10.3.1 20210424
(Alpine 10.3.1_git20210424)
2022/07/28 09:39:22 [notice] 1#1: OS: Linux 5.10.104-linuxkit
2022/07/28 09:39:22 [notice] 1#1: getrlimit(RLIMIT_NOFILE):
1048576:1048576
2022/07/28 09:39:22 [notice] 1#1: start worker processes
2022/07/28 09:39:22 [notice] 1#1: start worker process 22
2022/07/28 09:39:22 [notice] 1#1: start worker process 23
2022/07/28 09:39:22 [notice] 1#1: start worker process 24
2022/07/28 09:39:22 [notice] 1#1: start worker process 25
```

4. Forward the Web server's port to 8080, so we can access Nginx from the host:

```
$ kubectl port-forward web 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

5. Open a new browser tab and navigate to **`http://localhost:8080`**, and you should be greeted by the Nginx welcome message.

6. When done and before leaving

   a. Stop the port forwarding by pressing **`CTRL-c`** in the terminal

   b. Delete the Web-server pod:

```
attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-
entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-
on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: can not modify /etc/nginx/
conf.d/default.conf (read-only file system?)
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-
on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-
worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2022/07/28 09:39:22 [notice] 1#1: using the "epoll" event method
2022/07/28 09:39:22 [notice] 1#1: nginx/1.21.1
2022/07/28 09:39:22 [notice] 1#1: built by gcc 10.3.1 20210424
(Alpine 10.3.1_git20210424)
2022/07/28 09:39:22 [notice] 1#1: OS: Linux 5.10.104-linuxkit
2022/07/28 09:39:22 [notice] 1#1: getrlimit(RLIMIT_NOFILE):
1048576:1048576
2022/07/28 09:39:22 [notice] 1#1: start worker processes
2022/07/28 09:39:22 [notice] 1#1: start worker process 22
2022/07/28 09:39:22 [notice] 1#1: start worker process 23
2022/07/28 09:39:22 [notice] 1#1: start worker process 24
2022/07/28 09:39:22 [notice] 1#1: start worker process 25
```

4. Forward the Web server's port to 8080, so we can access Nginx from the host:

```
$ kubectl port-forward web 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

5. Open a new browser tab and navigate to **`http://localhost:8080`**, and you should be greeted by the Nginx welcome message.

6. When done and before leaving

   a. Stop the port forwarding by pressing **`CTRL-c`** in the terminal

   b. Delete the Web-server pod:

```
$ kubectl delete pod/web
"pod/web" deleted
```

Now, let us look at how we can do the same when running a plain Docker container.

Running a plain Docker container

If you want to run a Docker container on a Docker host directly, in read-only mode, then use this command:

```
$ docker run --read-only -d nginx
```

Please note the `--read-only` parameter, which mounts the container's root file system in read-only mode.

Scan for vulnerabilities and secrets

Our containers are as safe as the software we are running inside them.

Scanning for vulnerabilities

To guarantee that our images are free of vulnerabilities, we need to perform a scan for known vulnerabilities and exposures. These are commonly named **Common Vulnerabilities and Exposures (CVEs)**. We can use many free or commercial tools to perform the scanning.

Type	Name	Remarks
Free	Clair	
Free	Trivy	
Free	Docker Bench for Security	
Commercial	Snyk	OSS and free option available
Commercial	Anchore	OSS and free option available
Commercial	JFrog Xray	
Commercial	Qualys	

Table 13.1: tools for scanning vulnerabilities

Scanning for secrets

We should not forget to scan our images for accidentally included secrets. Again there exist free and commercial tools for this task. Let us just name two options:

```
$ kubectl delete pod/web
"pod/web" deleted
```

Now, let us look at how we can do the same when running a plain Docker container.

Running a plain Docker container

If you want to run a Docker container on a Docker host directly, in read-only mode, then use this command:

```
$ docker run --read-only -d nginx
```

Please note the `--read-only` parameter, which mounts the container's root file system in read-only mode.

Scan for vulnerabilities and secrets

Our containers are as safe as the software we are running inside them.

Scanning for vulnerabilities

To guarantee that our images are free of vulnerabilities, we need to perform a scan for known vulnerabilities and exposures. These are commonly named **Common Vulnerabilities and Exposures (CVEs)**. We can use many free or commercial tools to perform the scanning.

Type	Name	Remarks
Free	Clair	
Free	Trivy	
Free	Docker Bench for Security	
Commercial	Snyk	OSS and free option available
Commercial	Anchore	OSS and free option available
Commercial	JFrog Xray	
Commercial	Qualys	

Table 13.1: tools for scanning vulnerabilities

Scanning for secrets

We should not forget to scan our images for accidentally included secrets. Again there exist free and commercial tools for this task. Let us just name two options:

Type	Name	Remarks
Commercial	ggshield	OSS and free option available
Free	SecretScanner	

Table 13.2: tools for scanning secrets

The following two recommendations mainly apply when you run Docker containers outside of Kubernetes, say in a Docker swarm or a standalone Docker host.

Do not expose the Docker daemon socket

Never expose the UNIX socket on which Docker is listening to the application running in a container, except if you are very confident with what you are doing. The Docker socket is defined at `/var/run/docker.sock`. It is the primary entry point for the Docker API. Giving the application access to this API is equivalent to giving unrestricted root access to the container host. Thus, in general, avoid commands as the following:

```
$ docker run … -v /var/run/docker.sock://var/run/docker.sock … <image>
```

Network namespace of the host

It is of utmost importance that you do not share the network namespace of the host, except in very specific circumstances. Docker includes the **host** network by default. Let us list all network types of a fresh Docker installation:

```
$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
f453076d3c2e   bridge    bridge    local
77b3da317100   host      host      local
93085f159974   none      null      local
```

As you can see, the second entry in the list corresponds to the mentioned **host** network. What kind of capabilities does the **host** network provide?

The **host** network allows an application running inside the container to access the full network stack of the host system. The container does not get its own IP address assigned. By using this network type, we bypass the added security layer that Docker provides to our applications. Our application is directly exposed to any potentially malicious application running on the host. On the other hand, the containerized application itself, if it has been compromised, can also attack all other processes running on the host.

Type	Name	Remarks
Commercial	ggshield	OSS and free option available
Free	SecretScanner	

Table 13.2: tools for scanning secrets

The following two recommendations mainly apply when you run Docker containers outside of Kubernetes, say in a Docker swarm or a standalone Docker host.

Do not expose the Docker daemon socket

Never expose the UNIX socket on which Docker is listening to the application running in a container, except if you are very confident with what you are doing. The Docker socket is defined at `/var/run/docker.sock`. It is the primary entry point for the Docker API. Giving the application access to this API is equivalent to giving unrestricted root access to the container host. Thus, in general, avoid commands as the following:

```
$ docker run … -v /var/run/docker.sock://var/run/docker.sock … <image>
```

Network namespace of the host

It is of utmost importance that you do not share the network namespace of the host, except in very specific circumstances. Docker includes the **host** network by default. Let us list all network types of a fresh Docker installation:

```
$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
f453076d3c2e   bridge    bridge    local
77b3da317100   host      host      local
93085f159974   none      null      local
```

As you can see, the second entry in the list corresponds to the mentioned **host** network. What kind of capabilities does the **host** network provide?

The **host** network allows an application running inside the container to access the full network stack of the host system. The container does not get its own IP address assigned. By using this network type, we bypass the added security layer that Docker provides to our applications. Our application is directly exposed to any potentially malicious application running on the host. On the other hand, the containerized application itself, if it has been compromised, can also attack all other processes running on the host.

Conclusion

In this chapter, we have shown you how to define secrets in a Kubernetes cluster and how to use those secrets in containers running as part of the pods of a deployment. We have shown two variants of how secrets can be mapped inside a container, the first using files and the second using environment variables. Next, we have learned how we can use content trust to further secure our container (image) delivery pipeline and eliminate possible man-in-the-middle attacks. Then, we investigated how we can scan our Docker images for known vulnerabilities and exposures, and we learned how to fix a vulnerable Docker image. Next, we have learned how to firewall applications running in containers on our Kubernetes clusters. Last but not least, we proposed a shortlist of security best practices when working with containers.

Job interview questions

You have been invited to a job interview for a position as a senior software developer at this new start-up that is on everybody's mind. The area the company is working in is very prone to cyber-attacks, and thus, security know-how is a major deciding factor if a candidate is deemed fit for the job.

1. Explain how the use of containers can help us to more tightly secure our applications.
2. We are using a fully automated CI/CD pipeline to build, test, ship and run our applications. Our container orchestrator is Kubernetes. Explain how you would setup secrets for an environment such as production using the pipeline. What steps are involved?
3. We have heard that Kubernetes uses a single network, and each pod running on this network can reach any other pod running in the same cluster and namespace. Is that not too dangerous? Could not a malicious application running inside a pod exploit the other pods? Or are there any ways to prevent that?
4. Imagine running an application on Kubernetes, and this application needs to access secrets such as username and password or certificates, and so on. What is the recommended way to provide the secrets to the application.
5. We have some legacy applications that we want to containerize for added security and faster release cycles. We do not want to change those applications as they are based on outdated technology in which we have no more expertise in our company. These applications have been running for years now without any hiccups. These applications read application configuration and secret values through environment variables. What can we

Conclusion

In this chapter, we have shown you how to define secrets in a Kubernetes cluster and how to use those secrets in containers running as part of the pods of a deployment. We have shown two variants of how secrets can be mapped inside a container, the first using files and the second using environment variables. Next, we have learned how we can use content trust to further secure our container (image) delivery pipeline and eliminate possible man-in-the-middle attacks. Then, we investigated how we can scan our Docker images for known vulnerabilities and exposures, and we learned how to fix a vulnerable Docker image. Next, we have learned how to firewall applications running in containers on our Kubernetes clusters. Last but not least, we proposed a shortlist of security best practices when working with containers.

Job interview questions

You have been invited to a job interview for a position as a senior software developer at this new start-up that is on everybody's mind. The area the company is working in is very prone to cyber-attacks, and thus, security know-how is a major deciding factor if a candidate is deemed fit for the job.

1. Explain how the use of containers can help us to more tightly secure our applications.
2. We are using a fully automated CI/CD pipeline to build, test, ship and run our applications. Our container orchestrator is Kubernetes. Explain how you would setup secrets for an environment such as production using the pipeline. What steps are involved?
3. We have heard that Kubernetes uses a single network, and each pod running on this network can reach any other pod running in the same cluster and namespace. Is that not too dangerous? Could not a malicious application running inside a pod exploit the other pods? Or are there any ways to prevent that?
4. Imagine running an application on Kubernetes, and this application needs to access secrets such as username and password or certificates, and so on. What is the recommended way to provide the secrets to the application.
5. We have some legacy applications that we want to containerize for added security and faster release cycles. We do not want to change those applications as they are based on outdated technology in which we have no more expertise in our company. These applications have been running for years now without any hiccups. These applications read application configuration and secret values through environment variables. What can we

do to make those applications work inside a container and run as a service on Kubernetes. Specifically, focus on dealing with secrets.

6. Explain to us why we should use content trust for all Docker images we use in production?

Possible answers to the job interview questions

Here are possible answers to the preceding questions:

1. Containers provide different layers of added security; here, we list the most important ones:
    a. Docker containers use the Linux namespaces and cgroups to tighten down the security and limit access to resources
    b. Containers or pods can be protected by network policies defining ingress and egress rules
    c. Port and protocol mapping helps us to better limit and control who and how an application inside a container can be accessed
    d. Content trust used with Docker images can eliminate so-called man-in-the-middle attacks
    e. Scanners can be used to discover common vulnerabilities and exposures in the various layers of the Docker images we use. These scanners also help to remedy the threats by defining possible solutions
2. Let us describe two possible approaches to the problem:
    a. Secrets can be defined declaratively in the production environment by an authorized system administrator. The pods can then use those secrets by either mapping them to their internal filesystem or to local environment variables. Both ways will make the secrets available to the code running inside the containers in clear text. As long as we limit access to the production cluster and to the pods running on it to only authorized personnel and log all of their activities, then this approach is secure. It has one flaw, though, human error is still possible. Even the most professional system administrator makes errors.
    b. An even better way that completely eliminates human error is the use of the pipeline to define the secrets in the production cluster. How could this possibly work? Usually, CI/CD services support role based access control. Thus we can define areas and tasks of the pipeline that can be only accessed by system administrators that own the secrets needed to access a secret store, such as HashiCorp's KeyVault or the

do to make those applications work inside a container and run as a service on Kubernetes. Specifically, focus on dealing with secrets.

6. Explain to us why we should use content trust for all Docker images we use in production?

Possible answers to the job interview questions

Here are possible answers to the preceding questions:

1. Containers provide different layers of added security; here, we list the most important ones:
    a. Docker containers use the Linux namespaces and cgroups to tighten down the security and limit access to resources
    b. Containers or pods can be protected by network policies defining ingress and egress rules
    c. Port and protocol mapping helps us to better limit and control who and how an application inside a container can be accessed
    d. Content trust used with Docker images can eliminate so-called man-in-the-middle attacks
    e. Scanners can be used to discover common vulnerabilities and exposures in the various layers of the Docker images we use. These scanners also help to remedy the threats by defining possible solutions
2. Let us describe two possible approaches to the problem:
    a. Secrets can be defined declaratively in the production environment by an authorized system administrator. The pods can then use those secrets by either mapping them to their internal filesystem or to local environment variables. Both ways will make the secrets available to the code running inside the containers in clear text. As long as we limit access to the production cluster and to the pods running on it to only authorized personnel and log all of their activities, then this approach is secure. It has one flaw, though, human error is still possible. Even the most professional system administrator makes errors.
    b. An even better way that completely eliminates human error is the use of the pipeline to define the secrets in the production cluster. How could this possibly work? Usually, CI/CD services support role based access control. Thus we can define areas and tasks of the pipeline that can be only accessed by system administrators that own the secrets needed to access a secret store, such as HashiCorp's KeyVault or the

AWS SecretManager. These administrators can then provide to the pipeline task the API key and API key secret to access the KeyVault or SecretManager. The task can then read all necessary secrets from this secret store and use them to create or update the Kubernetes secrets in the target production cluster. This way, everything is automated, a complete audit trail exists, and human error is avoided.

3. The described situation can be mitigated by the use of network policies that can be applied to all deployments, services and/or pods running on the Kubernetes cluster. We can define policies that limit ingress and egress for each pod. It is, e.g., possible to define that a pod A can only be reached from outside through a given port using a specified protocol such as UDP or TCP. We can even more limit INGRESS by defining the source from where the incoming traffic can originate. This can be, e.g., all pods that have a certain label with a defined value. Similar rules apply for EGRESS policies which define the limiting factors to the outgoing network traffic of a pod.

4. Secrets should be defined on the cluster on which the pods run. A Kubernetes secret consists of a single or a list of key-value pairs, where the key corresponds to the name of the secret value and the value contains the actual secret information. A secret defined on the Kubernetes cluster can be either mapped to environment variables defined inside the container of the pod or as files mapped to the internal filesystem of the container. In both cases, the secret values containing the sensitive information, such as a password or an API key secret, are available in clear text to the application running inside the container.

5. Any application, even very old ones, is usually able to use environment variables or read files from the local filesystem. The two preferred ways of mapping Kubernetes secrets into a pod or container are exactly using either environment variables or files. Thus we can containerize legacy applications without a problem and leverage all the advantages, particularly in the security domain, that containers provide.

6. The use of content trust helps us to avoid so-called **man-in-the-middle attacks (MITM)**. Those MITN attacks can happen when we ship a container image from a registry such as Docker Hub to our production cluster. Some malicious actors can try to intercept our network call and tamper with the transferred data. We can avoid this scenario if we use content trust, with means that the originator of the image digitally and cryptographically signs the image. If a bad actor tries to modify even a single bit of the image during the transmission of the image to the target location, the signature will be invalid, and the recipient can immediately recognize this fact and refuse the image.

AWS SecretManager. These administrators can then provide to the pipeline task the API key and API key secret to access the KeyVault or SecretManager. The task can then read all necessary secrets from this secret store and use them to create or update the Kubernetes secrets in the target production cluster. This way, everything is automated, a complete audit trail exists, and human error is avoided.

3. The described situation can be mitigated by the use of network policies that can be applied to all deployments, services and/or pods running on the Kubernetes cluster. We can define policies that limit ingress and egress for each pod. It is, e.g., possible to define that a pod A can only be reached from outside through a given port using a specified protocol such as UDP or TCP. We can even more limit INGRESS by defining the source from where the incoming traffic can originate. This can be, e.g., all pods that have a certain label with a defined value. Similar rules apply for EGRESS policies which define the limiting factors to the outgoing network traffic of a pod.

4. Secrets should be defined on the cluster on which the pods run. A Kubernetes secret consists of a single or a list of key-value pairs, where the key corresponds to the name of the secret value and the value contains the actual secret information. A secret defined on the Kubernetes cluster can be either mapped to environment variables defined inside the container of the pod or as files mapped to the internal filesystem of the container. In both cases, the secret values containing the sensitive information, such as a password or an API key secret, are available in clear text to the application running inside the container.

5. Any application, even very old ones, is usually able to use environment variables or read files from the local filesystem. The two preferred ways of mapping Kubernetes secrets into a pod or container are exactly using either environment variables or files. Thus we can containerize legacy applications without a problem and leverage all the advantages, particularly in the security domain, that containers provide.

6. The use of content trust helps us to avoid so-called **man-in-the-middle attacks (MITM)**. Those MITN attacks can happen when we ship a container image from a registry such as Docker Hub to our production cluster. Some malicious actors can try to intercept our network call and tamper with the transferred data. We can avoid this scenario if we use content trust, with means that the originator of the image digitally and cryptographically signs the image. If a bad actor tries to modify even a single bit of the image during the transmission of the image to the target location, the signature will be invalid, and the recipient can immediately recognize this fact and refuse the image.

Questions

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

1. Explain how to define the values of a Kubernetes secret. Are those values secure?
2. Name two ways how secrets can be used by pods running on Kubernetes.
3. Explain to an interested layperson what content trust for containers is?
4. What are CVEs?
5. How can we detect if our Docker images contain known vulnerabilities or exposures?
6. How can I tighten down the network security of my Kubernetes cluster?

Answers

Here are the answers to the previous questions:

1. Both the key or name of the secret and the value have to be base64 encoded. And no, those values are not secure, as base64 encoding can be reversed.
2. We can use the following two methods to map secrets into a pod
    a. Define a volume mount for the container that uses a well-defined secret as the backing volume. Well-defined meaning, the secret must exist on the cluster. Note the secret key and value will be mapped as files inside the target folder in the container. The file name corresponds to the key, and the content of the file will be the value in clear text.
    b. Use environment variable mapping in the container specification and refer to an existing secret in the cluster. Here key and value of a secret have to be mapped individually, each to a different environment variable valid inside the container. Note the values will be in clear text
3. Content trust means that the originator of the image signs the image using a cryptographic algorithm. The recipient of the image trusts the originator and can verify the signature and thus assert that the image has not been tampered with and, indeed, corresponds to the original image bit by bit. This way so, called man-in-the-middle attacks can be prevented.
4. CVE stands for "common vulnerabilities and exposures". This is a public database of well-known security vulnerabilities and exposures. Each record in the database has a unique number and a detailed description of what the

Questions

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

1. Explain how to define the values of a Kubernetes secret. Are those values secure?
2. Name two ways how secrets can be used by pods running on Kubernetes.
3. Explain to an interested layperson what content trust for containers is?
4. What are CVEs?
5. How can we detect if our Docker images contain known vulnerabilities or exposures?
6. How can I tighten down the network security of my Kubernetes cluster?

Answers

Here are the answers to the previous questions:

1. Both the key or name of the secret and the value have to be base64 encoded. And no, those values are not secure, as base64 encoding can be reversed.
2. We can use the following two methods to map secrets into a pod
    a. Define a volume mount for the container that uses a well-defined secret as the backing volume. Well-defined meaning, the secret must exist on the cluster. Note the secret key and value will be mapped as files inside the target folder in the container. The file name corresponds to the key, and the content of the file will be the value in clear text.
    b. Use environment variable mapping in the container specification and refer to an existing secret in the cluster. Here key and value of a secret have to be mapped individually, each to a different environment variable valid inside the container. Note the values will be in clear text
3. Content trust means that the originator of the image signs the image using a cryptographic algorithm. The recipient of the image trusts the originator and can verify the signature and thus assert that the image has not been tampered with and, indeed, corresponds to the original image bit by bit. This way so, called man-in-the-middle attacks can be prevented.
4. CVE stands for "common vulnerabilities and exposures". This is a public database of well-known security vulnerabilities and exposures. Each record in the database has a unique number and a detailed description of what the

threat is, which software library is affected by this CVE, the severity of the threat, and more.

5. There exists a number of utilities—called scanners—that we can use to scan the layers of our Docker images. The most popular one is Snyk, which is used by Docker Desktop. The scanner is parsing library by library of each layer and looks for matches compared to the CVE database. If found, then those matches are reported by the scanner, including the severity of the threat and possible remedies.

6. We can use namespaces and ingress and egress rules. The latter two are policies that we apply to the different pods running in our network. Such a policy can, e.g., be that pod A only accepts incoming traffic on port 8080 and only from pods that have a label L1.

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



threat is, which software library is affected by this CVE, the severity of the threat, and more.

5. There exists a number of utilities—called scanners—that we can use to scan the layers of our Docker images. The most popular one is Snyk, which is used by Docker Desktop. The scanner is parsing library by library of each layer and looks for matches compared to the CVE database. If found, then those matches are reported by the scanner, including the severity of the threat and possible remedies.

6. We can use namespaces and ingress and egress rules. The latter two are policies that we apply to the different pods running in our network. Such a policy can, e.g., be that pod A only accepts incoming traffic on port 8080 and only from pods that have a label L1.

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



Index

Symbols

.NET application
  debugging 237-241
  hot reloading 225, 226

A

ADD keyword 123
alerts
  defining 393, 394
Amazon AWS ECS 326
Amazon AWS Elastic Kubernetes Service 356, 357
  cluster, creating 358-360
Amazon's elastic container service (ECS) 326
Apache Mesos 325
application debugging, in container
  Java application 232-237
  .NET application 237-241
  Node JS application 241-244
  Python application 244-247
application key
  monitoring, Grafana used 388-393
architecture, Kubernetes 327-329
  cluster nodes 331
  context-based routing 354, 355
  master nodes 330
  pod life cycle 333-335
  pods 331-333
  pods, working with 335-341
  services 350-354
  worker nodes 331
Arithmetic Processing Unit (APU) 318
availability zones 316
AWS CodePipeline 295-297
AWS DevOps 295-297

Index

Symbols

.NET application
  debugging 237-241
  hot reloading 225, 226

A

ADD keyword 123
alerts
  defining 393, 394
Amazon AWS ECS 326
Amazon AWS Elastic Kubernetes Service 356, 357
  cluster, creating 358-360
Amazon's elastic container service (ECS) 326
Apache Mesos 325
application debugging, in container
  Java application 232-237
  .NET application 237-241
  Node JS application 241-244
  Python application 244-247
application key
  monitoring, Grafana used 388-393
architecture, Kubernetes 327-329
  cluster nodes 331
  context-based routing 354, 355
  master nodes 330
  pod life cycle 333-335
  pods 331-333
  pods, working with 335-341
  services 350-354
  worker nodes 331
Arithmetic Processing Unit (APU) 318
availability zones 316
AWS CodePipeline 295-297
AWS DevOps 295-297

Azure container service (ACS) 327
Azure DevOps 294, 295

B
blue-green deployments
  enabling 414-422
bridge network 155
  creating 158
  DNS name resolution 161, 162
  inspecting 156, 157
  network stack of container, analyzing 158-161

C
cAdvisor 383
  used, for monitoring Docker metrics 383-387
canary releases 422
  realizing 422-425
  with Gloo 426-434
career paths 14
CGroups 2
Chocolatey
  installing, on Windows 10 or 11 23-25
CI/CD pipeline 14
Classic Swarm 324
cloud-based SaaS, for CI/CD pipeline 293
  AWS DevOps 295-297
  Azure DevOps 294, 295
  GitLab 297-299
cloud computing 15
Cloud Native Computing Foundation (CNCF) 6, 323
CMD keyword 127
code editor
  selecting 33, 34
Common Vulnerabilities and Exposures (CVEs) 11, 457, 461
consultant 15
container 2
  additional process, running 70, 71
  app, configuring 105-107
  apps, firewalling 454
  discussing, in context of serverless 367, 368
  evolution 3, 4
  friction, reducing in development process 12
  hot reload, enabling 224
  images, scanning for vulnerabilities 451-454
  improved security 10, 11
  listing 56-58
  log, retrieving 67
  logs, displaying in Docker Desktop 68, 69
  logs, displaying in VS Code 69, 70
  microservice architecture, turbocharging 8-10
  multiple instances, running 53
  removing 58-60
  removing, in Docker Desktop 61-65
  removing, in VS Code 61-65
  running 49-52
  running, in detach mode 52
  running processes, listing 65, 66
  standard deployment model, providing 10
  standardize infrastructure 7, 8
  stopping 58-60
  stopping, in Docker Desktop 61-65

Azure container service (ACS) 327
Azure DevOps 294, 295

B
blue-green deployments
  enabling 414-422
bridge network 155
  creating 158
  DNS name resolution 161, 162
  inspecting 156, 157
  network stack of container, analyzing 158-161

C
cAdvisor 383
  used, for monitoring Docker metrics 383-387
canary releases 422
  realizing 422-425
  with Gloo 426-434
career paths 14
CGroups 2
Chocolatey
  installing, on Windows 10 or 11 23-25
CI/CD pipeline 14
Classic Swarm 324
cloud-based SaaS, for CI/CD pipeline 293
  AWS DevOps 295-297
  Azure DevOps 294, 295
  GitLab 297-299
cloud computing 15
Cloud Native Computing Foundation (CNCF) 6, 323
CMD keyword 127
code editor
  selecting 33, 34
Common Vulnerabilities and Exposures (CVEs) 11, 457, 461
consultant 15
container 2
  additional process, running 70, 71
  app, configuring 105-107
  apps, firewalling 454
  discussing, in context of serverless 367, 368
  evolution 3, 4
  friction, reducing in development process 12
  hot reload, enabling 224
  images, scanning for vulnerabilities 451-454
  improved security 10, 11
  listing 56-58
  log, retrieving 67
  logs, displaying in Docker Desktop 68, 69
  logs, displaying in VS Code 69, 70
  microservice architecture, turbocharging 8-10
  multiple instances, running 53
  removing 58-60
  removing, in Docker Desktop 61-65
  removing, in VS Code 61-65
  running 49-52
  running, in detach mode 52
  running processes, listing 65, 66
  standard deployment model, providing 10
  standardize infrastructure 7, 8
  stopping 58-60
  stopping, in Docker Desktop 61-65

stopping, in VS Code 61-65
trusting 450, 451
use cases 6, 7
container image 46
downloading 46-48
containerized app
instrumenting 252
container logs
collecting 379-382
container metrics
scraping, Prometheus used 383
Container Network Interface (CNI) 457
container network model 152-154
container network model (CNM) 154, 177
container ports
auto mapping ports 163-166
containerized REST API, creating 167
containerized REST API, running 167-169
elucidating 162, 163
explicitly mapping ports 166
context-based routing 354, 355
continuous deployment 294
COPY keyword 123
counter 376
custom Docker image
creating 114
cybersecurity 15

D

data definition language (DDL) 82
data engineering 15
data resource management (DRM) 82
declarative syntax 178
Deployment
working with 346-349
Deployments 341
desired state 312
developers 12
DevOps engineer 14
DevOps engineers 13
distributed application architecture 305, 306
cluster 304
network 304
node 304
patterns and best practices 307
port 305
service 305
VM 304
Docker
high-level architecture 4, 5
playing with 38-40
Docker Build Kit
leveraging 139, 140
Docker CLI 6
Docker compose
dependencies between services, defining 213-215
logging with 212
overrides, using 218-220
project name, defining 217
service, scaling 215-217
usages 212
Docker-compose file
authoring 183
multi-service application, preparing 183
writing 200-204
Docker-compose syntax 178

stopping, in VS Code 61-65
trusting 450, 451
use cases 6, 7
container image 46
downloading 46-48
containerized app
instrumenting 252
container logs
collecting 379-382
container metrics
scraping, Prometheus used 383
Container Network Interface (CNI) 457
container network model 152-154
container network model (CNM) 154, 177
container ports
auto mapping ports 163-166
containerized REST API, creating 167
containerized REST API, running 167-169
elucidating 162, 163
explicitly mapping ports 166
context-based routing 354, 355
continuous deployment 294
COPY keyword 123
counter 376
custom Docker image
creating 114
cybersecurity 15

D

data definition language (DDL) 82
data engineering 15
data resource management (DRM) 82
declarative syntax 178
Deployment
working with 346-349
Deployments 341
desired state 312
developers 12
DevOps engineer 14
DevOps engineers 13
distributed application architecture 305, 306
cluster 304
network 304
node 304
patterns and best practices 307
port 305
service 305
VM 304
Docker
high-level architecture 4, 5
playing with 38-40
Docker Build Kit
leveraging 139, 140
Docker CLI 6
Docker compose
dependencies between services, defining 213-215
logging with 212
overrides, using 218-220
project name, defining 217
service, scaling 215-217
usages 212
Docker-compose file
authoring 183
multi-service application, preparing 183
writing 200-204
Docker-compose syntax 178

build instructions, using 181
Docker compose file, inspecting 179-181
imperative syntax, versus declarative syntax 178
networks, using 181-183
Docker Content Trust (DCT) 439, 450
Docker Desktop 26
configuring 32, 33
for executing container commands 30, 31
installing, on MacOS 28-30
installing, on Windows 10 or 11 26-28
Dockerfiles
best practices 134
Docker host 4
Docker Hub
account, creating 141, 142
Docker image 94, 113
building 136-139
building, with GitHub actions 262
creating interactively 114-118
custom Docker image, creating 114
Dockerfile, authoring 118
importing 134, 135
listing 54-56
publishing 142
publishing, to organization 144-146
publishing, to personal account 142, 143
tagging 140, 141
Docker image build process 119-122
ADD keyword 123-125
CMD keyword 127-130
COPY keyword 123-125
Dockerfile best practices 134
ENTRYPOINT keyword 127-130
EXPOSE keyword 130, 131
FROM keyword 122, 123
multi-step Dockerfiles 131-134
RUN keyword 125, 126
WORKDIR keyword 126, 127
Docker metrics
monitoring, with cAdvisor 383-387
Docker security best practices 457
Docker daemon socket expose, avoiding 462
images, checking 457
network namespace, of host 462
plain Docker container, running 461
privileged mode, avoiding 457
read-only filesystem, on Kubernetes 458-461
root file system, read access 458
secrets, scanning 461, 462
vulnerabilities, scanning 461
Docker Swarm 324, 325
Docker volumes
creating 99
deleting 100
host folder, mounting into container 103, 104
inspecting 100
mounting, into container 101-103
using, with containers 98
domain name system (DNS) 309

E

ENTRYPOINT keyword 127
EXPOSE keyword 130, 131

build instructions, using 181
Docker compose file, inspecting 179-181
imperative syntax, versus declarative syntax 178
networks, using 181-183
Docker Content Trust (DCT) 439, 450
Docker Desktop 26
configuring 32, 33
for executing container commands 30, 31
installing, on MacOS 28-30
installing, on Windows 10 or 11 26-28
Dockerfiles
best practices 134
Docker host 4
Docker Hub
account, creating 141, 142
Docker image 94, 113
building 136-139
building, with GitHub actions 262
creating interactively 114-118
custom Docker image, creating 114
Dockerfile, authoring 118
importing 134, 135
listing 54-56
publishing 142
publishing, to organization 144-146
publishing, to personal account 142, 143
tagging 140, 141
Docker image build process 119-122
ADD keyword 123-125
CMD keyword 127-130
COPY keyword 123-125
Dockerfile best practices 134
ENTRYPOINT keyword 127-130
EXPOSE keyword 130, 131
FROM keyword 122, 123
multi-step Dockerfiles 131-134
RUN keyword 125, 126
WORKDIR keyword 126, 127
Docker metrics
monitoring, with cAdvisor 383-387
Docker security best practices 457
Docker daemon socket expose, avoiding 462
images, checking 457
network namespace, of host 462
plain Docker container, running 461
privileged mode, avoiding 457
read-only filesystem, on Kubernetes 458-461
root file system, read access 458
secrets, scanning 461, 462
vulnerabilities, scanning 461
Docker Swarm 324, 325
Docker volumes
creating 99
deleting 100
host folder, mounting into container 103, 104
inspecting 100
mounting, into container 101-103
using, with containers 98
domain name system (DNS) 309

E

ENTRYPOINT keyword 127
EXPOSE keyword 130, 131

F
firewalling apps, in container 454
  ingress and egress rules 455, 456
  port and protocol mappings 455
  Software-Defined Networks (SDNs) 454
FROM keyword 122, 123

G
gauge 376
general availability (GA) 324
GitHub 262
GitHub Actions 264
  CI pipeline, creating 271-276
  Docker image workflow, adding 269-271
  workflows 264-269
GitHub Actions, for Docker image 262
  application, preparing 262-264
GitLab 297-299
Gloo
  canary releases, enabling with 426-434
Google Cloud Platform (GCP) 364-367
Grafana
  for monitoring application key 388-393

H
histograms 377
Homebrew
  installing, on MacOS 25, 26
host network 169-171
hot reload, enabling in container 224
  .NET application, hot reloading 225, 226
  NodeJS application, hot reloading 226, 227
  ReactJS application, hot reloading 228-231

I
image layers model
  Docker copy-on-write mechanism 97, 98
  investigating 93
  layered filesystem 94, 95
  mutable container layer 96, 97
imperative syntax 178
individual services
  application, running 209-211
  building 204
  building, with Docker compose 208
  database, preparing 204-207
  images, pushing to registry 208, 209
  removing 211
  restarting 211
  running 204, 211
  stopping 211
Infrastructure as Code (IaC) 14
IngressController 354
integration test
  running 248-251
  writing 247
issue's root cause
  finding 394, 395
iTerm2 Version 3
  installing, for MacOS 37
IT operations (IT OPS) department 6

J
Java application
  debugging 232-237

F
firewalling apps, in container 454
  ingress and egress rules 455, 456
  port and protocol mappings 455
  Software-Defined Networks (SDNs) 454
FROM keyword 122, 123

G
gauge 376
general availability (GA) 324
GitHub 262
GitHub Actions 264
  CI pipeline, creating 271-276
  Docker image workflow, adding 269-271
  workflows 264-269
GitHub Actions, for Docker image 262
  application, preparing 262-264
GitLab 297-299
Gloo
  canary releases, enabling with 426-434
Google Cloud Platform (GCP) 364-367
Grafana
  for monitoring application key 388-393

H
histograms 377
Homebrew
  installing, on MacOS 25, 26
host network 169-171
hot reload, enabling in container 224
  .NET application, hot reloading 225, 226
  NodeJS application, hot reloading 226, 227
  ReactJS application, hot reloading 228-231

I
image layers model
  Docker copy-on-write mechanism 97, 98
  investigating 93
  layered filesystem 94, 95
  mutable container layer 96, 97
imperative syntax 178
individual services
  application, running 209-211
  building 204
  building, with Docker compose 208
  database, preparing 204-207
  images, pushing to registry 208, 209
  removing 211
  restarting 211
  running 204, 211
  stopping 211
Infrastructure as Code (IaC) 14
IngressController 354
integration test
  running 248-251
  writing 247
issue's root cause
  finding 394, 395
iTerm2 Version 3
  installing, for MacOS 37
IT operations (IT OPS) department 6

J
Java application
  debugging 232-237

Java application, instrumenting
application level logging 253-255
Spring Boot actuators, using 255, 256
Java Runtime Environment (JRE) 9
Jenkins 276
URL 276
Jenkins, for build pipeline
CI/CD pipeline, implementing 277, 278
CI pipeline, creating 278-290
continuous delivery (CD), adding to pipeline 290-293
using 276

K
Kubernetes 323
playing with 40, 41
Kubernetes cluster 455
Kubernetes secrets 440, 441
declaratively defining 441-443
environment variables, using 448-450
imperatively defining 443-445
mapping, to filesystem 446-448
using 440
using, in pod 445

L
load balancing 315

M
managed Kubernetes offerings
leveraging, in cloud 355, 356
man-in-the-middle (MITM) attack 11, 321, 450
Marathon 326
Mesos 326
Microsoft Azure Kubernetes Service 360-364
multi-service application
API service, in Java 183-190
API service, in .NET 5.x 190-196
preparing 183
Web service 197-200
Mutual Transport Layer Security (MTLS) 319

N
networks
bridge 154
host 154
none 154
overlay 154
Node JS application
debugging 241-244
hot reloading 226, 227
none network 171, 172
no-SQL database
MongoDB, accessing from different container 91
Python app, for accessing MongoDB 91, 92, 93
running locally 88, 89, 90

O
Open Systems Interconnection (OSI) model 309
operations engineers 14
orchestrators 312
Amazon AWS ECS 326
Apache Mesos 325
Docker Swarm 324, 325
Kubernetes 323, 324
Marathon 326

Java application, instrumenting
application level logging 253-255
Spring Boot actuators, using 255, 256
Java Runtime Environment (JRE) 9
Jenkins 276
URL 276
Jenkins, for build pipeline
CI/CD pipeline, implementing 277, 278
CI pipeline, creating 278-290
continuous delivery (CD), adding to pipeline 290-293
using 276

K
Kubernetes 323
playing with 40, 41
Kubernetes cluster 455
Kubernetes secrets 440, 441
declaratively defining 441-443
environment variables, using 448-450
imperatively defining 443-445
mapping, to filesystem 446-448
using 440
using, in pod 445

L
load balancing 315

M
managed Kubernetes offerings
leveraging, in cloud 355, 356
man-in-the-middle (MITM) attack 11, 321, 450
Marathon 326
Mesos 326
Microsoft Azure Kubernetes Service 360-364
multi-service application
API service, in Java 183-190
API service, in .NET 5.x 190-196
preparing 183
Web service 197-200
Mutual Transport Layer Security (MTLS) 319

N
networks
bridge 154
host 154
none 154
overlay 154
Node JS application
debugging 241-244
hot reloading 226, 227
none network 171, 172
no-SQL database
MongoDB, accessing from different container 91
Python app, for accessing MongoDB 91, 92, 93
running locally 88, 89, 90

O
Open Systems Interconnection (OSI) model 309
operations engineers 14
orchestrators 312
Amazon AWS ECS 326
Apache Mesos 325
Docker Swarm 324, 325
Kubernetes 323, 324
Marathon 326

Microsoft Azure ACS 327
orchestrator, tasks
affinity and location awareness, providing 318, 319
desired state, reconciling 312, 313
fluctuating load, dealing with 315, 316
introspection, facilitating 322, 323
replicated and global services 313
routing services, providing 314, 315
security 319
self-healing, facilitating 316, 317
service discovery, enabling 314
work, distributing by load balancing 315
zero-downtime deployments, enabling 317, 318

P
package manager
Chocolatey, installing on Windows 10 or 11 23
installing 22
patterns and best practices, distributed
application architecture
calls, routing between components 309
load balancing traffic 310
loosely coupling components 307
programming defensively 310, 311
redundancy, guaranteeing 311
service discovery, using 308, 309
stateless components, preferring 307, 308
Payment Service Provider (PSP) 379
personal identifiable information (PII) 319
platform monitoring 374
alerts 378, 379
centrally collection logs 375, 376
dashboards 377, 378
log entries 374, 375
metrics 376, 377
run books 379
PostgreSQL command line interface (CLI) 82
PostgreSQL database server
running 78-82
Prometheus
used, for scraping container metrics 383
Python application
debugging 244-247

Q
QA automation engineers 13
Quality Engineering (QE) department 6

R
React JS application
hot reloading 228-231
ReplicaSet 341, 342
working with 342-346
rollbacks
using 404-414
rolling updates
using 404-414
RUN keyword 125, 126

S
salaries 14
security, orchestrator
communication and cryptographic node identity, securing 319, 320

Microsoft Azure ACS 327
orchestrator, tasks
affinity and location awareness, providing 318, 319
desired state, reconciling 312, 313
fluctuating load, dealing with 315, 316
introspection, facilitating 322, 323
replicated and global services 313
routing services, providing 314, 315
security 319
self-healing, facilitating 316, 317
service discovery, enabling 314
work, distributing by load balancing 315
zero-downtime deployments, enabling 317, 318

P
package manager
Chocolatey, installing on Windows 10 or 11 23
installing 22
patterns and best practices, distributed
application architecture
calls, routing between components 309
load balancing traffic 310
loosely coupling components 307
programming defensively 310, 311
redundancy, guaranteeing 311
service discovery, using 308, 309
stateless components, preferring 307, 308
Payment Service Provider (PSP) 379
personal identifiable information (PII) 319
platform monitoring 374
alerts 378, 379
centrally collection logs 375, 376
dashboards 377, 378
log entries 374, 375
metrics 376, 377
run books 379
PostgreSQL command line interface (CLI) 82
PostgreSQL database server
running 78-82
Prometheus
used, for scraping container metrics 383
Python application
debugging 244-247

Q
QA automation engineers 13
Quality Engineering (QE) department 6

R
React JS application
hot reloading 228-231
ReplicaSet 341, 342
working with 342-346
rollbacks
using 404-414
rolling updates
using 404-414
RUN keyword 125, 126

S
salaries 14
security, orchestrator
communication and cryptographic node identity, securing 319, 320

content trust 321
networks and network policies, securing 320
reverse uptime 321, 322
role base access control (RBAC) 320
secrets 320, 321
SemaText
URL 379
Service Level Agreements (SLAs) 393
Service Level Objective (SLO) 393, 394
single-page application (SPA) 228
software-defined network (SDN) 154, 320, 454
software developer 14
Software Development Life Cycle (SDLC) 4, 404
solid-state drive (SSD) 318
sound versioning strategy
establishing 402-404
SQL database
accessing, from another container 84, 85
Node JS application, for accessing PostgreSQL 85-88
PostgreSQL database server 78-82
running locally 78
working with 82-84
summaries 377
SwarmKit 325
system reliability engineers (SREs) 10
systems administrator 14

T
TeamCity 276
tests
integration, running 248-251
integration, writing 247
unit test, running 248-251
unit test, writing 247
trivia container
working with 67, 68

U
unit test
running 248-251
writing 247
universal unique identifier (UUID) 306

V
Virtual Machines (VMs) 2, 7
versus Docker containers 2
VS Code
extensions, installing 35, 36
installing, on MacOS 34, 35
installing, on Windows 10 34

W
Windows Terminal
installing 36, 37
iTerm2 Version 3, installing for MacOS 37
WORKDIR keyword 126, 127

content trust 321
networks and network policies, securing 320
reverse uptime 321, 322
role base access control (RBAC) 320
secrets 320, 321
SemaText
URL 379
Service Level Agreements (SLAs) 393
Service Level Objective (SLO) 393, 394
single-page application (SPA) 228
software-defined network (SDN) 154, 320, 454
software developer 14
Software Development Life Cycle (SDLC) 4, 404
solid-state drive (SSD) 318
sound versioning strategy
establishing 402-404
SQL database
accessing, from another container 84, 85
Node JS application, for accessing PostgreSQL 85-88
PostgreSQL database server 78-82
running locally 78
working with 82-84
summaries 377
SwarmKit 325
system reliability engineers (SREs) 10
systems administrator 14

T
TeamCity 276
tests
integration, running 248-251
integration, writing 247
unit test, running 248-251
unit test, writing 247
trivia container
working with 67, 68

U
unit test
running 248-251
writing 247
universal unique identifier (UUID) 306

V
Virtual Machines (VMs) 2, 7
versus Docker containers 2
VS Code
extensions, installing 35, 36
installing, on MacOS 34, 35
installing, on Windows 10 34

W
Windows Terminal
installing 36, 37
iTerm2 Version 3, installing for MacOS 37
WORKDIR keyword 126, 127

Docker: Up and Running

DESCRIPTION

Containers are one of the disruptive technologies in IT that have fundamentally changed how software is build, shipped, and run today. If you want to pursue a career as a Software engineer or a DevOps professional, then this book is for you.

The book starts by introducing Docker and teaches you how to write and run commands in Docker. The book then explains how to create Docker files, images, and containers, and while doing so, you get a stronghold of Docker tools like Docker Images, Dockerfiles, and Docker Compose. The book will also help you learn how to work with existing container images and how to build, test, and ship your containers containing your applications. Furthermore, the book will help you to deploy and run your containerized applications on Kubernetes and in the cloud.

By the end of the book, you will be able to build and deploy enterprise applications with ease.

KEY FEATURES

- Get familiar and work with key components of Docker.
- Learn how to automate CI/CD pipeline using Docker and Jenkins.
- Uncover the top Docker interview questions to crack your next interview.

WHAT YOU WILL LEARN

- Learn how to test and debug containerized applications.
- Understand how container orchestration works in Kubernetes.
- Monitor your Docker container's log using Prometheus and Grafana.
- Deploy, update, and scale applications into a Kubernetes cluster using different strategies.
- Learn how to use Snyk to scan vulnerabilities in Docker.

WHO THIS BOOK IS FOR

This book is for System administrators, Software engineers, DevOps aspirants, Application engineers, and Application developers.

BPB PUBLICATIONS
www.bpbonline.com

ISBN 978-93-5551-788-3

Docker: Up and Running

DESCRIPTION

Containers are one of the disruptive technologies in IT that have fundamentally changed how software is build, shipped, and run today. If you want to pursue a career as a Software engineer or a DevOps professional, then this book is for you.

The book starts by introducing Docker and teaches you how to write and run commands in Docker. The book then explains how to create Docker files, images, and containers, and while doing so, you get a stronghold of Docker tools like Docker Images, Dockerfiles, and Docker Compose. The book will also help you learn how to work with existing container images and how to build, test, and ship your containers containing your applications. Furthermore, the book will help you to deploy and run your containerized applications on Kubernetes and in the cloud.

By the end of the book, you will be able to build and deploy enterprise applications with ease.

KEY FEATURES

- Get familiar and work with key components of Docker.
- Learn how to automate CI/CD pipeline using Docker and Jenkins.
- Uncover the top Docker interview questions to crack your next interview.

WHAT YOU WILL LEARN

- Learn how to test and debug containerized applications.
- Understand how container orchestration works in Kubernetes.
- Monitor your Docker container's log using Prometheus and Grafana.
- Deploy, update, and scale applications into a Kubernetes cluster using different strategies.
- Learn how to use Snyk to scan vulnerabilities in Docker.

WHO THIS BOOK IS FOR

This book is for System administrators, Software engineers, DevOps aspirants, Application engineers, and Application developers.

BPB PUBLICATIONS
www.bpbonline.com

ISBN 978-93-5551-788-3

Document Outline

- Start